EXTENSIONS OF JADE AND JXTA
FOR IMPLEMENTING A DISTRIBUTED SYSTEM

by

Edward Kuan-Hua Chen

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

School of Engineering Science

© Edward Kuan-Hua Chen 2005

SIMON FRASER UNIVERSITY

Spring 2005

All rights reserved. This work may not be reproduced in whole or in part,
by photocopy or other means, without the permission of the author.

APPROVAL

NAME: Edward Kuan-Hua Chen

DEGREE: Master of Applied Science

TITLE OF THESIS: Extensions of JADE and JXTA for Implementing a Distributed System

EXAMINING COMMITTEE

Chair: **John Jones**
Professor, School of Engineering Science

William A. Gruver
Academic Supervisor
Professor, School of Engineering Science

Dorian Sabaz
Technical Supervisor
Chief Technology Officer
Intelligent Robotics Corporation

Shaohong Wu
External Examiner
National Research Council

Date Approved: April 8, 2004

SIMON FRASER UNIVERSITY

PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

W. A. C. Bennett Library
Simon Fraser University
Burnaby, BC, Canada

ABSTRACT

Distributed systems offer a useful approach for resolving critical networking limitations that result from the use of centralized topologies. Currently available distributed software platforms, however, have limitations that can limit their usefulness.

This thesis examines the architectures of two distributed software platforms, JADE and JXTA, and compares their strengths and weaknesses. It is shown that JADE is a superior platform in terms of efficiency and latency, mainly due to the partially centralized approach of its Agent Management System. On the other hand, the decentralized management system and unrestricted scalability of JXTA has the advantage that it is not critically dependent on any node.

ACKNOWLEDGEMENTS

I would like to thank William A. Gruver and Dorian Sabaz for their guidance and support throughout the course of this thesis. I am grateful for their continuous and unwavering support.

I would also like to thank John Jones of the School of Engineering Science, Simon Fraser University, and Shaohong Wu of the National Research Council of Canada.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

ACC	Agent Communication Channel
ACL	Agent Communication Language
AMS	Agent Management System
API	Application Program Interface
DF	Directory Facilitator
FIPA	Foundation for Intelligent Physical Agents
GAMS	Global Agent Management System
GDF	Global Directory Facilitator
GPA	Global Peer Administration
GPM	Global Peer Monitoring
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
JADE	Java Agent DEvelopment framework
JVM	Java Virtual Machine
JXTA	Juxtapose Project begun by Sun Microsystems
LAN	Local Area Network
OMG	Object Management Group
ORB	Object Request Broker
RMI	Remote Method Invocation
RTT	Round Trip Time

SDK	(Java) Standard Development Kit
SFU	Simon Fraser University
VNET	Virtual Network Project
WACC	Wireless Agent Communication Channel
WDS	Wireless Distributed System
WPP	Wireless Peer Pipes

1 INTRODUCTION

1.1 Limitations of Centralized Networks

With the explosive growth of networks, there exists a critical need to deliver information in a robust and efficient manner. Although applications such as the Internet were built on the vision of a completely decentralized network that allowed unlimited scalability [14], the reality is that most systems today are still built on the client-server concept.

In a centralized system, all functions and information are contained within a server with clients connecting directly to the server to send and receive information, as illustrated in Figure 1.

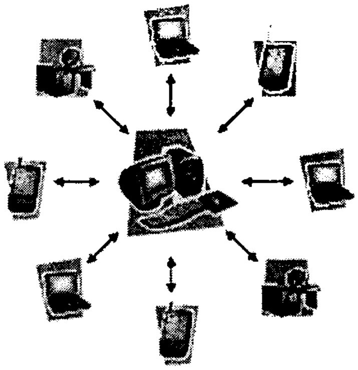


Figure 1. Traditional Client-Server Topology

Typically there are three key requirements for a central server: large data storage, significant processing power, and continuous reliable communication between the server and its clients [24]. Most applications, file and database servers systems are implemented with this kind of centralized topology [8].

However, as the network continues to grow, this traditional topology is inadequate to meet the demand of its users. The heavy emphasis on a central server places an undue burden on the network. As a centralized network expands, issues of scalability, fault-tolerance, security and infrastructure cost will hinder its growth.

1.1.1 Scalability

Centralized topologies are useful when the number of clients is unlikely to increase significantly. A server only has a finite processing capacity before a request is either lost or rejected. Since a server can only accommodate a fixed number of clients at a given time, it will need to allocate resources that would otherwise remain idle to accommodate the "bursty" nature of network traffic. Network resources are not utilized to their full potentials, thus creating areas of network congestion while other resources are idle [8].

1.1.2 Fault Tolerance

All critical data and information is stored at a central location, the server. The success or failure of the entire system is critically dependent on the reliable and consistent operation of the server.

As illustrated in Figure 2, the failure of a central server will have a catastrophic effect on the entire network. All exchanges of information between the server and client will stop. In practise, secondary servers are usually in place to avoid a complete shutdown. They are usually redundant systems that remain idle the majority of time.

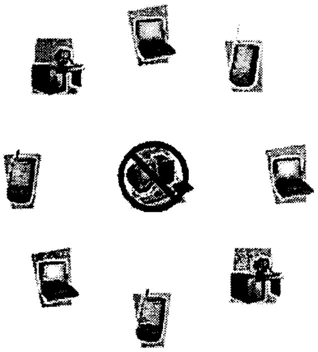


FIGURE 2. CATASTROPHIC SYSTEM FAILURE

A robust system should *not* have a *single-point of failure* that will have a catastrophic consequence on the system.

1.1.3 Security and Privacy

Since all critical data is stored at a central server, the privacy of all clients may be at risk when the security is compromised. By gaining access to the server alone, individuals are able to access information of the entire system, including information private to each client such as credit card numbers, bank accounts and medical files.

1.1.4 Connectivity

Currently, centralized topologies are usually implemented by wireline for which fibre-optic cables, twisted pairs and coaxial cable are the most commonly used medium. Users usually do not have the physical capacity to roam freely within the network and are limited by the physical topology of this infrastructure. The need for wireless connectivity has resulted in the standardization of the wireless protocol, IEEE 802.11. Users are now able to roam freely within a wireless LAN by communicating with access points in the LAN and no longer physically constrained to their desks.

Although the establishment of the IEEE 802.11 standard is a step in the right direction, its implementation is generally sbased on a centralized topology . In a typical wireless LAN environment, illustrated in Figure 3, clients utilize access points in networks to connect with other clients. Information is first sent from a sender to the Access Point and is then forwarded to the receiver. This approach still retains deficiencies of centralized systems, e.g., the failure of access points will have a catastrophic effect on the overall network.

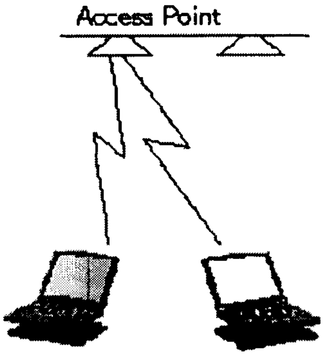


FIGURE 3. WIRELESS LOCAL AREA NETWORK (LAN)

The 802.11 standard does allow a form of distributed connectivity, called Ad-Hoc Mode. However, it only provides point-to-point communication, rather than multi-point-to-multi-point communication, as illustrated in Figure 4.

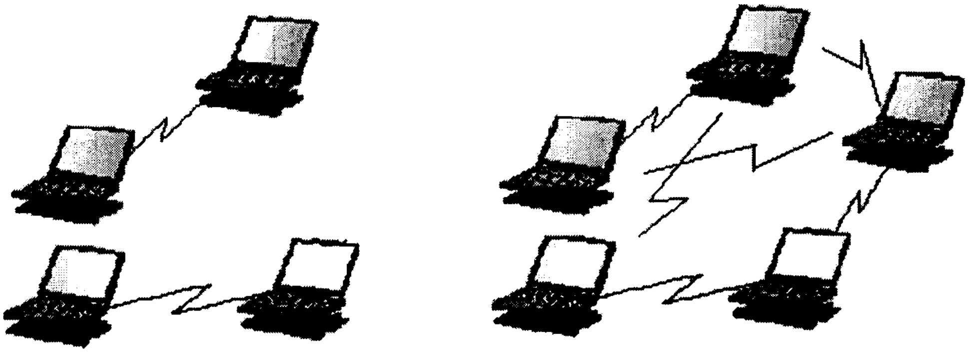


FIGURE 4. SINGLE-POINT VS MULTI-POINT COMMUNICATION

We would like to combine the IEEE 802.11 standard with the functionality of a distributed system environment. Many issues in the wireline centralized approach can be resolved using a decentralized architecture. The resulting system would be the basis of a distributed system that functions in a wireline or wireless environment.

1.1.5 Infrastructure Cost

The expansion of a wireline network has always been partially limited by the cost of additional infrastructures. Fibre optics cables are often used to interconnect two locations and the material and labour cost of switches and routers has restricted the growth of network in rural areas. Also, the time required to complete such an expansion can hinder the growth of the network.

1.2 Distributed Systems

In the last section we saw that a client-server topology has limitations in the areas of scalability, security, connectivity and infrastructure cost. This topology is unable to keep pace with the explosive growth of modern networks. Another approach that has been gaining interest is a Distributed Topology.

A Distributed System is a network topology that decentralizes the system so that no node has a greater central role than any other node. This topology fulfills the need for a robust, open-ended and highly scalable system by eliminating the central server and efficiently utilizes network resources [8]. Network resources are allocated across the network to alleviate computational bottlenecks within a single node or network area.

The Internet is an example of a Distributed System. Initially the Internet was designed to be a robust system with unrestricted scalability [13]. In reality, however, it is still reliant on localized web servers for database and file storage. Also, heavy emphasis is placed on routers that interconnect multiple networks. If the servers and routers fail, the LAN will be unable to communicate with other networks on the Internet. Issues related to a centralized topology are still prevalent with the current Internet.

In a fully distributed system, all nodes on the network are of equal significance, the failure of one node should not have a catastrophic effect on any other node on the network. A fully Distributed System has the potential to enhance system efficiency, reliability, extensibility and flexibility [8].

Some of the characteristics and advantages of Distributed Systems are now discussed.

1.2.1 Distributed System Privacy and Security

Unlike a centralized system, a distributed system lacks a central server for storage of critical information. The information is spread among nodes and is retrieved only at the demand of the requesting node. When the security of any node is compromised, the breach is localized and has no detrimental effect.

In addition, a message sent between nodes can be *packetized* to enhance security. It can be broken down into multiple data-packets, each containing a portion of the original message. The different data-packets can be sent through different paths to reach their destination. The receiver node will then re-arrange the packets to obtain the original information. This method ensures that no node except the receiver has complete access to the message, but can only route it onto the receiver.

1.2.2 Distributed System Fault Tolerance

Centralized systems have a *single-point* of failure. Centralized topologies are dependent on reliable performance of the servers and the consistent operation of communications between the servers and their clients. When a failure does occur to a server, all activities within the network cease. However, when a node fails in a distributed environment, information is simply routed around the failed node and continues its path to the receiver node. The distributed system will maintain its functionalities as long as there is an alternate path available.

1.2.3 Distributed System Scalability

Unlike a centralized system that utilizes a central server to process incoming data from all clients, nodes in a fully distributed system communicate directly among themselves.

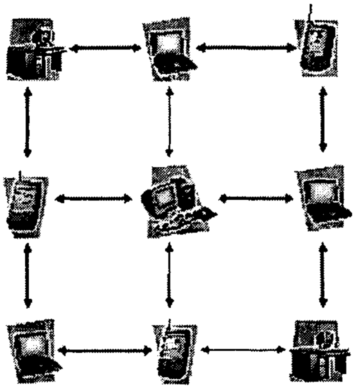


FIGURE 5: DISTRIBUTED SYSTEM TOPOLOGY

Requests for information and the actual transfer of information are performed locally between individual nodes. This eliminates the need for a powerful server and thus provides enhanced scalability as opposed to a client-server topology. Additional nodes are able to freely join the network without incurring computational burden on the system.

Each additional node that joins is also an additional resource for the network to utilize to ensure that the overall network remains efficient and robust.

1.2.4 Distributed System Connectivity

Nodes themselves may sometimes act as relays between two nodes if the sender and the receiver nodes cannot communicate directly. Different transmission paths can be formed from the sender to the receiver, thus ensuring the robustness of the distributed system in the event of the failure of a node. Nodes cooperate and collaborate with neighbour nodes to decide the most efficient path for message transmission. Nodes will use routing algorithms to direct traffic away from congested areas of the network and improve overall network efficiency robustness.

1.2.5 Distributed System Infrastructure Cost

In a wireless Distributed System, nodes communicate through wireless protocols. There are no fibre optics to implement and the amount of time and labour needed is far less when compared to a wireline system. Nodes are no longer physically limited to a geographic location; they are now able to roam freely within the boundaries of the wireless LAN.

1.2.6 Implementation Issues

A wireless application that utilizes a Distributed System is *Automated Meter Reading*. There is a need for utility companies to avoid the slow and expensive manual process of meter reading by automatically monitoring and acquiring utility meter from each customer location in real time.

A solution to this problem has been proposed by Sabaz, et al. [26]. In Figure 6, *Intelligent Wireless MicroRouters* are located at each house and these devices self organize to form a distributed network. Due to the overlapping coverage of the devices, meter data can be passed from one device to another, thus eliminating the need for a dedicated RF system or wireline system. Each *Intelligent Wireless MicroRouter* can only communicate with others in their area of coverage, and distributed intelligence software enable multiple *Intelligent Wireless MicroRouters* to perform negotiations that determine the best path for sending information to the collectors, as shown in Figure 6.

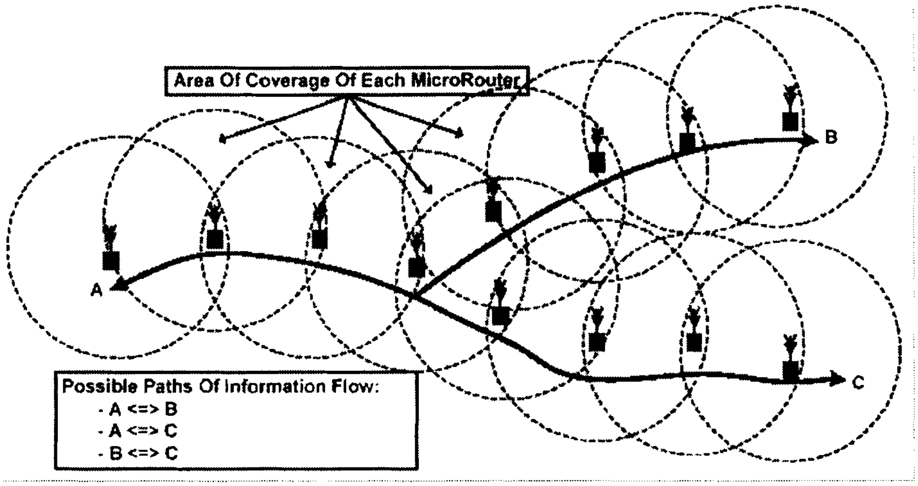


FIGURE 6. INTELLIGENT WIRELESS MICROROUTERS FOR AUTOMATED METER READING [1]

1.3 Distributed Computing Models and Architectures

Although considerable research has been devoted to the transformation of client-server topologies into a distributed topology, there remain many unresolved issues.

Traditionally, applications were designed for a single host operating within a single-address space utilizing a single operating system [14]. With the increasing growth of networks, applications now have to interact with other components on the network in a dynamic yet robust way [14]. However, there still exist fundamental issues with the implementation of distributed application programming environments:

> ***Address Space:*** Techniques to explicitly distinguish between local and remote objects and to handle remote interactions.
>
> ***Network Dimension:*** Handling of variance in hardware, software and operating systems within the network
>
> ***Programming-related:*** Handling of variance in programming language implementation
>
> ***Infrastructure-related:*** Distributed architectures defining their own protocols for processing method parameter and return values, e.g., IIOP for CORBA, JRMP for RMI and ORPC for DCOM.

Source: Bellifemine, et al.[12]

Distributed computing architectures have been developed over the years to handle these distributed computing issues. Three architectures are briefly described and compared in the following subsections.

1.3.1 Common Object Request Broker Architecture (CORBA)

CORBA is an architecture and specification for creating, distributing, and managing distributed program objects in a network. CORBA allows programs at different locations and developed by different vendors to communicate in a network through its "interface broker." CORBA was developed by OMG (Object Management Group) and is sanctioned by both ISO and X/Open as the standard architecture for distributed objects.

1.3.2 Distributed Component Object Model (DCOM)

DCOM is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner. Previously called "Network OLE," DCOM is designed for use across multiple network transports, including Internet protocols such as HTTP. DCOM is based on the Open Software Foundation DCE-RPC specification, and operates with both Java applets and Microsoft ActiveX components through its use of the Component Object Model (COM).

1.3.3 Remote Method Invocation (RMI)

RMI is a set of protocols that enable Java objects to communicate remotely with other Java objects. RMI is a relatively simple protocol, but unlike more complex protocols such as CORBA and DCOM, it works only with Java objects. CORBA and DCOM are designed to support objects created in any language.

Table 1 briefly compares these three distributed computing techniques with respect to the issues described above.

Table 1. Comparison of distributed computing techniques

	CORBA	**DCOM**	**RMI**
Address Space Issue (Calling remote hosts)	No explicit distinction from local and remote objects	No explicit distinction from local and remote objects	No explicit distinction from local and remote objects
Network Dimension Issue (Variance in soft/hardware and OS)	ORB layer handles data and call format conversions	ORPC layer handles data and call format conversions	No conversion necessary. Strictly JVM-JVM communication
Programming Language Related Issues	Problems with inter-ORB compatibility	Uses C, C++ and VB as programming language	Uses Java and is a Java-to-Java solution. Objects explicitly categorized as local or remote
Infrastructure Related Issues	Strong dependence on Internet Inter Orb Protocol (IIOP)	Strong dependence on ORPC	Strong dependence on JRMP

Source: Li[15]

However, neither of the currently available distributed applications provide a complete solution. There is a need for a better distributed architecture to function better, not just in wireline networks, but especially in wireless distributed systems. This is particularly vital as current and future networking implementations will require a distributed wireless system environment.

1.3.4 Distributed Application Development

Although there is much interest in distributed system applications, the complexity of building them has hindered development.

Many organizations are developing distributed software platforms to facilitate the development of distributed topologies. The platforms hide some of the intricacies of a distributed environment and allow developers to concentrate their efforts on the higher-level design of the system, rather than the low level communication transport. Examples of distributed software platforms include *JADE [3], FIPA-OS [6], JXTA [4]* and *JACK [7]*.

1.4 Overview

1.4.1 Objective

This thesis will discuss the architecture and extensions needed for two distributed software platforms, JADE and JXTA, to facilitate the development of distributed systems. We shall examine the architectural characteristics of both platforms, outlining their strengths and weaknesses. Then we shall examine the architectural extensions needed to improve the current platform. Quantitative and qualitative results will be given for both platforms.

1.4.2 Outline

Chapter 1 provides a brief overview of this thesis and suggests potential flaws in current centralized networks. It also provides a brief introduction to distributed systems and their advantages.

Chapter 2 briefly outlines the distributed software platforms available today and describes in detail the architecture of JADE and JXTA that are modeled in this thesis to facilitate the development of distributed systems.

Chapter 3 discusses the different architectural extensions required by each platform for an improved Distributed System. Conceptual details are presented along with an outline of the implementation approach.

Chapter 4 provides an analysis of the extensions implemented for the two software platforms. Example software listings and classes are presented.

Chapter 5 provides the qualitative and quantitative analysis of the JADE and JXTA platforms with the proposed extensions. A summary of this research is provided with directions for future research.

2 DISTRIBUTED SOFTWARE PLATFORMS

Centralized architectures are inherently more focused on simplicity, rather than on scalability and robustness, whereas a distributed system depends on a network that is scalable, robust and relatively inexpensive to maintain. However, the complexity of software implementation for a distributed system is greater than that for a centralized system. As the number of nodes within a distributed system increases, the inherent combinatorial nature of the network becomes exponentially more complex. Current distributed computing techniques do not provide a complete solution to handle distributed computing issues.

Presently, the potential strength that a distributed system may offer has focused research attention to develop software platforms that facilitate the implementation of a distributed system over a wireline network. Table 2 illustrates some of the distributed software platforms and their vendors.

TABLE 2. DISTRIBUTED SOFTWARE PLATFORMS AND VENDORS

Software Platform	**Vendor**
JADE [3]	Telecom Italia
JXTA [4]	Sun MicroSystems Inc
FIPA-OS [7]	Nortel Networks
JACK [6]	Agent Oriented Software Group
Grasshopper [22]	GMD FOKUS
Zeus [23]	BT Intelligent Agent Research
Agent Development Kit [24]	MADKIT Project

In this thesis, we concentrate on Java Agent Development Framework (JADE) and JXTA.. Both platforms are based upon Java, taking advantage of the native utility for interoperability. JADE and JXTA are built to handle infrastructure issues. Protocols and classes are abstracted to provide software developers with ease in implementing a distributed system. The platforms serve as middleware that deals with communication transport and message encoding. Software developers can therefore concentrate on the development of complex models and reasoning that constitute the distributed system, rather than on the low-level communication protocols.

Because of these features [12] [15] [17] and their research and commercial interest, JADE and JXTA were chosen for this thesis.

2.1 JADE Overview

JADE is an open source software platform developed by Telecom Italia Labs implemented in the Java language to simplify the development of a distributed system. It is in compliance with the Foundation for Intelligent Physical Agent (FIPA) specifications to ensure standard compliance through a set of system services and agents. FIPA is an international non-profit organization established in 1996 to produce standards for the interoperation of agents and agent-based systems [5].

JADE is composed of two core components: a platform that allows developers to create FIPA-compliant agent-based systems, and a Java package to develop software agents for inter-platform and intra-platform communication between agents, as illustrated in Figure 7

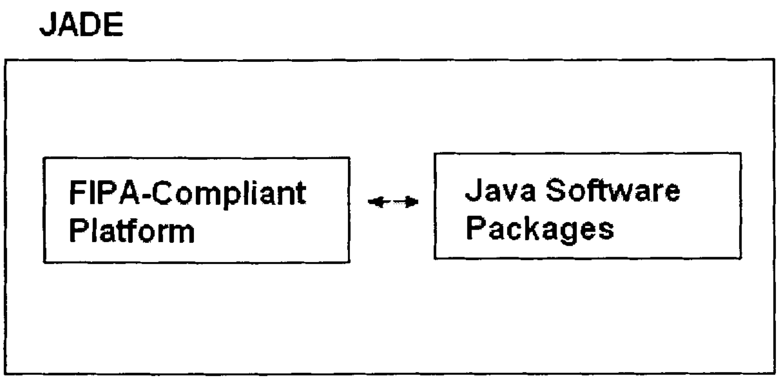


FIGURE 7. JADE COMPONENTS

2.1.1 JADE Agent Platform

JADE's communication system is based upon FIPA standards. There are three agents that must be present in a FIPA compliant agent platform, as illustrated in Figure 8 and described as follows:

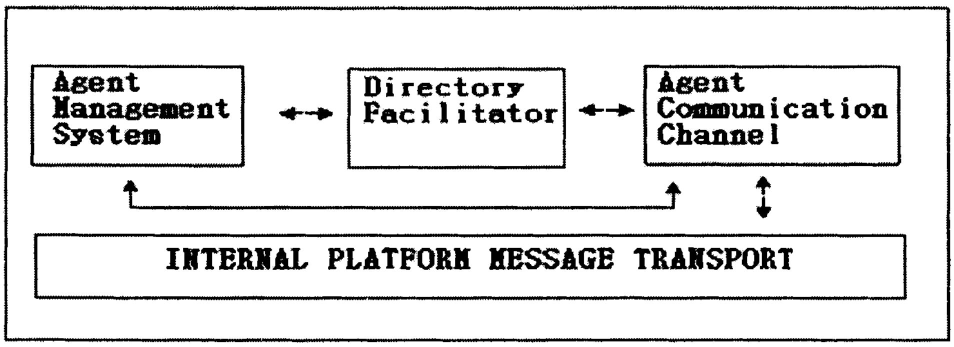


FIGURE 8. FIPA COMMUNICATION FRAMEWORK

- ***Agent Management System (AMS):*** An agent responsible for managing the operation of an Agent Platform (AP), such as the creation, deletion and oversight of the migration of agent to and from the Agent Platform (AP)

- ***Directory Facilitator (DF):*** An agent that provides "yellow page" services to other agents. It stores description of the agents and the services they offer.

- ***Agent Communication Channel (ACC):*** An agent that uses the information provided by the AMS to route messages between agents either within the same platform or agents on other platforms.

Source: FIPA[5]

The AMS and DF are automatically created when the JADE platform is first launched. The ACC allows message communication within and to/from different platforms (host computers). Both the AMS and DF utilize the ACC for communication.

Each instantiation of JADE is termed a *container*. While multiple instantiations of JADE, thus multiple containers, can exist on the same platform, there can be only a single *main container* on which the DF and AMS reside. As a result, within a JADE network, there can only be one DF and AMS. Agents residing on other platforms must rely on constant and reliable communication with the main container for a complete JADE runtime environment [8], as illustrated in Figure 9.

JADE uses various methods for message delivery between agents. If both the sender and the receiver agents reside in the same container, JADE uses event passing for communication. When the sender and the receiver reside in different containers but in the same platform, JADE uses Remote Method Invocation (RMI). For agents residing in different platforms, JADE uses Internal Message Transport Protocols (IMTP) such as IIOP, HTTP and WAP.

Figure 9 and Figure 10 illustrate the message delivery between agents in different scenarios.

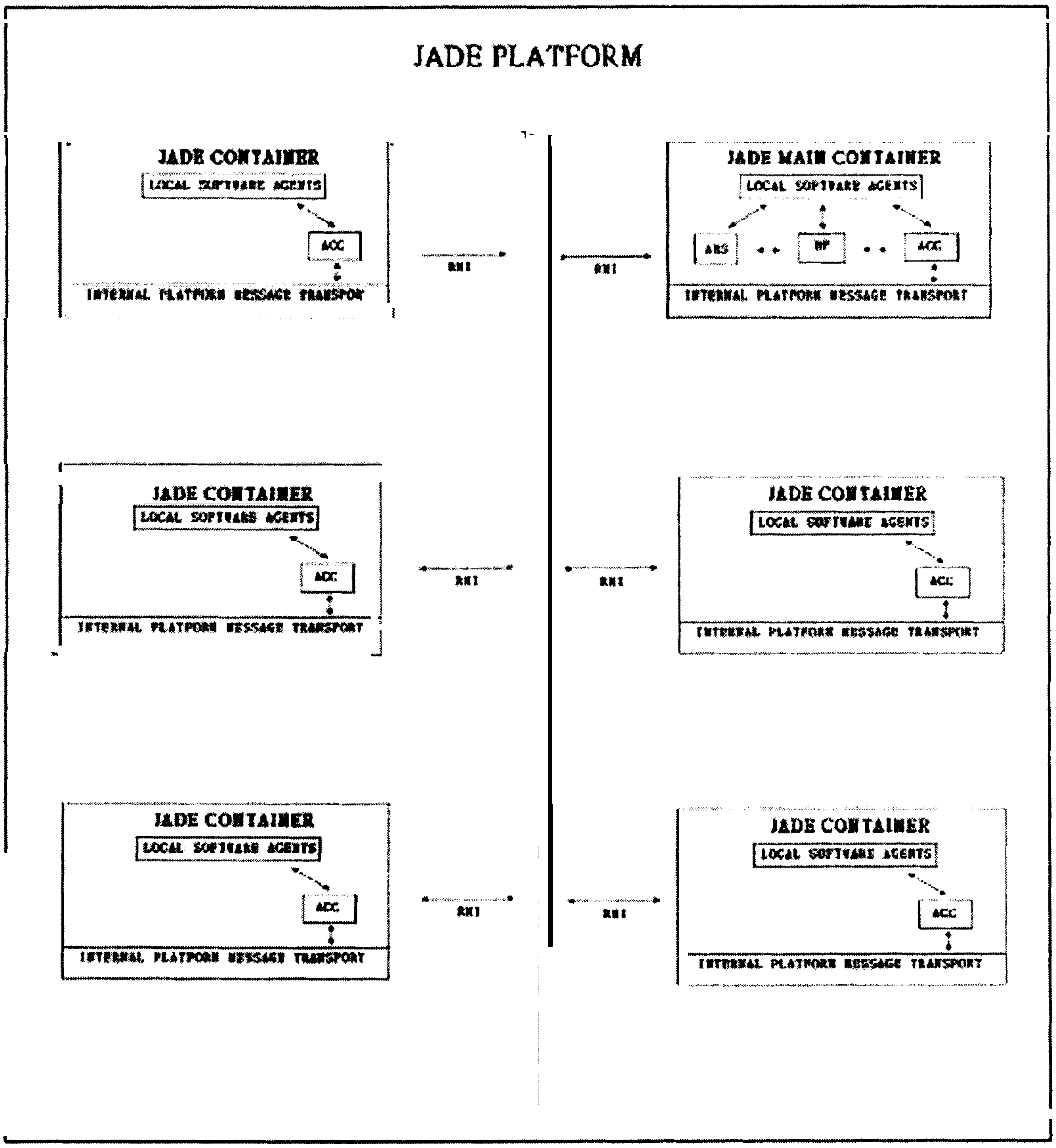


Figure 9. JADE Intra-Platform Message Delivery [12]

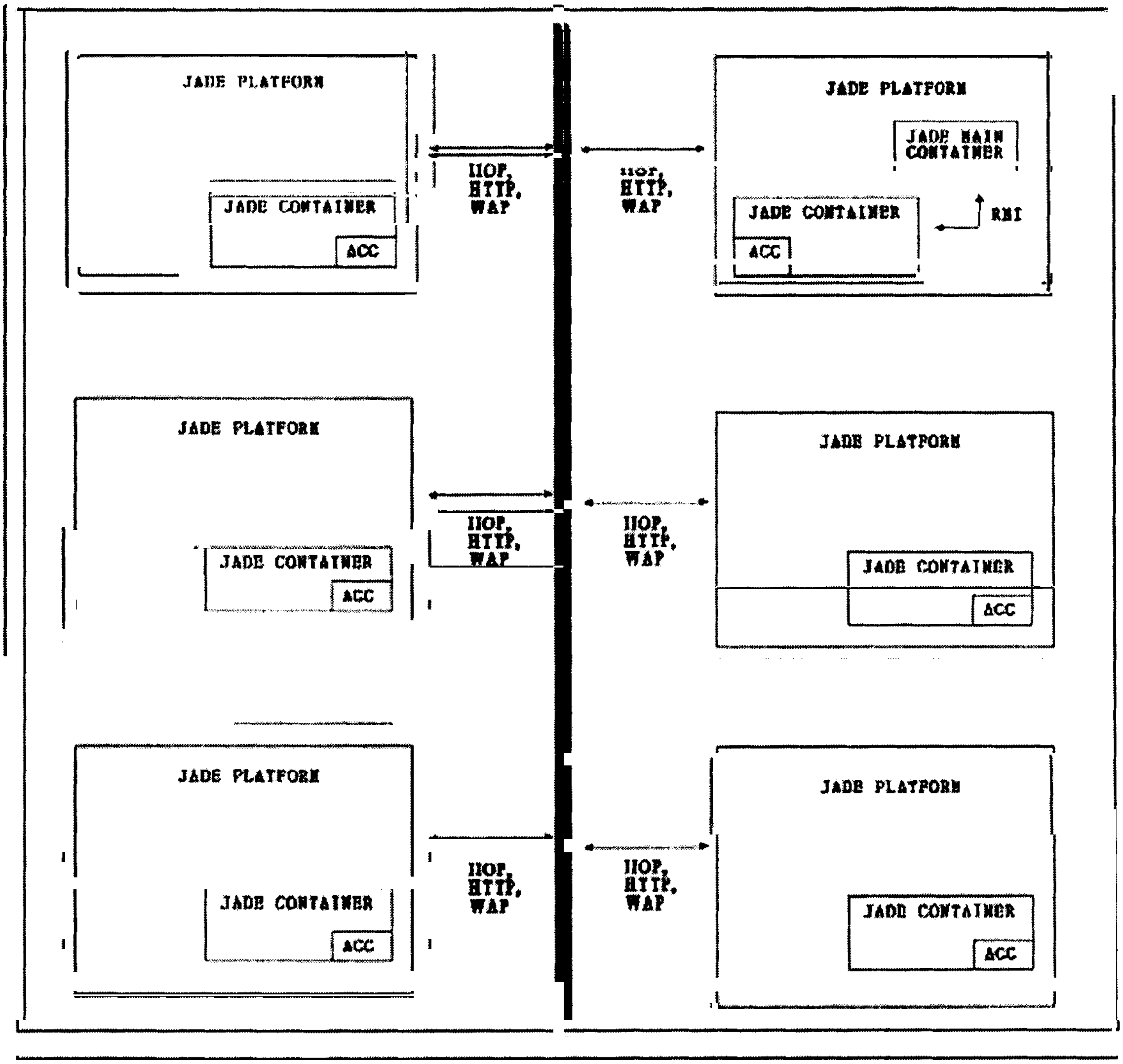


FIGURE 10. JADE INTER-PLATFORM MESSAGE DELIVERY [12]

2.1.2 JADE Software Architecture and Behaviours

Java was chosen by Telecom Italia Labs because of its many features geared towards object-oriented programming in distributed heterogeneous environment including Object Serialization, Reflection API and Remote Method Invocation (RMI) [17]. It provides application programmers with ready-made functionality and abstract interfaces for custom application dependent tasks [17].

JADE is composed of the following major software packages:

- ***Jade.core:*** Implements the kernel of the system. It includes the *Agent* class that must be extended by application programmer. *Behaviour* class hierarchy contained in the sub-package implements the logical tasks that can be composed in various ways to achieve complex tasks.
- ***Jade.lang.acl:*** Provides Agent Communication Language according to FIPA Standard Specifications.
- ***Jade.domain:*** Contains all Java class that represent Agent Management System defined by FIPA standards
- ***Jade.gui:*** Contains generic classes useful to create GUIs
- ***Jade.mtp:*** Contains the Message Transport Protocol that should be implemented to readily integrate with the JADE framework.
- ***Jade.proto:*** Provides classes to model standard FIPA interaction protocols *(fipa-request, fipa-query, fipa-contract-net)*

Figure 11 illustrates the interactions between the different Jade software packages and the AMS, DF and ACC.

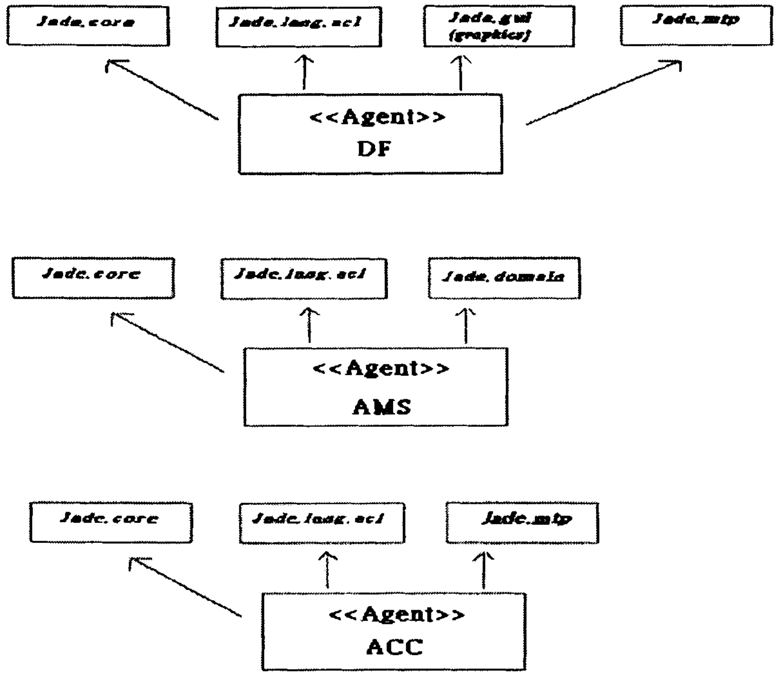


FIGURE 11. JADE AGENTS AND SOFTWARE PACKAGE INTERACTIONS

Figure 12 illustrates the dependencies between the different Jade software packages.

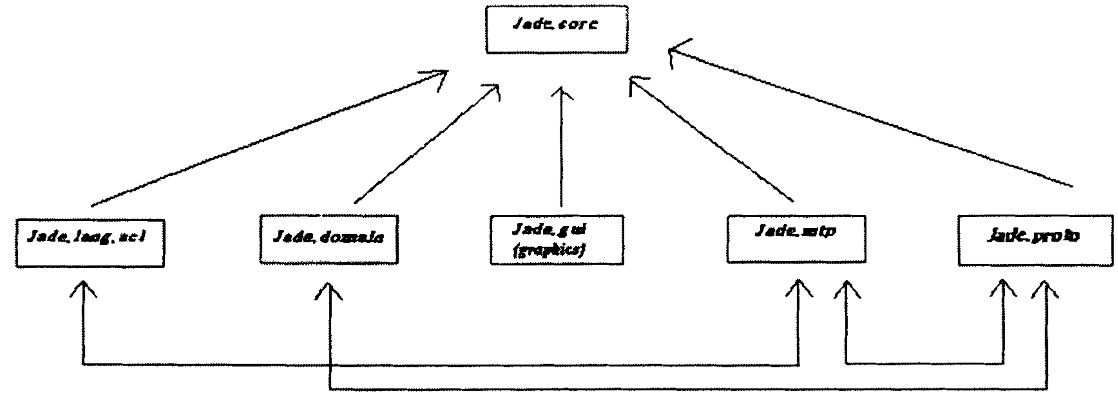


FIGURE 12. JADE SOFTWARE PACKAGE INTERACTIONS

Internally, each JADE agent is composed of a single execution thread and all its tasks are modelled and implemented as *Behaviour* objects, and implemented as a finite state machine. Adding a *Behaviour* object is equivalent to spawning a new (cooperative) execution thread within the agent [17]. Agent behaviours can therefore be described as a Finite State Machine.

There are two main types of *Behaviour*: *Simple* and *Composite*. A *Simple Behaviour* models a task that is not composed of subtasks while a *Composite Behaviour* models a task that is a combination of smaller, subtasks. Table 3 illustrates a few of the Behaviour models that are available.

TABLE 3. JADE BEHAVIOUR MODEL DESCRIPTION

Behaviour	Description
One Shot	• Tasks only performed once • Agent returns to idle state immediately after completion of task
Cyclic	• Task cycle repeats indefinitely • Agent never return to idle state
Complex	• Agent tasks model a Finite State Machine • Each state dependent on current condition and previous state • Agent returns to idle when given condition and state are met

Figure 13 illustrates and briefly describes the Jade class behaviour hierarchy.

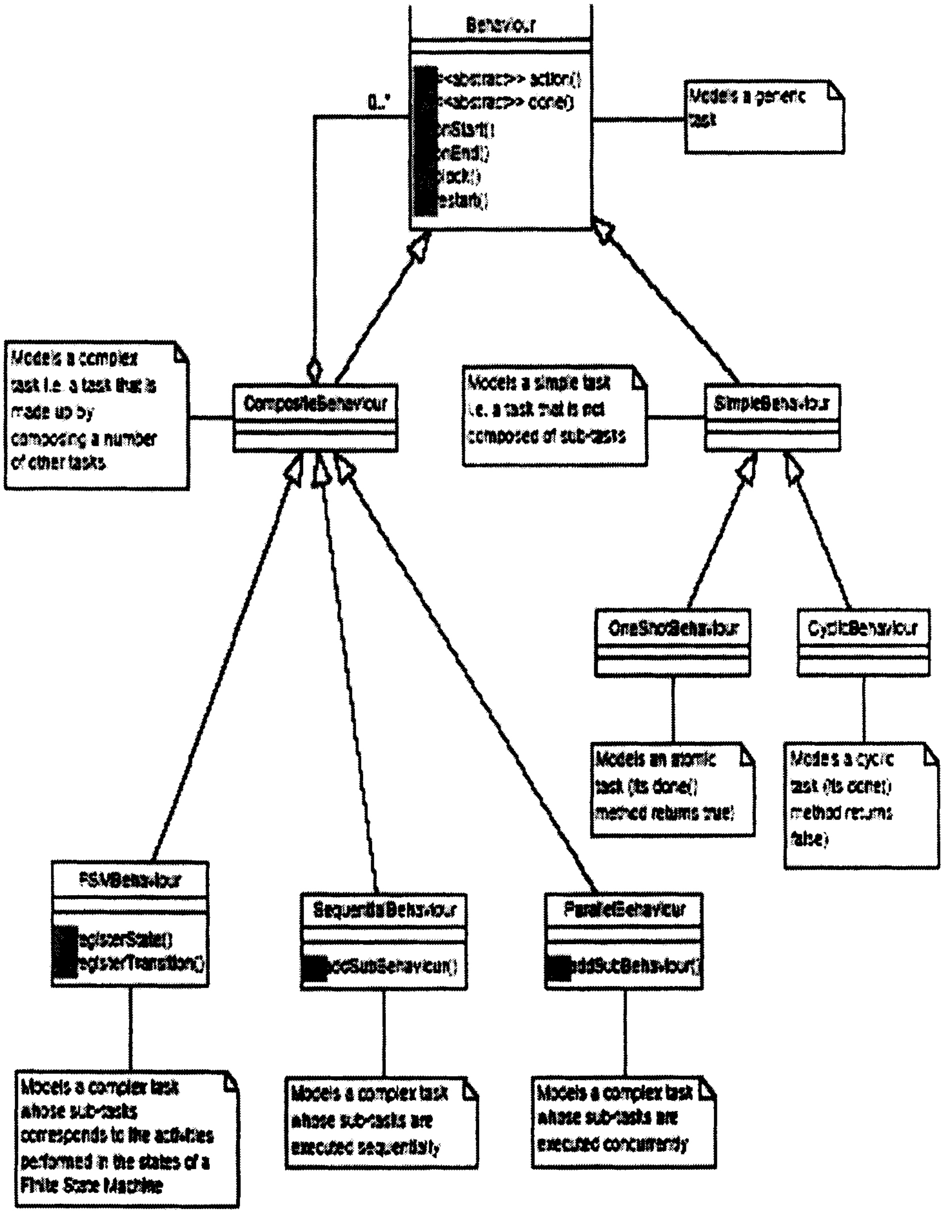


FIGURE 13. JADE BEHAVIOUR CLASS HIERARCHY [17]

2.1.3 Issues for JADE as a Distributed System

Some of the limitations of JADE that we will address in subsequent chapters are briefly described here.

Message transport between agents in JADE is handled internally and users have no knowledge and control of the exact path that the message is traversing.

Individual nodes in a Distributed System may not be able to directly communicate with each other. They rely on intermediary nodes to relay their information across the network. In a Wireless Distributed System application, wireless connectivity scenarios (e.g., dynamic link failure/establishment) cannot be simulated. Extensions are required to the current version of JADE to facilitate the simulation of a Distributed System.

A JADE application is dependent on the AMS and DF, which resides in the main container. Critical functions such as agent creation, migration, deletion and yellow page service cannot operate without the aid of AMS and DF. A complete JADE runtime system is critically dependent on the constant and reliable communication between the main and other containers. The failure of the main container will have a catastrophic effect on the entire JADE system.

Nevertheless, JADE also has advantages over conventional distributed computing techniques that facilitate the development of a distributed system. Table 4 lists some of the advantages and disadvantages that result from utilizing JADE in a distributed system.

TABLE 4. ADVANTAGES AND ADVANTAGES OF JADE IN A DISTRIBUTED SYSTEM

Advantages:	**Disadvantages**
• Open source, completely written in JAVA and FIPA-compliant • Serves as middleware to deal with communication transport and message encoding • Concise and efficient software architecture • All agent tasks modeled as *Behaviors* objects for simple implementation of complex tasks • Ability for agents to migrate from container to container, regardless of platform	• Cannot define specific path to receiving node • Critical dependence of AMS and DF of the *main container* for communication • Unable to simulate different transmission scenarios

2.2 JXTA

JXTA was developed by Sun Microsystems to enable end users to build distributed systems. It is a software framework that utilizes a set of protocols to support the development of distributed applications. JXTA does not define a specific type of application, but rather a standard for how the application should be created. Because the protocols are not rigidly defined, their functionalities can be extended to satisfy uniquely different applications [20]. The goal of JXTA is to achieve the following features:

- Operating System Independence
- Language Independence
- Provide services and infrastructures for distributed applications

Source: Li[15]

A JXTA application is able to incorporate a large number of potential participants in a JXTA-enabled distributed application. Because the architecture lacks a central management hierarchy, no failures of any client should result in a catastrophic failure of the entire application.

Participants in a JXTA network are known as peers. They are software entities that are similar to agents in JADE. Multiple peers can coexist on a single node, with each peer able to perform tasks individually. However, unlike agents in JADE, peers in JXTA are not FIPA-compliant and are not able to freely migrate. They are physically tied to the node on which they reside.

JXTA is composed of a set of protocols and a JXTA platform. The protocols allow an individual to easily produce a new JXTA application without extensive knowledge of the underlying distributed domain. The JXTA platform utilizes the protocols for the development of the distributed application and the different layers of abstractions behind each application such as peer communication and peer management

2.2.1 JXTA Protocols

The JXTA protocols are used to enable nodes to discover, interact, and manage a distributed application. The protocols abstract the implementation details, making the task of creating a distributed application much easier and less sustained. The protocol specification only describes how nodes communicate and interact; it does not restrict the implementation of a distributed application [20].

The protocols are built to smoothly handle communication between different operating systems, development languages and even exchanges between clients behind firewalls. The peer is assumed by JXTA Protocol to be any type of device, from *"the smallest embedded device to the largest supercomputer cluster"* [18].

The protocols have been specifically designed for *"ad hoc, pervasive, and multi-hop network computing"*. By using the JXTA protocols, peers in a JXTA application can cooperate to form *"self-organized and self-configured peer groups independently of their positions in the network (edges, firewalls), and without the need of a centralized management infrastructure." [20]*

JXTA protocols are based on XML – a widespread language-independent and platform-independent form of data representation.

Table 5 lists the JXTA protocols, their descriptions, and their functionalities within a JXTA application.

TABLE 5. JXTA PROTOCOLS AND DESCRIPTIONS

JXTA Protocol	**Functionalities within JXTA Application**	**Description**
Peer Discovery *(PDP)*	Resource Search	• Allows a peer to discover other peer advertisements (peer, group, service, or pipe). • The search mechanism used to locate information. Can also find peers, peer groups, and all other published advertisements.
Peer Resolver *(PRP)*	Generic Query Service	• Allows a peer to send a search query to another peer. • The resolver protocol is a basic communications protocol that follows a request/response format. • The resolver is used to support communications in the JXTA protocols like the discovery protocols. It is used by other protocols to send messages/requests

		to other peers
Peer Information *(PIP)*	Monitoring	• Allows a peer to learn about the status of another peer.
Rendezvous *(RVP)*	Message Propagation	• Responsible for propagating message within JXTA groups. • Defines a base protocol for peers to send and receive message within the group of peers and to control how messages are propagated.
Peer Membership *(PMP)*	Security	• Allows a peer to join or leave a peer group. • Supports the authentication and authorization of peers into peer groups. Provides security for peer group
Pipe Binding *(PBP)*	Addressable Messaging	• Used to create the physical pipe endpoint to a physical peer • Communication path between one or more peers • Connecting peers via the route(s) supplied by the Peer Endpoint Protocols.
Peer Endpoint *(PEP)*	Message Routing	• Uses gateways between peers to create a path that consists of one or more peers. • Utilizes the pipe binding protocol and its the list of peers to create the route between peers • Searches for gateways that allow the barriers, such as firewalls and others, to be traversed • Automatic protocol detection and conversion to allow two peers with different supporting protocols to communicate

Source: Developer[20]

Figure 14 illustrates the interaction between the various JXTA protocols. All protocols require the support of *PEP* to facilitate a path to the receiving peer. After a path has been determined, *PBP* is used to create the physical pipe communication between two peers. Finally, *PRP* is used to support generic query services that are basic to all peer communication. The sequence of interactions is illustrated in Figure 14.

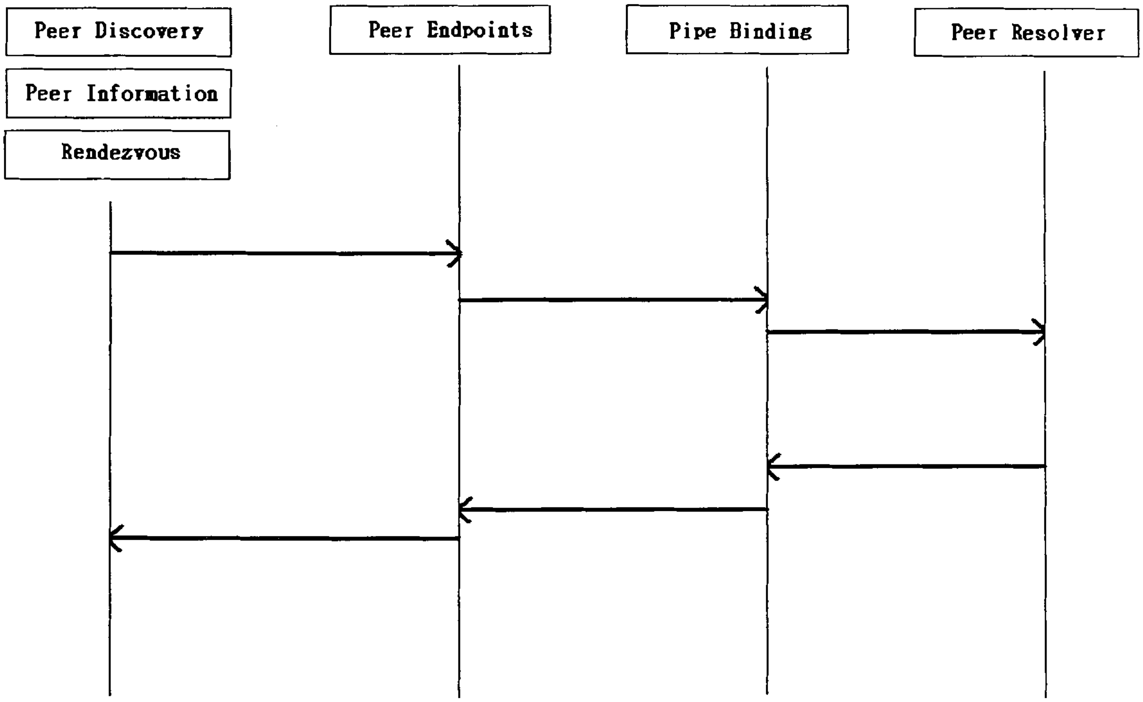


FIGURE 14. JXTA PROTOCOL SEQUENCE DIAGRAM

2.2.2 JXTA Platform

The JXTA Platform is modeled after the standard operating system, where there are three distinctive layers consisting of the Core, Services and Applications, as illustrated in Figure 15.

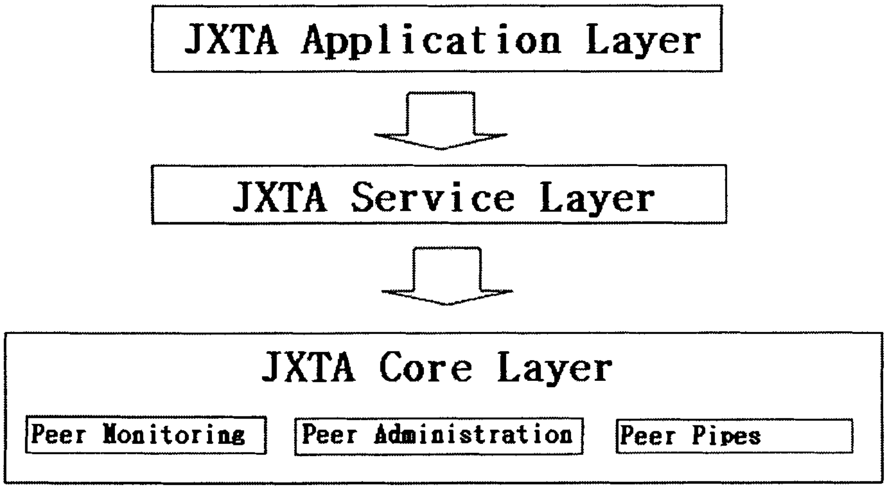


FIGURE 15. JXTA PLATFORM ARCHITECTURE [6]

The JXTA Core layer provides the foundation of any distributed application. Its components and functionalities are utilized by the Service layer. The Applications layer in turn uses the Services layer to access the JXTA network and utilities [18].

2.2.2.1 JXTA Core Layer

The JXTA Core layer provides the basis of all JXTA applications. New entities such as peers, peer groups, pipes and identifiers are created.

Table 6 lists the objects created in the Core layer and their involvement in the development of a distributed application.

TABLE 6. JXTA CORE LAYER CONCEPT DESCRIPTION

Entity Name	Description
Peer	• An entity on the network that implements one or more JXTA protocols • Rendezvous Peers support searches and store advertisements within the JXTA group
Pee/Node Group	• A collection of peers on the network with common interests or objectives. • A way to advertise specific services that are available only to group members. • Peers can join/resign from specific groups and be members in multiple groups • Membership authentication provides security for access to group with specific services or information.
End Point	• An address of a peer that implements a dedicated pipe of communication with another peer • Multiple end-points provide communication with multiple peers
Pipes	• A dedicated, virtual connection between two peers. • Used as abstraction to hide the fact multiple peers may be used to relay information to receiving peer. • Several types of pipes available: Uni-directional Asynchronous, Synchronous request/response, Bulk Transfer, Streaming, and Secure.

Advertisement	• An XML document that describes a JXTA message, peer, peer group, or service. • Advertisements stored in local Rendezvous Peers to support advertisement search within specific sub-section of a group
Identifiers	• Globally unique IDs that specify a resource, not the physical network address. Randomly generated to globally identify peers, peer groups, pipes or advertisements.

Source: Wilson[18]

2.2.2.2 JXTA Service Layer

The JXTA Service Layer provides network services that could be incorporated into different JXTA program. They include searching for resources on a peer, sharing documents among peers and performing peer authentication. Each JXTA application can only utilize a specific set of network services that are relevant to its application goals. The Service Layer can include additional functionalities that are being built by either open source developers working with JXTA or by the JXTA development team.

2.2.2.3 JXTA Application Layer

The Applications Layer builds on the resources of the service layer to provide end users with a complete JXTA solution. Various services are collectively used to provide such a solution. Instant messaging and file sharing are two of the most popular applications of distributed systems. A User Interface is typically present for a JXTA Application.

2.2.3 JXTA Communication

In the JXTA environment, different types of peers are used to coherently manage requests and communications. JXTA uses three types of peers to accomplish this task:

- Rendezvous peers are used to relay and search for requests,
- Router peers are used to implement the peer end-point protocol and establish a multi-hop path to the receiving node
- Gateway peer are used to relay messages between peers.

2.2.3.1 JXTA Rendezvous Peer

The key purpose of a Rendezvous peer is to facilitate the searching of advertisements beyond a peer's local network. Rendezvous peers usually have more resources than other peers and store a large amount of information about the peers around them, such as their identifications and services [20]. If the information requested cannot be found locally, the Rendezvous peer will act as a relay and forward the request to other rendezvous peers around the network.

Figure 16 illustrates a typical search involving multiple Rendezvous peers. The sequence of the search is as follows:

- Peer 1 initiates search by querying local Peer 2 and 3 via IP Multicast
- If specified resource not found, local Rendezvous peer is searched.
- If the rendezvous peer does not have the advertisement, successive rendezvous peers are searched. Besides peers local to the querying peer, only rendezvous peers are used.

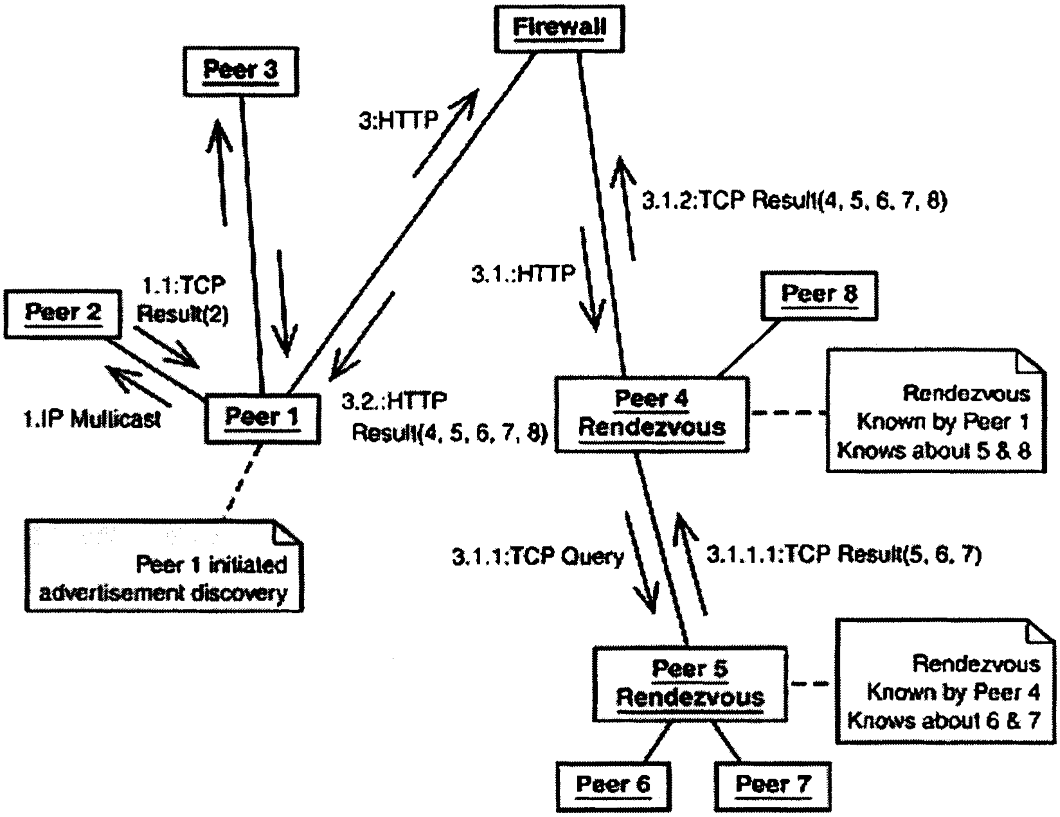


FIGURE 16. JXTA RENDEZVOUS PEER SEARCH [20]

Any peer has the option of being a Rendezvous, though not required. The Rendezvous peer can retain a cached copy of the results from previous searches. This feature expedites future searches with requests similar to previous searches.

2.2.3.2 JXTA Router Peer

A Router peer is any peer in JXTA that supports the *Peer Endpoint Protocol.* The protocol internally implements routing to determine the most efficient route to the destination peer.

The request for a route starts with a peer initiating the request to the Router peer. The Router peer first search the local network for the destination peer. If the peer is not found, other Router peers are contacted until the destination peer is located. Previous requests are also cached to expedite future requests

Figure 17 illustrates how a route is determined between two distant peers.

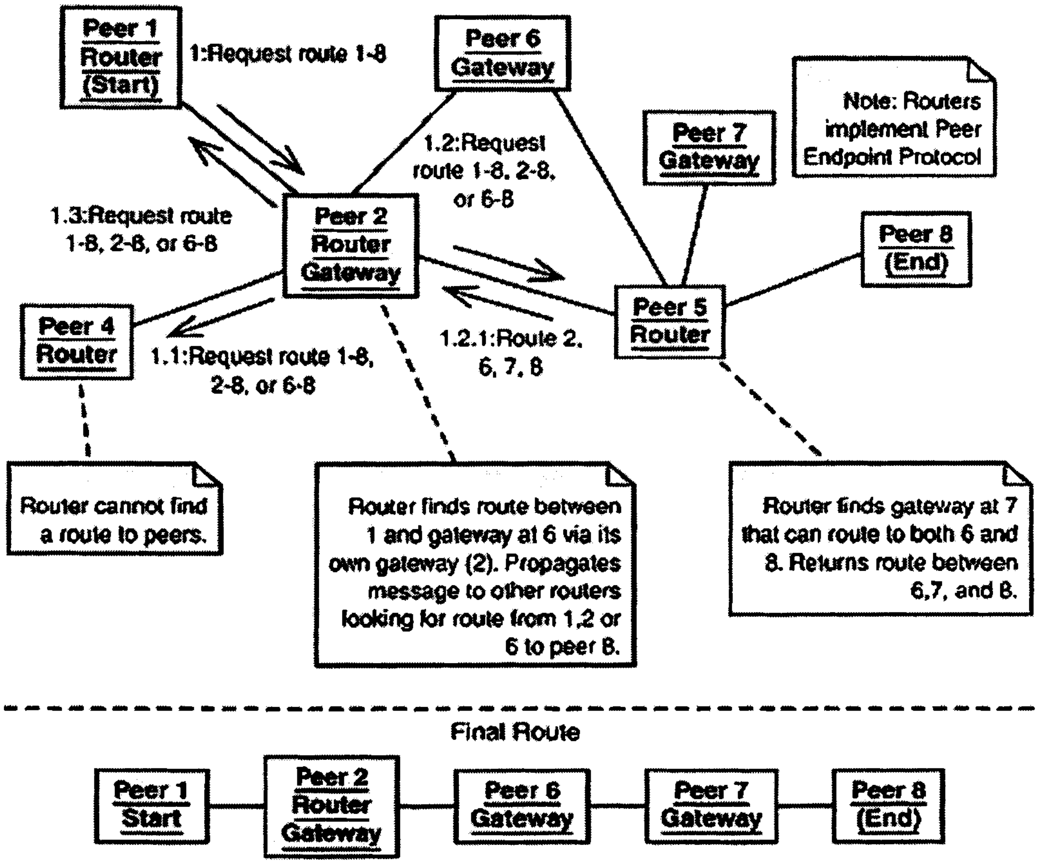


FIGURE 17: JXTA ROUTER PEER [20]

2.2.3.3 JXTA Gateway Peer

A Gateway peer is used to relay messages, not request, between peers. It can also store messages and wait for the receiving peer to collect the messages.

Gateway peers arise from the fact that different communication protocols are used by different peers. Some peers may use TCP, while other may use IP. To support wireless connectivity, the Wireless Application Protocol (WAP) is also needed [20]. Gateway peers act as intermediaries between the different protocols and provide translation service.

Gateway peers are also used to go through common security barriers such as firewalls, which filters nearly everything except HTTP. Figure 18 illustrates how a Gateway peer is used to interface between Peer 1 and Peer 3.

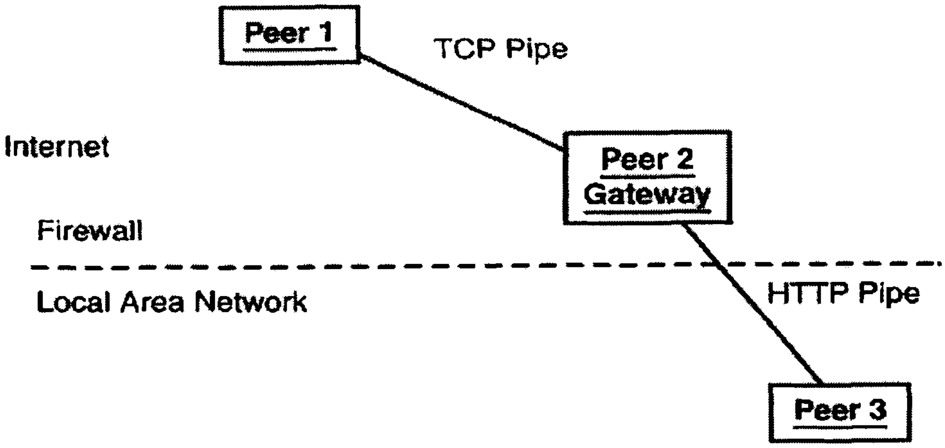


FIGURE 18: JXTA GATEWAY PEER [20]

When the messages are sent from Peer 3 to Peer 1, they are first sent via TCP to peer. The Gateway peer then holds the message until Peer 1 makes an HTTP request to retrieve the data [20].

2.2.4 Issues for JXTA as a Distributed System

Some of the limitations of JXTA that we will address in subsequent chapters are briefly described here.

Message transport between nodes in JXTA is handled internally and users have no knowledge and control of the exact path that the message is traversing. JXTA uses the *End-point Routing Protocol (ERP)* to systematically direct messages from the sender peer to the receiving peer.

Individual nodes in a Distributed System may rely on intermediary nodes to relay their information across the network In a Wireless Distributed System application, wireless connectivity scenarios (e.g., dynamic link failure/establishment) cannot be simulated with the current version of JXTA. Extensions of JXTA are required.

The XML message may reduce network efficiency. Its mandatory 256-bit peer ID and path specifications imply that an "*empty*" message that has no application-specific payload can easily reach 1 KB in size and thus affect the performance of the message exchange. Also, the complex messaging architecture of JXTA that involves the XML parser and several layers of abstraction will add significant overhead and affect the efficiency of the messaging framework [19].

Rendezvous, Relays and Gateway peers are used in JXTA to cache routes and pass messages/requests between peers. As the size of the network grows, the amount of processing required by these nodes will grow exponentially, resulting in a degradation of network efficiency.

Nevertheless, JXTA has advantages over conventional distributed computing techniques that facilitate the development of a distributed system. Its protocols and the abstraction of the underlying distributed domain allow developers to more easily develop distributed systems. Also, caching of network information allows messages and requests to be transported more efficiently. Table 7 lists some of the advantages and disadvantages of utilizing JXTA in a distributed system.

TABLE 7. ADVANTAGES AND DISADVANTAGES OF JXTA IN A DISTRIBUTED SYSTEM

Advantages:	**Disadvantages**
• No extensive knowledge of underlying distributed domain • Support large number of potential peers with no central management system • Network resources distributed among multiple machines • Automatic protocol translation for communication between peers with different protocols • Cached network information reduces search time requests	• Developers unaware of mechanisms and path used for message transport. • Sizeable XML messages, XML parser and several layers of abstraction may lead to network inefficiency. • Dependence on specific types peers for routing, messaging and requests between peers. • Increased memory overhead by caching network configuration for every peer

2.3 Differences between JADE and JXTA in Distributed Systems

Both JADE and JXTA are designed with the goal of achieving a distributed system. However, both platforms have issues that must be resolved before a distributed system can be established.

In JADE, agents residing on remote containers are dependent on the AMS and the DF that reside in the main containers. Although remote containers are contained on different platforms than the *main container,* the remote container is critically dependent on the agents of the *main containers* and their services. The failure of the *main container* would also indicate the failure of the entire JADE network. JXTA, on the other hand, does not employ remote containers. A JXTA peer cannot be subdivided and it resides on a single host. Every host represents a JXTA peer and they communicate either directly or through relay nodes with other peers. Failure of one peer will not have a catastrophic effect on the overall system.

In JADE, agents are able to freely migrate from container to container, regardless of the physical location of the platform on which the container resides. However, in JXTA, a peer is represented by a physical host such as a hand-held device or a desktop computer. Peers cannot migrate freely across the network. They are embedded within the hosts.

Another major difference between them is their respective message protocols. The messaging architecture of JXTA when compared to JADE is complex. The use of XML parsers and several layers of abstractions add significant overhead to the efficiency of the network. The increased use of relay peers in JXTA can also lead to congestion and degrade overall network performance.

Table 5 below illustrate some key differences between JADE and JXTA when utilized in a distributed system.

TABLE 8. COMPARISON OF JADE AND JXTA IN DISTRIBUTED SYSTEM

	JADE	JXTA
Messaging Architecture	Relatively simple. Uses IMTP for Inter-platform and RMI for Intra-platform communication	Uses XML parser and several layers of abstraction. Pipes used for communication. Significant overhead
Node/Peer Migration	Agents able to freely move to different containers	Peers are embedded within the host they reside in
Distributiveness	Limited by the *main container*. Remote containers dependent on main container.	Unrestricted scalability. Each peer is uniquely identified and independent.
Platform Complexity	Very manageable and coherent	More sophisticated and steep learning curve.
FIPA Compliance	Yes	No

3 JADE/ JXTA EXTENSIONS FOR IMPROVED DISTRIBUTED SYSTEMS

Both JXTA and JADE have limitations for implementing a distributed system. Both JADE and JXTA lack the ability to simulate wireless connectivity conditions such as dynamic link establishment/failures and data quality over multiple hops. Although the use of *Endpoint Routing Protocol* in JXTA ensures messages are efficiently routed to their destination, it does not specify the absolute path they must traverse. In JADE, communication transport is also handled internally and no user-defined routing mechanisms are available. Ideally, a true WDS should combine wireless protocols with the functionality of a peer-to-peer collaborative system environment. This would enable multi-hop capabilities to find distant nodes on the network without the need for a centralized management system.

3.1 Virtual Wireless Environment

In current wireline networks, nodes are physically connected and information is systematically routed from sender to recipient. However, in a WDS, each node is not fully aware of the extent of the entire network and with whom it can communicate directly. For example, suppose that we wish to model a wireless network consisting of 5 nodes using a wireline LAN. Individual nodes can only communicate with a set of receiver nodes as predetermined by the wireless conditions. This set of receivers need not be constant; they can be dynamically changed to model the wireless nature of a WDS, such as user roaming.

In the wireless scenario illustrated in Figure 19, we suppose that Node_A is a roaming node. At $t = t_0$, Node_A has only Node_B as its receiver.

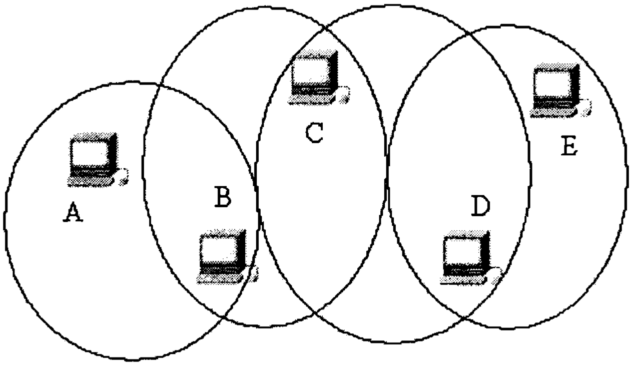


FIGURE 19. ROAMING NODE WITH INTELLIGENT LINK AT T=T0

However, at $t=t_1$, the sender (Node_A) will be at a different location, as shown in Figure 20, and has different receivers (Node_D and Node_E).

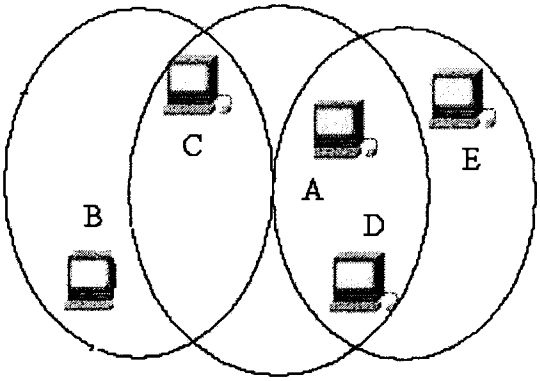


FIGURE 20. ROAMING NODE WITH INTELLIGENT LINK AT T=T1

This situation models a roaming node where its linkages to other nodes are dynamically changing.

We could also model other scenarios such as dynamic link congestion/failure by setting the links between nodes to be deleted or created as a function of time. Such a scenario can also be used to model the uncertainty of wireless transmission.

Timing and administrative overhead issues can also be modeled. We can calculate the time required by messages to travel from one end of the network to another and the effects of multiple messages. Stress test can be carried out to ensure that the system can adequately perform under heavy traffic. We can also measure the effectiveness of different routing algorithms and also peer-to-peer environments.

Currently, this type of distributed system is still mainly a research topic. Extensions are required to current distributed systems to simulate a true distributed system.

3.2 JADE Architecture Extension

Fully distributed systems must not be dependent on any particular node. The key to improved distributiveness in JADE is the elimination of the central influence of the main container. Each host will be completely independent of other hosts and a failure of one host will not a have catastrophic effect on the network.

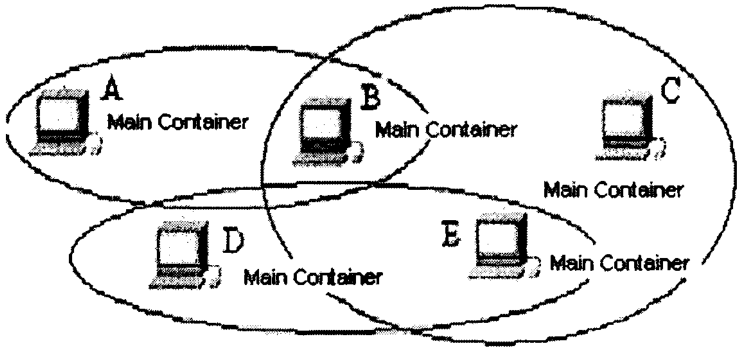


FIGURE 21. JADE IN A VIRTUAL WIRELESS ENVIRONMENT

As illustrated in Figure 21, each host will become a *main container* and the use of *remote containers* will be eliminated.

For example, in a wireless environment, nodes can only communicate directly with neighbour nodes and thus are not aware of all available nodes on the network. Also, specific message paths that transverse several intermediary nodes may be required to relay messages. Finally, the added administrative overhead must be properly handled to ensure a coherently managed Wireless Distributed System.

We can accomplish these tasks by extending the components in the established JADE Agent Platform to include the *Global Directory Facilitator (GDF)*, *Wireless Agent Communication Channel (WACC)*, and the *Global Agent Management System (GAMS).*

3.2.1 Wireless Agent Communication Channel (WACC)

In a wireless environment, nodes can only communicate directly with neighbour nodes. Messages can only be sent directly to a list of available receivers as predetermined by a user-defined scenario. This limitation is used to model the wireless nature of the WDS.

This feature is accomplished by extending the *Agent Communication Channel (ACC)* of the JADE Agent Platform, as illustrated in Figure 22. The *WACC* is in constant communication with the *GDF* for the current list of available nodes.

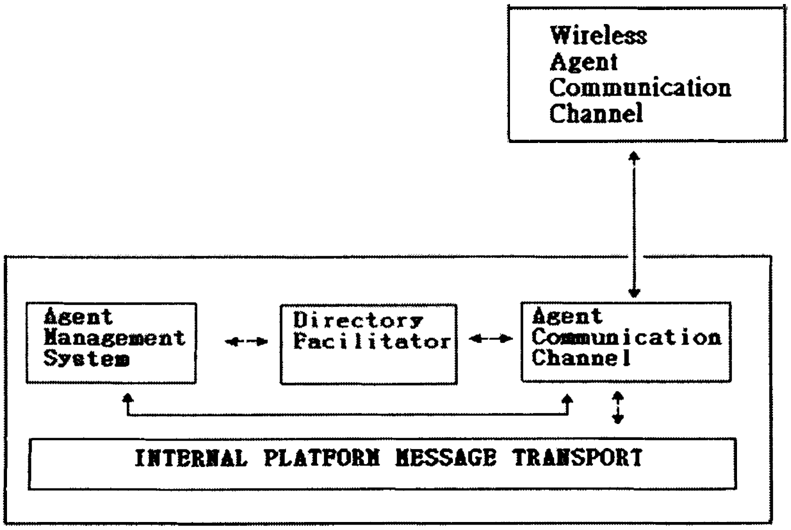


FIGURE 22. WIRELESS AGENT COMMUNICATION CHANNEL IN AN AGENT PLATFORM

3.2.2 Global Directory Facilitator (GDF)

Unlike wireline networks for which all nodes are aware of the existence of all other nodes, a wireless system is only aware of nodes within its signal range. When a new node becomes available, that information must be made available to the network by broadcasting its presence to neighbour nodes, which they broadcast to their neighbours.

This multi-hop functionality feature is incorporated into JADE by extending the *DF* to include the *GDF*, as shown in Figure 23. The *GDF* is responsible for maintaining a current list of all agents and their services. This extension enables a node to be aware of both neighbour and distant nodes.

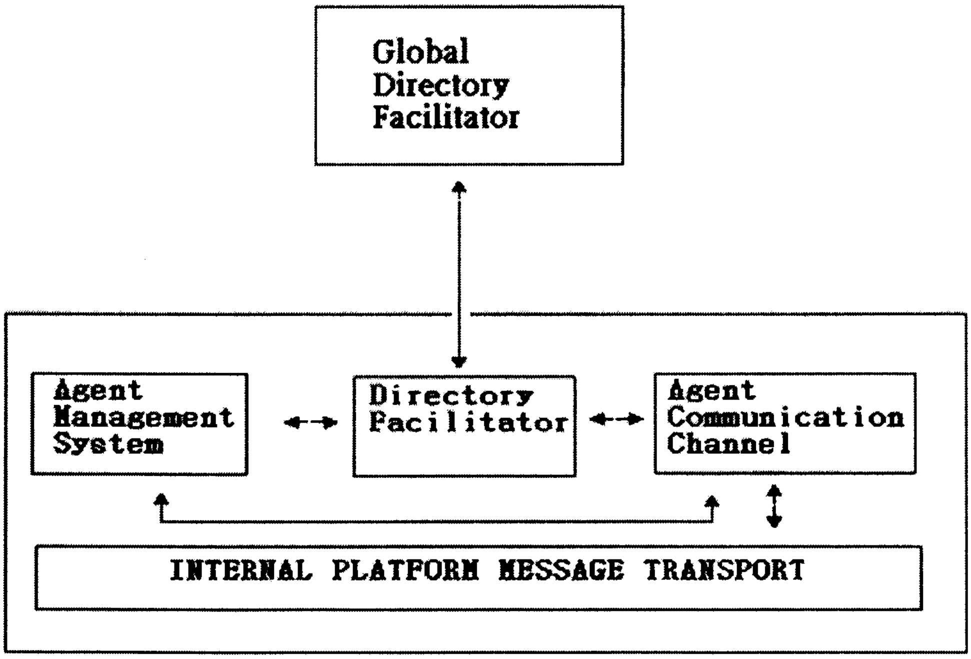


FIGURE 23. GLOBAL DIRECTORY FACILITATOR IN AN AGENT PLATFORM

3.2.3 Global Agent Management System (GAMS)

As illustrated in Figure 24, the *GAMS* extends the functionalities of the *AMS* to manage the additional administrative overhead at the network level. It is also responsible for providing agent management service for its respective node in the Wireless Distributed System. Its tasks also include agent creation, migration, and retirement.

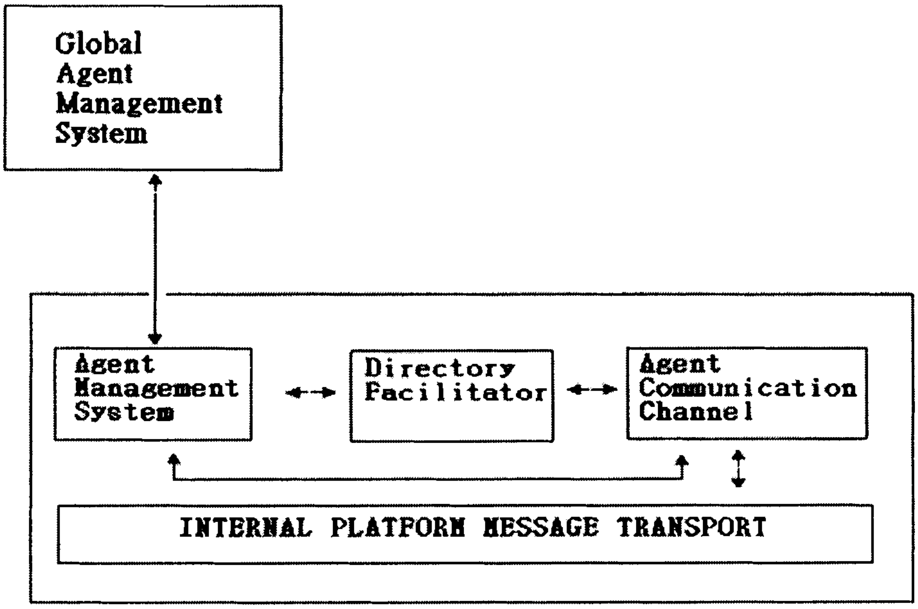


FIGURE 24. GLOBAL AGENT MANAGEMENT SYSTEM IN AN AGENT PLATFORM

The *GAMS* is in constant communication with the *WACC* and *GDF* to provide a complete WDS environment from a wireline LAN.

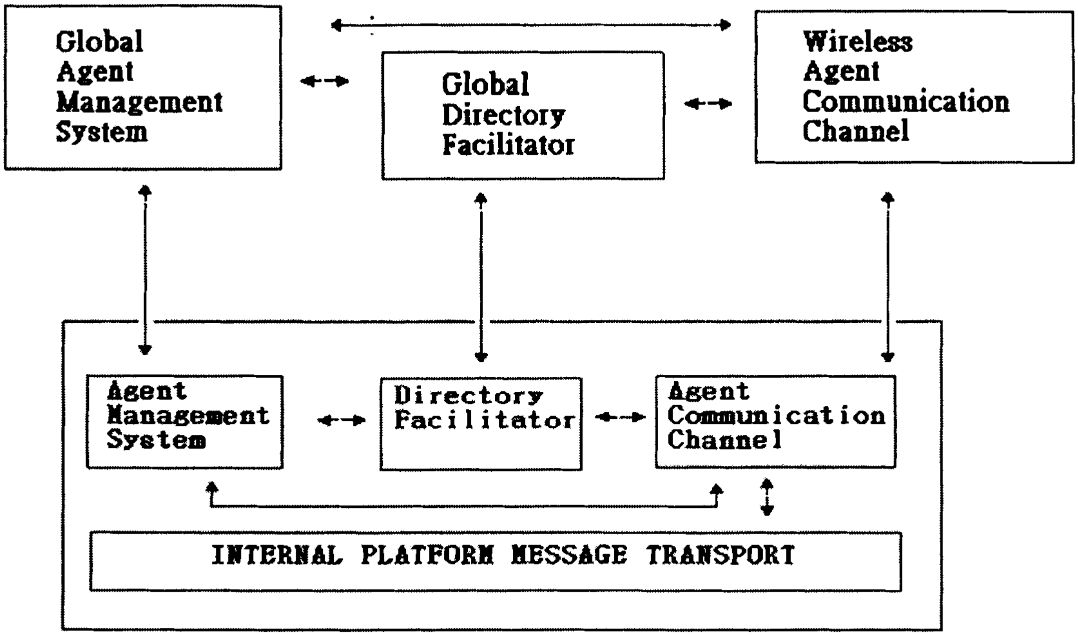


FIGURE 25. MODIFIED JADE FRAMEWORK

3.3 JADE Software Architecture Overview

Based on Figure 25, extensions are required of the JADE Agent Platform to implement an improved Distributed System. In this thesis, the extensions are based on the use of three distinct JADE agents -- *Broadcast, Sender, Receiver* -- that would operate even for a wireless application.

- The *Broadcast Agent* handles broadcasted messages to/from other nodes and is responsible for maintaining a current list of all nodes currently available on the network.
- The *Sender Agent* provides management service for the respective node, and is responsible for sending messages.

- The *Receiver Agent* receives messages from other nodes and internally determines the subsequent nodes that the message should traverse.

3.3.1 Broadcast Agent

To incorporate multi-hop functionality into JADE, each node must know precisely which other nodes are currently available. This task is accomplished by the *Broadcast Agent.* It is responsible for maintaining a current list of all nodes on the network.

When a node is initiated, the *Broadcast Agent* will first broadcast its existence to the JADE network, after which it will loop indefinitely for a reply message. When a message arrives, the *Broadcast Agent* writes the agent information contained in the message to the *GDF*. Just before the node retires, an exit message is again broadcast to the network to indicate its termination.

3.3.2 Receiver Agent

Similar to the *Broadcast Agent*, the *Receiver Agent* also waits indefinitely for a message to arrive. Its main task is to process incoming messages and acts as an intermediary node if necessary. Routing algorithms determines the path of the next node and messages are routed accordingly. Table 8 lists the types of incoming messages that the *Receiver Agent* currently supports.

TABLE 9. MESSAGE TYPES SUPPORTED BY THE RECEIVER AGENT

Message Type	**Description**
Administrative	• Used to establish virtual connection with neighbour nodes.
Broadcast	• Used to establish global directory of all nodes available on the network
Specific-Path	• Used to route messages according to user-specified path
Update-Hop Message	• Used to update global hop-list
Update-Hop-List-Header	• Used to update Global Directory Facilitator

3.3.3 Sender Agent

The *Sender Agent* is responsible for providing agent management service for its respective node in the Wireless Distributed System. Its tasks also include agent creation, migration, and retirement. It is also in charge of administrative overhead at the network level.

The *Sender Agent* contains the entry point for the end user to operate a JADE node. A simplified GUI displays all available nodes currently on the network to communicating with a specific node through a user-defined routing method. Messages can be sent either directly to the destination node, or routed through a number of predefined methods.

3.4 JXTA Architecture Extension

Unlike JADE, where containers residing on remote machines are dependent on the *main container* on the host machine, each JXTA node is an independent entity that is not reliant on any other network resources. Multiple peers can coexist on a single JXTA node.

The Rendezvous peer allows network resources to be discovered in a robust and efficient manner. The Router peer plots a suitable path for the message to traverse, and the Gateway peer systematically routes the message according to that path. The three peers work in conjunction to coherently manage any JXTA application with unrestricted scalability.

However, the extensive use of the three nodes limits its ability to fully simulate a fully distributed system. The path taken by the Router node is accomplished automatically by utilizing the *End-Point Routing Protocol.* The system developer is unaware of the specific path and messages are routed automatically by the Gateway node.

To simulate a fully Distributed System, the system developer must be able to specify the exact path that the message must traverse, and also the conditions of the links between peers. Then, the system developer will be able to simulate wireless scenarios such as dynamic link establish and user roaming. Different routing algorithms can then also be implemented to test their efficiency and robustness under congestion. Also, the added administrative overhead must be properly handled to ensure a coherently managed Wireless Distributed Environment.

In this thesis, these tasks are accomplished by extending the components in the established JXTA Core layer to include the *Wireless Peer Pipes (WPP)*, *Global Peer Messaging (GPM),* and the *Global Peer Monitoring (GPM).*

The JXTA Core layer and its components are shown in Figure 26 for reference.

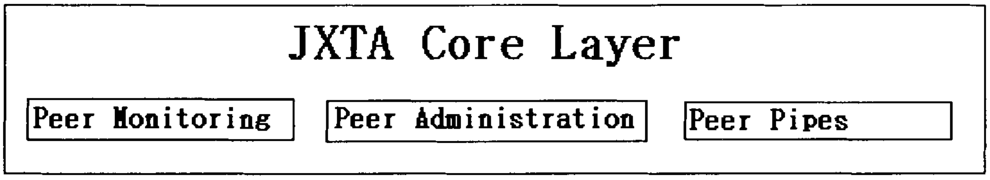


FIGURE 26. JXTA CORE LAYER AND COMPONENTS

3.4.1 Wireless Peer Pipes (WPP)

Similar to the Agent Communication Channel (*ACC*) in the JADE architecture, the *Peer Pipe* is responsible for communication between peers. It must be extended to restrict sending messages to neighbour peers. This extension is termed *Wireless Peer Pipes*, as illustrated in Figure 27. The *WPP* is in constant communication with the *GPM* for the current list of available peers and restricts sending messages to a list of predetermined neighbour peers.

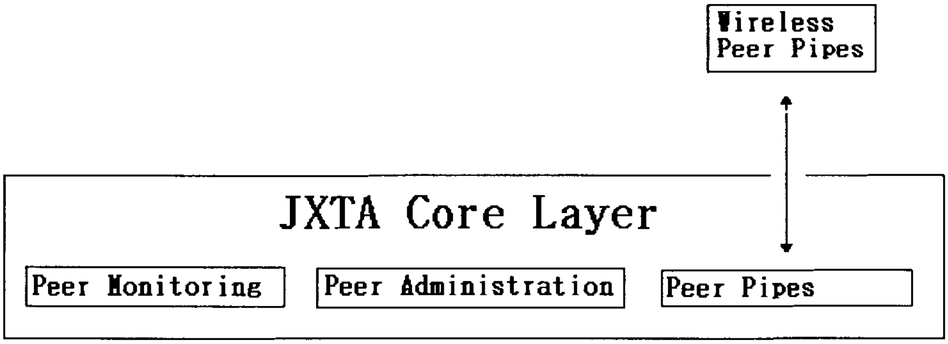


FIGURE 27. JXTA EXTENSION: WIRELESS PEER PIPE

3.4.2 Global Peer Monitoring (GPM)

Unlike wireline networks in which all nodes are aware of the existence of all other nodes, a wireless system is only aware of nodes within its signal range. When a new node becomes available, that information must be made available to the network by broadcasting its presence to neighbour nodes..

This multi-hop functionality feature is incorporated into JXTA by extending the *Peer Monitoring* to include the *Global Peer Monitoring (GPM)*, as illustrated in Figure 28. The *GPM* is responsible for maintaining a current list of all peers currently available in the JXTA network. This extension, illustrated in Figure 28, enables each peer to be aware of both neighbour and distant peers.

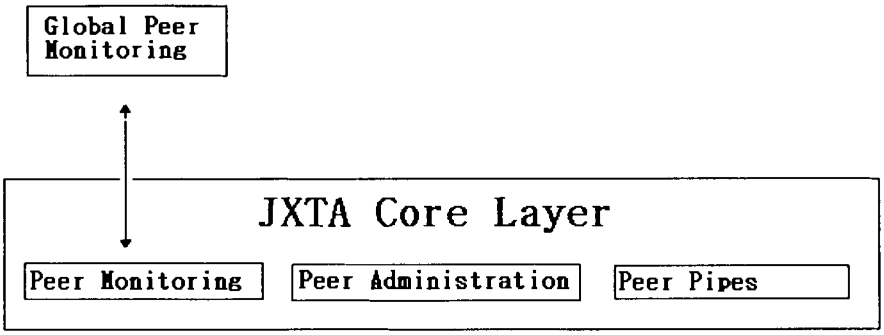


FIGURE 28. JXTA EXTENSION: GLOBAL PEER MONITORING

3.4.3 Global Peer Administration (GPA)

The *GPA,* as illustrated in Figure 29, extends the functionalities of the *Peer Administration* to manage the additional administrative overhead at the network level. It is also responsible for providing peer management service for the respective peer.

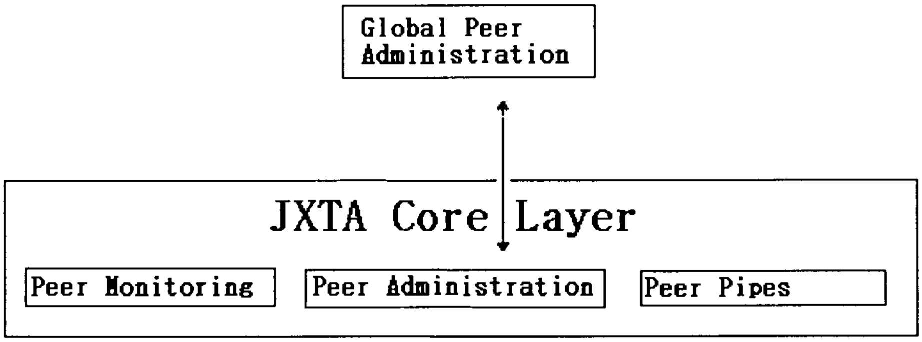


FIGURE 29. JXTA EXTENSION: GLOBAL PEER ADMINISTRATION

The *GPA* is in constant communication with the *WPP* and *GPM* to provide a complete distributed environment from a wireline LAN in JXTA, as illustrated in Figure 30.

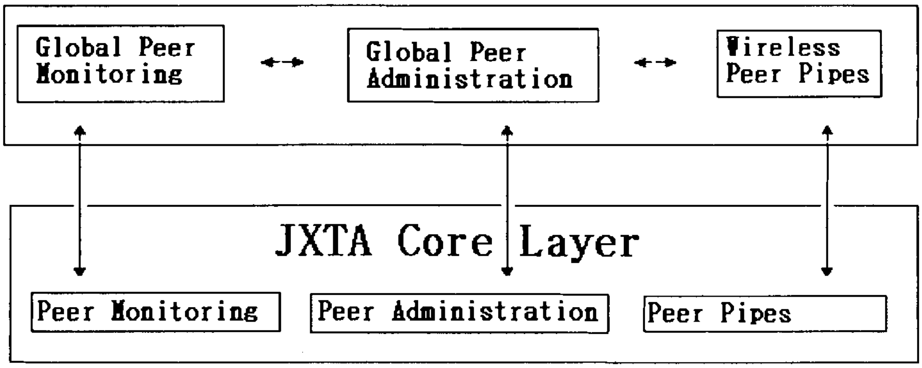


FIGURE 30. MODIFIED JXTA FRAMEWORK FOR AN IMPROVED DS

3.5 JXTA Software Architecture Overview

As shown in Figure 30, extensions are required from the JXTA Core Layer for an improved Distributed System. In this thesis, the extensions are accomplished by implementing four distinct Java Classes: *PipeListener(), PipeSender(), PipeComm(), PeerRoute()* that would operate even for wireless environments.

- *PeerRoute()* models the *GPM.* It handles broadcasted messages to/from other nodes and is responsible for maintaining a current list of all nodes currently available on the network.
- *PipeListener()* and *PipeSender()* are used to model the *WPP.* Together they send and receive messages according to a user-defined scenarios.
- *PipeComm()* models the *GPA.* It is used to handle the added administrative overhead and is used to initialize and supervise JXTA nodes. It also contains the entry point for developers to operate JXTA nodes.

3.5.1 PipeComm() Class

The *PipeComm() Class* contains the entry point for the end user to operate a JXTA node. It is also in charge of administrative overhead at the network level.

A simplified GUI gives the users the functionalities ranging from displaying all available nodes currently on the network to communicating with a specific node through a user-defined routing method. Messages can be sent either directly to the destination node, or routed through a number of predefined methods, such as direct, specific path, or maximum hops allowed.

3.5.2 PeerRoute() Class

The *PeerRoute() Class* is responsible for maintaining a current list of all nodes available on the JXTA network. When the JXTA node is first initialized, it advertises its existence to the network. This task is accomplished by:

- Create an input pipe
- Bind itself to that input pipe
- Publish the pipe advertisement so that other peers can obtain the advertisement

Pipes are used extensively in JXTA as the core mechanism for message exchange between JXTA peers. They provide a simple, unidirectional and asynchronous channel of communication [20].

Using the JXTA *Binding Protocol*, a sender node will dynamically search for the pipe advertisement belonging to this receiving node. When the advertisement is found, an output pipe is created by the sender and the message is sent through the pipe.

Once initialized, the *PeerRoute() Class* is used to handle broadcast messages from other nodes to maintain a current list of nodes.

3.5.3 PipeSender() Class and PipeListener() Class

The two classes work in conjunction to model the *WPP* and restrict the sending of messages according to a user-defined scenario.

The *PipeSender* class creates a dedicated output pipe to the specified receiving peer and sends messages on it. The class first asynchronously creates an output pipe with a specified receiving peer. Once the end-points have been resolved (input pipe advertisement found and output pipe successfully created), a message is created and sent through the pipe.

The *PipeListener* class creates input pipes used to receive messages. A dedicated input pipe is first created, and the receiving peer then binds itself to the input pipe. Finally, the input pipe is advertised on the JXTA network so other peers are able to dynamically discover the receiving peer.

Whenever a message arrives, the *PipeListener* class will be called asynchronously to retrieve and parse the message. Table 10 lists the types of incoming messages that the *PipeListener() Class* currently supports.

TABLE 10. MESSAGE TYPES SUPPORTED PIPELISTENER CLASS

Message Type	**Description**
Administrative	• Used to establish virtual connection with neighbour nodes.
Broadcast	• Used to establish global directory of all nodes available on the network
Specific-Path	• Used to route messages according to user-specified path
Update-Hop Message	• Used to update global hop-list
Update-Hop-List-Header	• Used to update Global Directory Facilitator

The *PipeListener Class* is also responsible for forwarding the messages onto the next peer. The *GPM* is consulted to retrieve the list of available node and messages are routed accordingly.

4 JADE/JXTA SOFTWARE EXTENSION IMPLEMENTATION

Both JXTA and JADE are software platforms designed to facilitate the implementation of a distributed system. However, they have limitations discussed in Chapters 2. With a distributed system having the potential of becoming an efficient, robust, and scalable system, the extensions discussed in Chapter 3 must be implemented. This chapter discusses the software implementation details of the extensions put forth in Chapter 3.

4.1 JADE Implementation

The standard FIPA agent model utilized by JADE is shown again in Figure 31. The model must be extended to fully simulate an improved distributed system, one that even operates in a wireless environment.

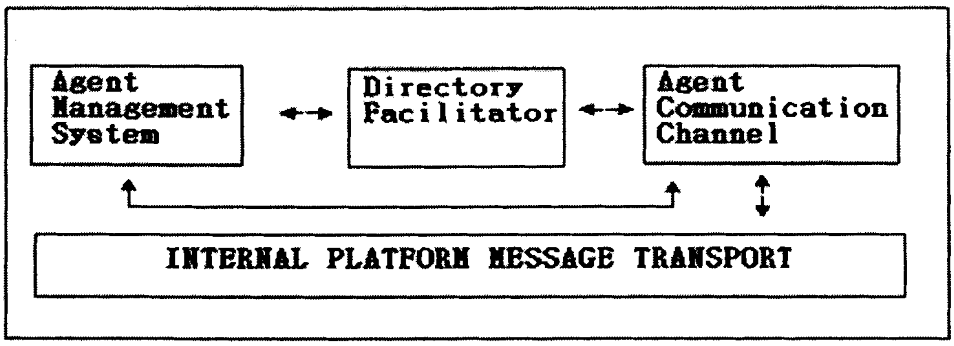


FIGURE 31. FIPA COMMUNICATION FRAMEWORK [5]

The extensions are achieved by establishing three new subcomponents: *Wireless Agent Communication Channel (WACC), Global Directory Facilitator (GDF)*, and *Global Agent Management System (GAMS).* These three subcomponents and their interactions are shown in Figure 32.

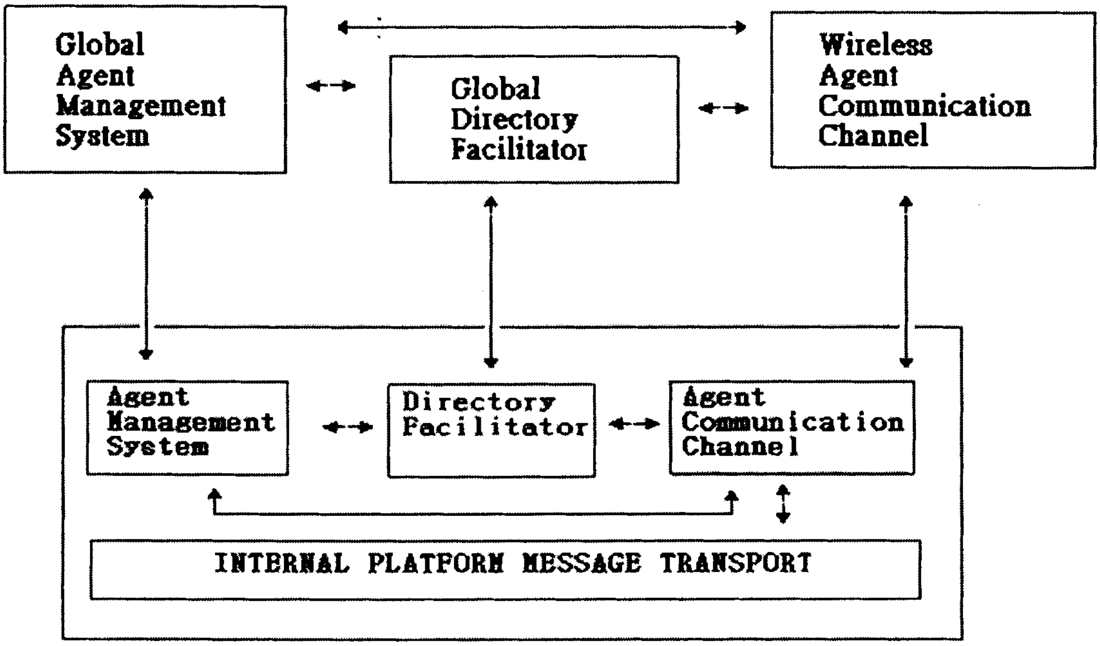


FIGURE 32. EXTENSIONS OF A JADE AGENT MODEL

In this thesis, the extensions are accomplished by utilizing three distinct JADE agents; *Broadcast, Sender, Receiver agents* that would work even for a wireless environment.

4.1.1 Broadcast Agent Implementation

The *Broadcast Agent* is responsible for dynamically maintaining a current list of all nodes available on the network. After broadcasting its existence to the network, it waits indefinitely for a broadcast message to arrive. The operations of the Broadcast Agent are summarized as follows:

```
while (true)
{
    // Set Java Multicast address and port for message reception
    Multicast_setup();

    // Wait indefinitely for broadcast message
    Multicast_receive();

    // Process incoming message and write to GDF
    Store_GDF();

    //Reply to Sender
    reply();
}
```

The interactions between the *Broadcast Agent* and JADE software packages are illustrated in Figure 33.

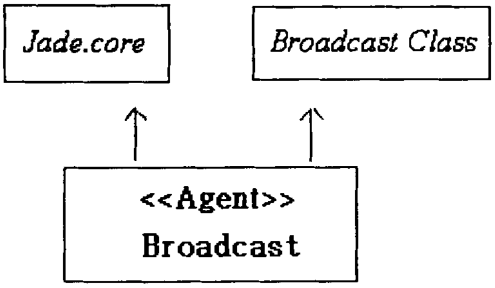


FIGURE 33. BROADCAST AGENT INTERACTION WITH JADE SOFTWARE PACKAGES

The *Broadcast Class* that makes up the *Broadcast Agent* implements the different methods required to receive and process a broadcast message.

4.1.1.1 Multicast_setup Method

The *multicast_setup* method initializes the Java *Multicast Address* and local port for message reception.

```
// ---------------------------------------------------------------------------
// This function sets up the multicast address and joins the group
// ---------------------------------------------------------------------------
public MulticastSocket multicast_setup(String MULTICAST_ADDR, int MULTICAST_PORT) throws IOException
{
    MulticastSocket multicastSocket = new MulticastSocket(MULTICAST_PORT);
    InetAddress inetAddress = InetAddress.getByName(MULTICAST_ADDR);
    multicastSocket.joinGroup(inetAddress);
    return multicastSocket;
}
```

4.1.1.2 Multicst_Receive Method

After the Multicast address and port has been setup, the *multicast_receive* method is called and is blocked indefinitely until a message arrives. When a *broadcast message* arrives, the method appropriately parses the message and returns the String component of the message.

```
// ---------------------------------------------------------------------------
// This function blocks indefinitely until a message is received on Multicast Port
// ---------------------------------------------------------------------------
public String multicast_receive(MulticastSocket multicastSocket) throws IOException
{
    byte [] temp = new byte [1024];
            DatagramPacket datagramPacket = new DatagramPacket(temp, temp.length);

    // infinitely stuck here until receive a packet
            multicastSocket.receive(datagramPacket);
    String message = new String(datagramPacket.getData(), 0, datagramPacket.getLength());

    return message;
}
```

4.1.1.3 Multicast_Setup Method

When the String component of the message is retrieved, the *Broadcast Agent* will store the information so it can be used by the *Sender* and the *Receiver Agents.*

```
    // -----------------------------------------------------------------------------
// This function writes the message to the specified file
// ----------------------------------------------------------------------------
 public void store_GDF(String filename, String message) throws IOException
 {
      BufferedWriter bufWriter = new BufferedWriter(new FileWriter(filename, true));
      bufWriter.write(message);
      bufWriter.newLine();
      bufWriter.close();
 }
```

4.1.1.4 Reply Method

Finally, a reply message is created and sent to the original sender to inform the node of the existence of this node.

```
// ----------------------------------------------------------------------------
// This function replies to the sender of the broadcast message
// ----------------------------------------------------------------------------
 public void reply(String message) throws IOException
 {
      int index = message.indexOf("/");
      String node_name = message.substring(0,index);

      InetAddress ownAddress = get_own_Inet();
      String host_name = ownAddress.getHostName();
      String msg = "Broadcast_Setup: ".concat(host_name);
      send_msg(node_name, msg);
 }
```

4.1.2 Receiver Agent Implementation

The *Receiver Agent* is used to process different types of incoming messages and relay messages to appropriate nodes if necessary. Using the standard JADE message receiving mechanism listed below, the *Receiver Agent* waits indefinitely until a message arrives.

```
public void action()
{
        ACLMessage msg = myAgent.receive();
        if (msg != null) {
                // Process the message
        }
        else {
        block()
        }
}
```

The *block()* method of the *Behaviour Class* removes the current *Behaviour* from the agent pool. The current *Behaviour* is only interrupted when a message is received and the blocked *Behaviour* is put back in the agent pool and can process the incoming message. This mechanism will not waste CPU by idling for a message to arrive.

When a message does arrive, its *String* component is extracted and the message is processed according to the type, identified by the message header. Currently there are six message types *Receiver Agent* recognizes and they are listed in Table 11.

TABLE 11. MESSAGE HEADERS AND DESCRIPTIONS

Message Type	Message Header	Message Description
Administrative Message	Admin_Setup:	Used to establish virtual connection with neighbours
Broadcast Message	Broadcast_Setup:	Used to establish Global Directory Facilitator (GDF)
Multi Hop Message	Multi_Hop_Message_Header:	Used to route packet according to specified number of hops
Specific Message	Specific_Hop_Message_Header:	Used to route packet according to specified path
Update Hop Message	Update_Hop_Message_Header:	Used to obtain hop information
Update Hop List Message	Update_Hop_List_Header:	Used to update global hop list

The Receiver Agent will process each message differently depending on the Header that the message contains.

4.1.2.1 Administrative Message

The *Administrative Message Header* is used to establish a virtual connection with a specific node. Once a virtual connection is established, the current node will consider the specified node as its neighbour node, thus enabling them to communicate directly. This simulates that the two nodes that are within signal proximity in a wireless environment.

The *Receiver Agent* will use the *ADMIN_HEADER()* method to extract the specified node and stores the information as a neighbour node.

4.1.2.2 Broadcast Message

The *Broadcast Message Header* is used to handle incoming request from new nodes. When a new node is on the network, a *Broadcast Message* will be sent to every node on the network to notify them of its existence. When the *Receivers Agent* receives such a message, it will use the *BROADCAST_HEADER()* method to extract the name of the new node and stores the information as a global node.

4.1.2.3 Multi Hop Message

The *Multi Hop Message* is used to send a message to a specific node on the network if the node is less than a specified number of hops. When a *Multi Hop Message* is received, the *Receiver Agent* will use the *MULTI_HOP_HEADER()* to decrement the number of hops outstanding in the message and relay the message to all of its neighbour nodes. When the number of hops reaches zero, this implies that the node is not within the pre-set number of hops, thus the message is discarded.

4.1.2.4 Specific Path Message

The *Specific Path Message* is used to send a message to a node through a predefined path. When a *Specific Path Message* is received, the *Receiver Agent* uses the *SPECIFIC_HOP_HEADER()* method to re-direct the message to its next destination.

4.1.2.5 Update Hop Message

The *Update Hop Message* is used to update the number of hops each node is away from the current node. When a *Update Hop Message* is received, the *Receiver Agent* uses the *UPDATE_HOP_HEADER()* method to decrement the hop count contained within the message and re-direct the message to every neighbour node. If the hop count is zero, a special *Update Hop List Message* is created and is sent directly back to the originator of this message.

4.1.2.6 Update Hop List Message

The Update Hop List Message is a special type of message used to update the Global Hop List. The *Receiver Agent* uses the *UPDATE_HOP_LIST_HEADER()* method to update its G*lobal Hop List.* The list stores all nodes on the network and the number of hops they are away from the current node. This information is crucial in determining the best routing method that should be used to transmit the message. Different wireless scenarios can also be used based on this information.

The interactions between the *Receiver Agent* and JADE software packages are illustrated in Figure 34.

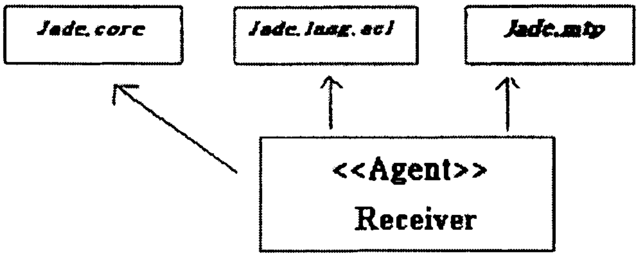
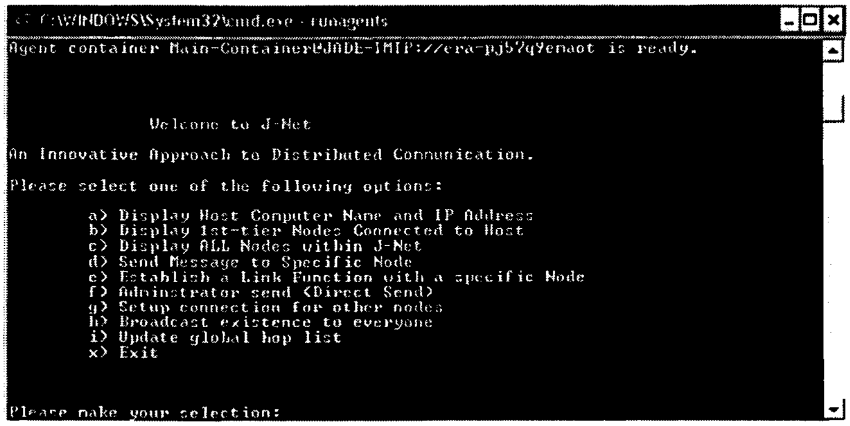


FIGURE 34. RECEIVER AGENT INTERACTION WITH JADE SOFTWARE PACKAGES

4.1.3 Sender Agent Implementation

The *Sender Agent* contains the entry point for the end user to operate a JADE node. The simplified user interface has functionalities ranging from displaying all available nodes currently on the network to communicating with a specific node through a user-defined routing method. Messages can be sent either directly to the destination node, or routed through a number of predefined methods. Figure 35 illustrates the user interface.



```
Agent container Main-Container@JADE-IMTP://era-pj5?q9enaot is ready.



             Welcome to J-Net

An Innovative Approach to Distributed Communication.

Please select one of the following options:

        a) Display Host Computer Name and IP Address
        b) Display 1st-tier Nodes Connected to Host
        c) Display ALL Nodes within J-Net
        d) Send Message to Specific Node
        e) Establish a Link Function with a specific Node
        f) Adminstrator send (Direct Send)
        g) Setup connection for other nodes
        h) Broadcast existence to everyone
        i) Update global hop list
        x) Exit


Please make your selection:
```

FIGURE 35. USER INTERFACE

There are three classes within *Sender Agent*. They are *Display(), J_Node()* and *Route()*

4.1.3.1 Class Display()

The *Display()* class is used to output critical system information onto the screen for the end user. From this information the user can then make appropriate decision regarding message routing and determine the state of the network. Table 12 lists the methods of this class and their functionalities.

TABLE 12. CLASS DISPLAY() METHOD DESCRIPTION

Method Name	**Method Description**
Host_info()	Displays local host name and IP
Neighbour_nodes()	Display all nodes with virtual connection to current node
All_nodes()	Display all nodes on the JADE network
Hop_nodes()	Display all nodes at specified number of hops away from current node

4.1.3.2 Class Route()

The *Route()* class implements the routing algorithms that the end users can choose to send the message. Currently, there are three routing algorithms: *Direct*, *Maximum Hop* and *Specific Path*.

- **Direct Algorithm:** Messages are directly sent to the receiving node, no message header is needed. This simple algorithm is used to send messages directly to neighbour nodes.

- **Multi Hop Algorithm:** Messages are sent to the specified node if the node is within the maximum specified number of hops. A *Multi Hop Header* and maximum hops information are attached to the message body so receiving nodes can properly process and relay the information onto the next node. A message sent by the Multi Hop Algorithm has the following format:

  Multi_Hop_Message_Header: max_hop#dest_node$msg_body

- **Specified Path Algorithm:** Messages are sent to the specified node through a path specified by the end user. A Specific Path Header and a series of relay nodes specified by the user are attached to the message. A message sent by the Specified Path Algorithm has the following format:

  Specific_Path_Message_Header: dest_1# dest_2# dest_3 $msg_body

This class can be expanded easily by future developers to implement additional routing algorithms.

4.1.3.3 Class J_Node()

The *J_Node()* class contains the entry point for the end users and performs all initializations before a JADE node is able to communicate with other nodes on the network. The *J_Node()* class is also responsible for setting virtual links with any node on the network, broadcasting its existence onto the network and sending update hop messages to update its global hop list.

- **Virtual Connection:** A JADE node is able to virtually connect with any other node on the network to become neighbour nodes. Only neighbour nodes are allowed to send messages directly, otherwise intermediary nodes are used to relay messages. A request for virtual connection message has the following format:

  Admin_Setup: host_name

  When the receiving node accepts the request, the sender node is added to its list of neighbour nodes. The two nodes have now become neighbours and is able to communicate directly.

- **Broadcasting Existence:** A JADE node must make itself known to others on the network. This is achieved by using the *Java MulticastSocket Class* to broadcast to all JADE nodes listening at a predetermined port and address. Address ***"230.0.01"*** and ***Port*** *7777* are used to receive Multicast messages on the JADE network.

- **Update Hop Message:** An *Update Hop Message* provides the node with the number of hops all nodes on the network are away from the current node. This information is crucial in determining the best routing method to be used and provides users with the whereabouts of all nodes on the network.

  An *Update Hop Message* has the following format:

  Update_Hop_Message_Heder: original_sender#current_count#original_count

Table 13 summarizes the core methods used in *J_Node() class* to implements its functionalities.

TABLE 13. CLASS J_NODE METHOD DESCRIPTION

Method Name	**Method Description**
~~initialize()~~	~~Initializes JADE node~~
main_menu()	Entry point for end user. Allows for complete operation of JADE node
establish_connection()	Establish virtual connection with another JADE node
remote_setup()	Remotely establish virtual connections between ANY two JADE nodes
broadcast()	Broadcast existence onto JADE network
update_hop_list()	Dynamically update number of hops all nodes are away from current JADE node

The interactions between the *Receiver Agent* and JADE software packages are illustrated in Figure 36.

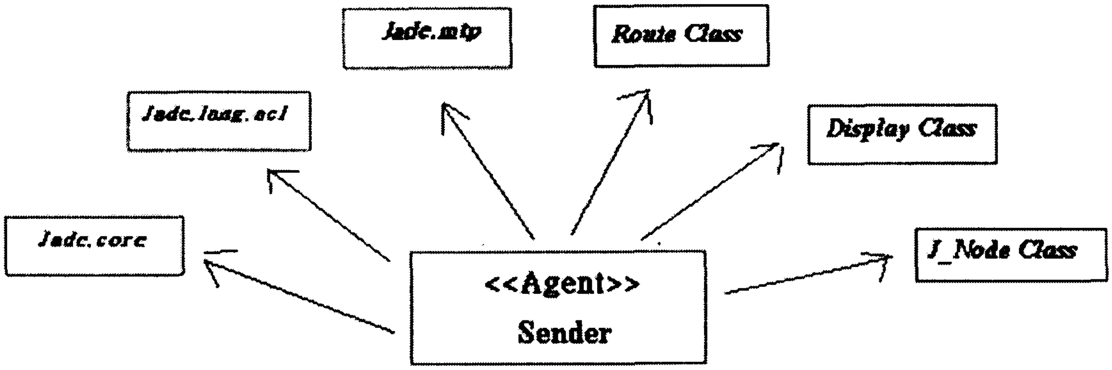


FIGURE 36. SENDER AGENT AND JADE SOFTWARE PACKAGES INTERACTIONS

4.2 JXTA Implementation

Like JADE, the JXTA software platform has limitations that need to be addressed. The extensions discussed in Chapter 3 must be implemented to achieve a better distributed system.

Figure 37 again shows the extensions required to the JXTA Core Layer.

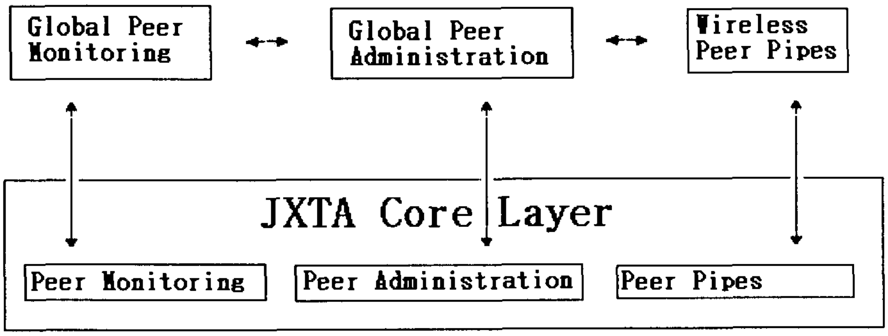


FIGURE 37. MODIFIED JXTA FRAMEWORK FOR AN IMPROVED DS

- The *Global Peer* Monitoring maintains a current list of all nodes currently available on the network. It also handles broadcasted messages to/from other nodes.
- The *Wireless Peer Pipes* extension is used to restrict the sending of message to only nodes available according to the user-defined scenario.

- The *Global Peer Administration* extension is used to handle the added administrative overhead. It also initializes and supervises the JXTA node. An entry point is contained in the *GPA* to allow the developer to operate the JXTA node.

In this research, the extensions are accomplished by implementing four distinct Java Classes; *PipeListener(), PipeSender(), PipeComm(), PeerRoute().* A fifth class, *PeerDisplay(),* is used to output network information.

The *PipeListener()* and *PipeSender()* classes are used in conjunction to model the *Wireless Peer Pipe.* The *PipeComm()* and the *PeerRoute()* classes are used to model the *Global Peer Administration* and *Global Peer Monitoring* respectively.

4.2.1 Class *PipeListener()*

The *PipeListener()* class creates input pipes used to receive messages. This task is accomplished by:

- Create and bind to input pipe
- Register pipe and publish the pipe advertisement
- Wait indefinitely until an message arrives

4.2.1.1 Input Pipe Creation and Binding

The method *bind_input_pipes()* is called to create and bind the peer to an input pipe. JXTA uses XML files as advertisements. The advertisement is first read then bound to the node with the following command:

```
FileInputStream is = new FileInputStream(XML_filename);
    pipeAdv = (PipeAdvertisement) AdvertisementFactory.newAdvertisement(MimeMediaType.XMLUTF8, is);
is.close();
    pipeIn[i] = pipe.createInputPipe(pipeAdv, this);
```

4.2.1.2 Pipe Registration and Advertising

After successfully creating and binding to the input pipe, the node must be registered as a *PipeMsgListener* to receive messages. This allows the receiving node to infinitely wait for a message to arrive, but would not block the CPU from performing other tasks. When a message does arrive, a *pipeMsgEvent* is generated and interrupts the CPU from its activities to process the message.

4.2.1.3 Message Reception and Processing

This *pipeMsgEvent(PipeMsgEvent event)* method is called asynchronously when a message is received on the input. The receiving node then must properly process the incoming message to obtain its *String* component. This is achieved with the use of the following:

```
// grab the message from the event
  msg = event.getMessage();
  if (msg == null) {
     return;
  }

// get all the message elements
Message.ElementIterator enum = msg.getMessageElements();
if (!enum.hasNext()) {
  return;
}

// get the message element named SenderMessage
MessageElement msgElement = msg.getMessageElement(null, SenderMessage);
String received = msgElement.toString();
```

After the message has been correctly received, it will be processed to determine its type and what further action, if any, should be taken. Identical to processing a message in JADE, the types of messages are determined by the message header. Again, currently there are six message types that the *PipeListener* class recognizes, as listed in Table 14.

TABLE 14. MESSAGE HEADERS AND DESCRIPTIONS

Message Type	**Message Header**	**Message Description**
Administrative Message	Admin_Setup:	Used to establish virtual connection with neighbours
Broadcast Message	Broadcast_Setup:	Used to establish a global peer directory
Multi Hop Message	Multi_Hop_Message_Header:	Used to route packet according to specified number of hops
Specific Message	Specific_Hop_Message_Header:	Used to route packet according to specified path
Update Hop Message	Update_Hop_Message_Header:	Used to obtain hop information
Update Hop List Message	Update_Hop_List_Header:	Used to update global hop list

The mechanism of processing each message type is identical to its JADE counterpart. Detailed descriptions of each message type can be found in Section 4.1.

The sequences of interactions between the *PipeListener()* class and JXTA protocols are illustrated in Figure 38.

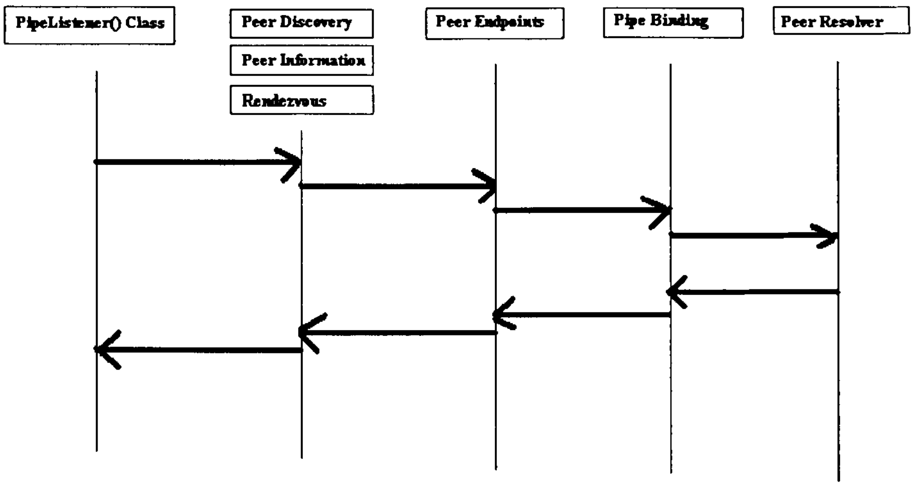


FIGURE 38. INTERACTIONS BETWEEN PIPELISTENER() AND JXTA PROTOCOLS

4.2.2 Class *PipeSender* Implementation

The *PipeSender* class creates a dedicated output pipe to a specified receiving peer and sends messages on it. This is accomplished by:

- Creating an output pipe with the specified receiving node.
- Triggering an *event* to send the message.

4.2.2.1 Output Pipe Creation

The *run()* method is called to initialize an output pipe to a specific receiving peer. An XML file is created and parsed as a pipe advertisement and the node attempts to create and bind itself to the output pipe. The *getRremoteAdvertisement* method of the *Discovery Protocol* attempts to locate the specified receiving peer. Once the receiving node is located, the two end-points of the communication pipe will be resolved and a dedicated pipe is now in place for communication.

```
FileInputStream is = new FileInputStream(dest_node);
pipeAdv = (PipeAdvertisement) AdvertisementFactory.newAdvertisement(MimeMediaType.XMLUTF8, is);
is.close();

// obtain receiving peer information
discovery.getRemoteAdvertisements(null, DiscoveryService.ADV, null, null, 1, null);
// create output pipe asynchronously
// Send out the first pipe resolve call
pipe.createOutputPipe(pipeAdv, this);
```

4.2.2.2 Message Sending

Messages placed on this dedicated pipe will asynchronously trigger an *event* and invoke the *pipeMsgEvent* method. Similar to the *PipeListener* class, a dedicated output pipe will not block the CPU from other activities. When a message is to be sent, a *pipeMsgEvent* is generated and interrupts the CPU from its activities to process the message.

```
OutputPipe op = event.getOutputPipe();
Message msg = null;

    try {
       msg = new Message();
       StringMessageElement sme = new StringMessageElement(SenderMessage, message , null);
       msg.addMessageElement(null, sme);
       op.send(msg);
    } catch (IOException e) {
       System.out.println("failed to send message");
       e.printStackTrace();
       System.exit(-1);
    }
    op.close();
```

The sequences of interactions between the *PipeSender()* class and JXTA protocols are illustrated in Figure 39.

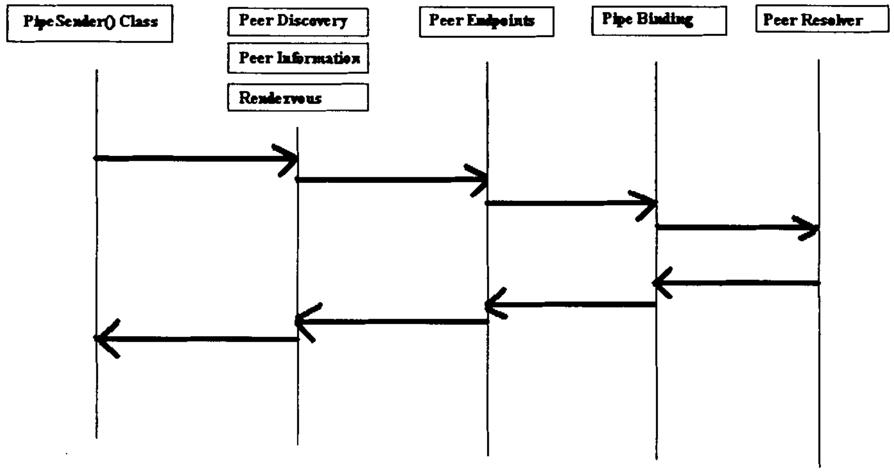


FIGURE 39. INTERACTIONS BETWEEN PIPESENDER() AND JXTA PROTOCOLS

4.2.3 Class *PipeComm()*

The *PipeComme()* class contains the entry point for the end users and performs all initializations and tasks that a JXTA node requires for communication. It utilizes the *PipeSender( ) Class* and *PipeListener() Class* for message sending and reception.

The *PipeComm()* class is also responsible for setting virtual links with any node on the network, advertising its existence onto the network and sending update hop messages to update its global hop list.

- **Initialization:** By calling the initialization method, the node will obtain a valid peer group ID, peer group name, as well as the name and ID of the current peer. The peer ID is a randomly generated 256-byte number. By default all JXTA peers belongs to the *netpeergroup*.

```
try {
        // create, and Start the default jxta NetPeerGroup
      netPeerGroup = PeerGroupFactory.newNetPeerGroup();
}
      catch (PeerGroupException e) {
      // could not instantiate the group, print the stack and exit
      System.out.println("fatal error : group creation failure");
      e.printStackTrace();
      System.exit(1);
      }
```

- **Virtual Connection:** A JXTA node is able to virtually connect with any other node on the network. Only neighbour nodes are allowed to send messages directly, otherwise intermediary nodes are used to relay messages. A request for virtual connection message has the following format:

  Admin_Setup: host_name

  When the receiving node accept the request from the *PipeListener()* class, the sender node is added to its list of neighbour nodes. The two nodes have now become neighbours that are able to communicate directly.

- **Broadcasting Existence:** A JXTA node must make itself known to others on the network. This is achieved by publishing the node's advertisement once the node has been successfully created. Once published, other nodes on the JXTA network are able to remotely locate this node.

```
try {

    // publish this advertisement
    //(send out to other peers and rendezvous peer)
    discoSvc.remotePublish(adv, DiscoveryService.PEER);
    System.out.println("Peer published successfully.");
}
catch (Exception e) {
    System.out.println("Error publishing peer advertisement");
    e.printStackTrace();
return; }
```

- **Update Hop Message:** An *Update Hop Message* provides the node with the number of hops from the current node to all peers on the network. This information is crucial in determining the best routing method and provides users with the location of all nodes on the network.

  An *Update Hop Message* has the following format:

  Update_Hop_Message_Heder: original_sender#current_count#original_count

The following table summarizes the core methods used in *PipeComm()* class to implement its functionalities.

TABLE 15. CLASS PIPECOMM () METHOD DESCRIPTION

Method Name	**Method Description**
Initialize()	Initializes JXTA node
Main_menu()	Entry point for end user. Allows for complete operation of JXTA node
Establish_connection()	Establish virtual connection with another JXTA node
Remote_setup()	Remotely establish virtual connections between ANY two JXTA nodes
broadcast()	Broadcast existence onto JXTA network
Update_hop_list()	Dynamically update number of hops all nodes are away from current JXTA node

The sequences of interactions between the *PipeComm()* class and JXTA protocols are illustrated in Figure 40.

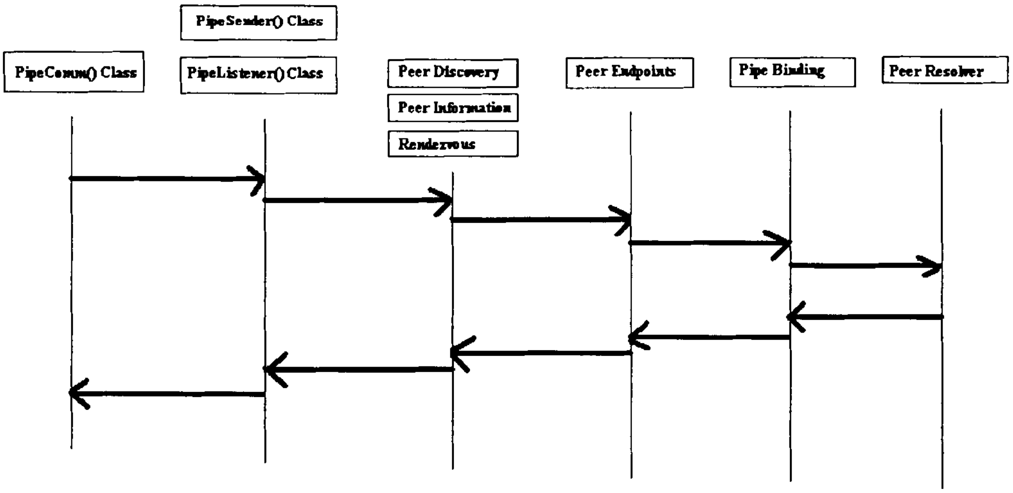


FIGURE 40. INTERACTIONS BETWEEN PIPECOMM() AND JXTA PROTOCOLS

4.2.4 Class *PeerRoute()*

The *PeerRoute()* class implements the different routing algorithms that the end users can choose to send the message. Again, there are three routing algorithms: *Direct*, *Maximum Hop* and *Specific Path*.

- **Direct Algorithm:** Messages are directly sent to the receiving node, no message header is needed. This algorithm is used to send messages directly to neighbour nodes.

- **Multi Hop Algorithm:** Messages are sent to the specified node provided that the node is within the maximum specified number of nodes. A *Multi Hop Header* and maximum hops information are attached to the message body so that receiving nodes can properly process and relay the information onto the next node. A message sent by Multi Hop Algorithm has the following format:

Multi_Hop_Message_Header: max_hop#dest_node$msg_body

- **Specified Path Algorithm:** Messages are sent to the specified node through a path specified by the end user. A Specific Path Header and a series of relay nodes specified by the user are attached to the message. A message sent by the Specified Path Algorithm has the following format:

  Specific_Path_Message_Header: dest_1# dest_2# dest_3 $msg_body

This class can be expanded easily by future developers to implement additional routing algorithms.

4.2.5 Class *PeerDisplay()*

The *PeerDisplay()* class is used to output critical system information to the screen for the end user. From this information the user can then make appropriate decisions regarding message routing and determine the state of the network. Table 16 lists the methods of this class and their functionalities.

TABLE 16. CLASS DISPLAY() METHOD DESCRIPTION

Method Name	Method Description
Host_info()	Displays local host name and IP
Neighbour_peers()	Display all nodes with virtual connection to current node
All_peers()	Display all nodes on the JADE network
Hop_peers()	Display all nodes at specified number of hops away from current node

5 PLATFORM ANALYSIS

5.1 Qualitative Analysis

The traditional centralized architectures are inherently more focused on simplicity than on scalability and robustness. A distributed system requires the creation of a network that is scalable, robust and inexpensive to maintain. However, the complexity of software implementation of a distributed system is much greater than a centralized system.

JADE and JXTA are distributed software platforms that facilitate the creation of distributed networks by providing developers with ready-made protocols and software platforms. Both platforms are built with a similar purpose, but contain key similarities and differences in areas such as scalability, interoperability, and platform complexity.

5.1.1 Platforms Scalability

Both JADE-based and JXTA-based distributed systems are built for expansion. A key advantage of a true distributed system over a conventional centralized system is the unrestricted ability to expand and add new nodes. In a true Distributed System, additional network resources are added and utilized by the network with the addition of every node.

In JADE, agents residing on remote containers are dependent on the AMS and the DF that resides in the *main container*. *Remote containers* are critically dependent on the agents of the *main containers* and their services. The failure of the *main container* would also indicate the failure of the entire JADE network. The state of a JADE network is dependent on the continual operation of the host on which the *main container* resides.

JXTA on the other hand does not use remote containers. Failure of one node will not have a catastrophic effect on the overall system. No JXTA node is critically dependent on another JXTA node. However, the extensive use of Rendezvous peers that reside on a JXTA node may result in bottlenecks in localized areas. If a network grows while the number of Rendezvous peers remains constant, the amount of processing required by these nodes will grow exponentially. Network latency and efficiency will also increase significantly due to these strained peers.

The extensions implemented by this thesis for JADE eliminate the use of *remote container* in JADE to provide better distributiveness. This eliminates the central influence of the *main container*. Each host is completely independent of other hosts and a failure of one host will not have a catastrophic effect on the network, as illustrated in Figure 41.

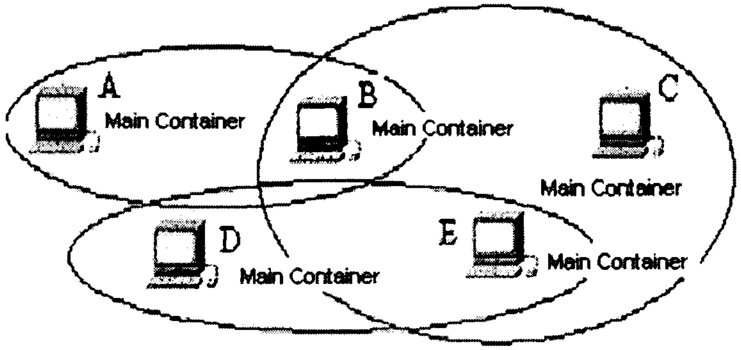


FIGURE 41. JADE IN A VIRTUAL WIRELESS ENVIRONMENT

In JXTA, this extension is already embedded with the standard version. Each JXTA peer is a unique entity that is not critically dependent on any other JXTA peer. Also, each JXTA peer is also a Rendezvous peer to reduce latency and maximize efficiency on the network.

5.1.2 Interoperability

A true distributed system should be designed to interoperate with all nodes on the network, regardless of the distributed platform on which it was built. The communication language and messaging format should be consistent to ensure standardization among all nodes.

Although JADE is built to be a FIPA-compliant system that is aimed to be interoperable with other FIPA-compliant platforms, issues such as degree of compliancy, addressing method, and messaging architecture still exist among FIPA-compliant systems [16].

JADE agents cannot easily communicate with agents from other FIPA-compliant systems. The FIPA specification leaves many issues as *"implementation specific"* that results in non-compliancy between platforms [16].

JXTA on the other hand is not a FIPA-compliant platform and thus does not follow the standardization set forth by FIPA. It is a standalone system without the ability to easily integrate with other distributed platforms for interoperability. It is mainly a closed network that functions only with other JXTA nodes.

In the standard version of JADE without extensions, agents from different JADE networks are unaware of each other and thus unable to interact. They are closed networks with no interactions between multiple *main containers*. The extensions implemented in this thesis allow remote JADE nodes to join the existing JADE network to create a vast yet robust and scalable JADE network. Nodes are able to dynamically discover each other and are aware of all nodes currently available on the network.

Unfortunately, even with the extensions implemented by this project, both JADE and JXTA remain relatively closed platforms that have very limited interoperability with other software platforms. A JXTA peer cannot easily interact with a JADE node to provide the same service to the network. It will be interesting to see the development of a universal software gateway to interconnect multiple distributed networks built on different software platform to interact in a distributed environment.

5.1.3 Messaging Architecture

The XML language is used extensively in JXTA. It is a widespread platform-independent form of data representation [18]. It is used to represent *advertisements, messages* and *identifiers.*

The XML message used may reduce network efficiency. Its mandatory 256-bit peer ID and path specifications implies that an "*empty*" message that has no application-specific payload can easily reach 1 KB in size and thus affecting the performance of the message exchange. Also, the complex messaging architecture of JXTA that involves XML parser and several layers of abstraction will add significant overhead and affect the efficiency of the messaging framework [19].

In the FIPA-compliant JADE, Agent Communication Language (ACL) messages are used for message representation. ACL is a language "*with precisely defined syntax, semantics and pragmatics that is the basis of the communication between independently designed and developed agent platforms*" [21]. An ACL message is an ASCII string consisting of communicative act type and parameters [21].

The use of ACL messages greatly simplifies the communication between agents. Messages are easily parsed and understood by the receiving agent. It is shown in Section 5.2 that the JADE messaging architecture is more efficient and robust when compared to the JXTA messaging architecture.

5.1.4 Platform Complexity

The platform complexity and thus the learning curve of a JXTA system is much higher than a JADE system. We found that in JADE, concepts and operations are easier to understand and carry out than in a JXTA system. Less system configuration is needed to operate a JADE system.

Because a JXTA system offers many customizable functions that a developer needs to choose, this amounts to a great burden to people unfamiliar with JXTA to get started initially. Also, the complex messaging architecture of JXTA that involves XML parser and several layers of abstraction will add significant overhead and affect the efficiency of the messaging framework [19]. Extensive use of Rendezvous peers will also create bottlenecks within the network.

The extensions implemented by this project enabled every peer in JXTA to be a Rendezvous peer. This will decrease latency since peers will no longer be required to query neighbour peers for route or network information. The information is now cached internally. Therefore, the failure of any peer should not have create partial failure of a JXTA network.

5.1.5 Protocols

Both JADE and JXTA utilize Java-based software protocols and packages for the development of a Distributed System.

5.1.5.1 JADE Software Packages

The JADE software packages give application programmers *"ready-made functionality and abstract interfaces for custom application dependent tasks"* [17]. Table 17 briefly describes the different JADE software packages.

TABLE 17. JADE SOFTWARE PACKAGE DESCRIPTION

Software Package	**Description**
Jade.core	• Implements the kernel of the system. Includes the *Agent* class that must be extended by application programmer. *Behaviour* class hierarchy contained in the sub-package implements the logical tasks that can be composed in various ways to achieve complex tasks.
Jade.lang.acl	• Provides Agent Communication Language according to FIPA Standard Specifications.
Jade.domain	• Contains all Java class that represent Agent Management System defined by FIPA standards
Jade.gui	• Contains generic classes useful to create GUIs
Jade.mtp	• Contains the Message Transport Protocol that should be implemented to readily integrate with the JADE framework
Jade.proto	• Provides classes to model standard FIPA interaction protocols *(fipa-request, fipa-query, fipa-contract-net)*

Figure 42 illustrates the dependencies between the various Jade software packages.

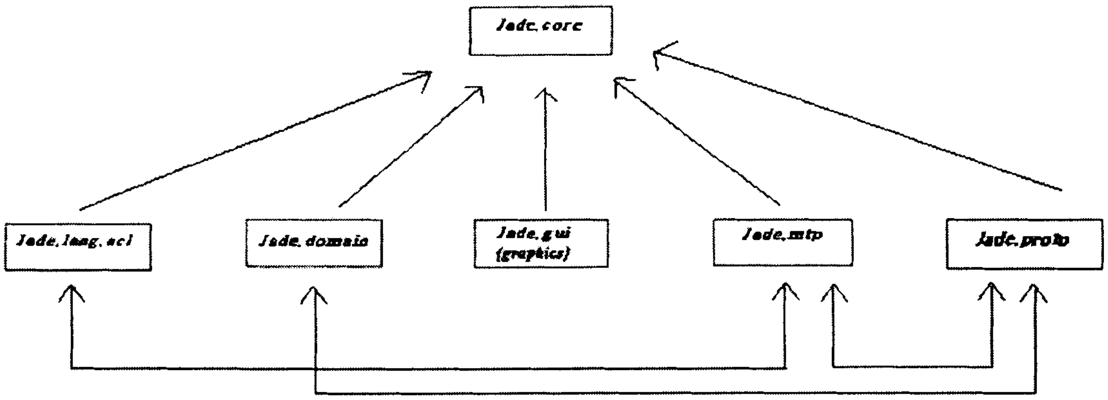


FIGURE 42. JADE SOFTWARE PACKAGE INTERACTIONS

Figure 34 – 36 in Section 4.1 illustrates the interactions of the JADE extensions to the standard JADE software packages.

5.1.5.2 JXTA Protocols

The JXTA protocols have been specifically designed for *"ad hoc, pervasive, and multi-hop network computing" [20]*. By using the JXTA protocols, nodes in a JXTA application can cooperate to form *"self-organized and self-configured peer groups independently of their positions in the network (edges, firewalls), and without the need of a centralized management infrastructure." [20]*

Table 18 briefly describes the different JXTA software protocols.

TABLE 18. JXTA PROTOCOLS AND DESCRIPTIONS

JXTA Protocol	**Description**
Peer DiscoveryProtocol	- Resource Search
Peer ResolverProtoco	- Generic Query Service
Peer Information Protocol	- Monitoring
Rendezvous Protocol	- Message Propagation
Peer Membership Protocol	- Security
Pipe Binding Protocol	- Addressable Messaging
Peer Endpoint Protocol	- Message Routing

Source: Developer [20]

Figure 43 illustrates the sequences of interactions between the different JXTA software protocols

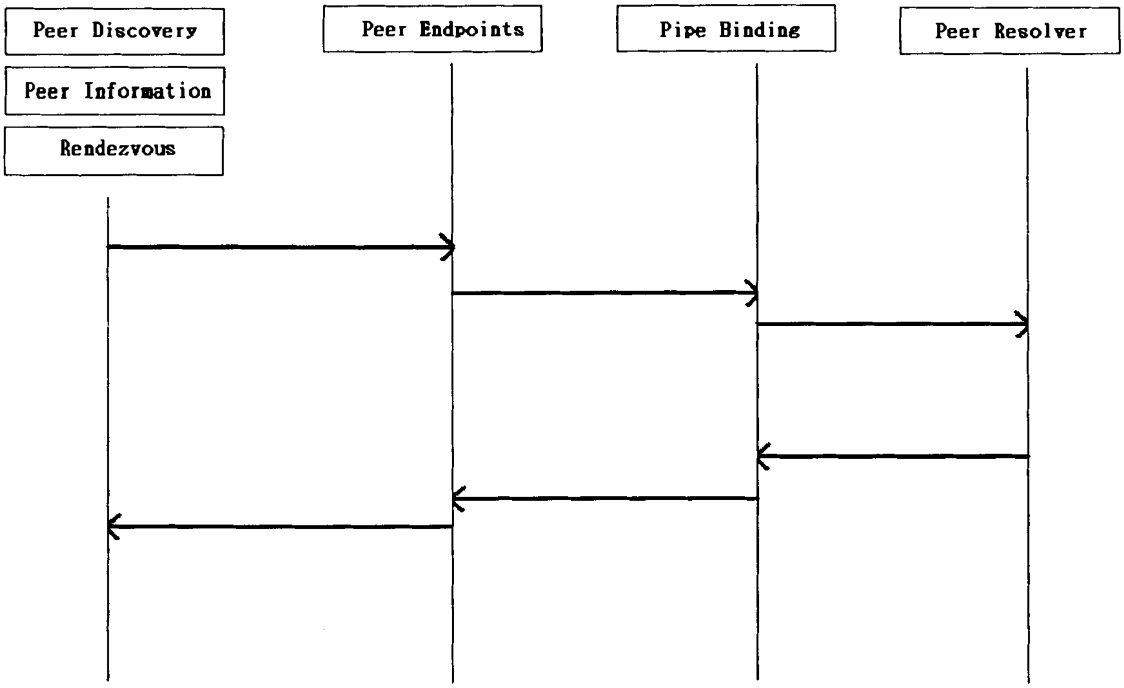


FIGURE 43. JXTA PROTOCOL SEQUENCE DIAGRAM

The components of the JXTA Core Layer are extended to improve upon the existing JXTA environment. The new classes necessary for the extensions and their interactions to the JXTA protocols are illustrated in Figure 44.

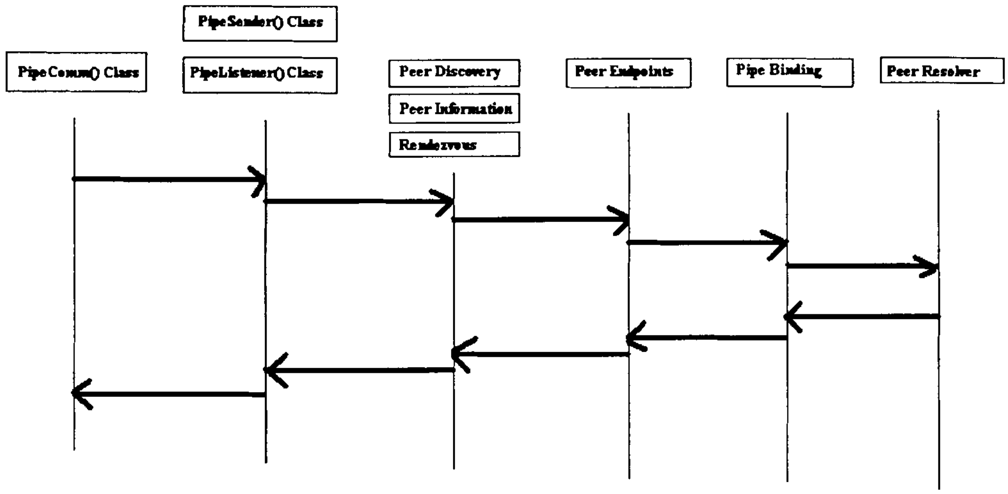


FIGURE 44. INTERACTIONS BETWEEN PIPECOMM() AND JXTA PROTOCOLS

5.1.6 Agent Migration

In the JADE system, all agents except the AMS and the DF are free to migrate to and from different containers and platforms. This ability allows developers more freedom and possibility when designing a Distributed System. Agents can move away from congested areas and perform their tasks in areas where network is not constrained. The JADE messaging architecture internally takes care of addressing issues and messages are sent to the containers in which the receiving agent resides.

However, in a JXTA system, a peer is physically tied to the residing host (PC, PDA, cell-phone). The host is free to move around a JXTA network (e.g., a wireless PDA), but the software entity that resides within the host is unable to migrate from one host to another.

Table 19 illustrates some key differences between JADE and JXTA when utilized in a distributed environment.

TABLE 19. COMPARISON OF JADE AND JXTA IN DISTRIBUTED SYSTEM

	JADE	JXTA
Messaging Architecture	Relatively simple. Uses IMTP for Inter-platform and RMI for Intra-platform communication	Uses XML parser and several layers of abstraction. Pipes used for communication. Significant overhead
Node/Peer Migration	Agents able to freely move to different containers	Peers are embedded within the host they reside in
Distributiveness	Limited by the *main container*. Remote containers dependent on main container.	Unrestricted scalability. Each peer is uniquely identified and independent.
Platform Complexity	Very manageable and coherent	More sophisticated and steep learning curve.
FIPA Compliance	Yes	No
Interoperability	FIPA-Compliant system. Unable to communicate with other agents on different distributed system software platforms	Standalone system without FIPA Compliancy

5.2 Quantitative Analysis

Although both JADE and JXTA are distributed software platforms aimed to facilitate the creation of distributed systems, their respective performances in a distributed system may vary significantly. This section briefly compares quantitatively scalability and performance of both software platforms.

In a distributed system, nodes may be requested to act as relay nodes to forward messages and requests onto the next node. The efficiency and latency involved in this multi-hop transaction depends heavily on the node's user-defined routing logic and system's hardware and software.

To ensure a fair comparison, it is assumed that the all nodes have identical routing logic and system hardware and software. The added latency involved in a multi-hop transaction will then only be platform dependent, since both JADE and JXTA are Java-based and utilize the identical system setup

As a result, multi-hop latency across multiple nodes can be omitted when comparing the two platforms quantitatively, since the two platforms will be subjected to identical lag.

5.2.1 Test Setup

In the following experiments, two hosts on a 100 Mbps LAN. The two hosts utilize identical system hardware and software configuration, as illustrated in Figure 45.

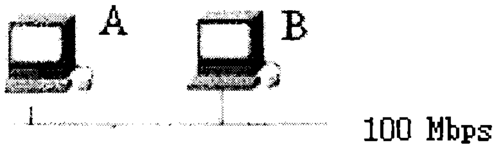


FIGURE 45. LOCAL AREA NETWORK TEST ENVIRONMENT

For each experiment, the Sender sends a payload to the Receiver, and the Receiver replies with the identical message. The time between the sending of the initial message and the reception of the reply message is defined as the *Round Trip Time (RTT).* The test is then repeated 1000 times and the average time is used.

5.2.2 Multiple Agent-Pairs on Same Host

Scalability is a very important indication of the competency of a particular distributed software platform. In this test, varying number of agent-pairs all residing on the same host are used for the message exchange. The Sender agents exchange messages with Receiver agents residing on the same host.

In the standard JADE without extensions, the agent-pairs residing on a single host could either be in the same or different containers. However, in the extended JADE, the host will only accommodate the *main container*, the use of *remote containers* is not allowed. All agents residing on a single host will reside in the *main container* of the host.

The results of the standard JADE message exchanges are illustrated in Figure 46 and Figure 47.

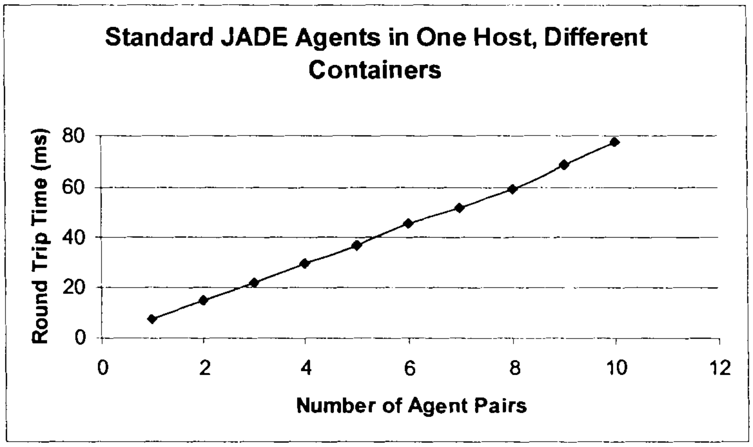


FIGURE 46. STANDARD JADE AGENTS IN SINGLE HOST, DIFFERENT CONTAINERS [19]

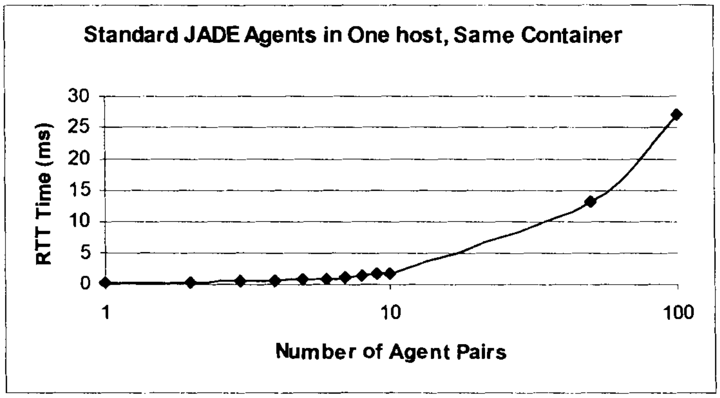


FIGURE 47. STANDARD JADE AGENTS IN SINGLE HOST, SAME CONTAINER [19]

In Figure 48, the results of both the extended JADE and JXTA are presented when multiple agent pairs residing on the same host (same container for JADE).

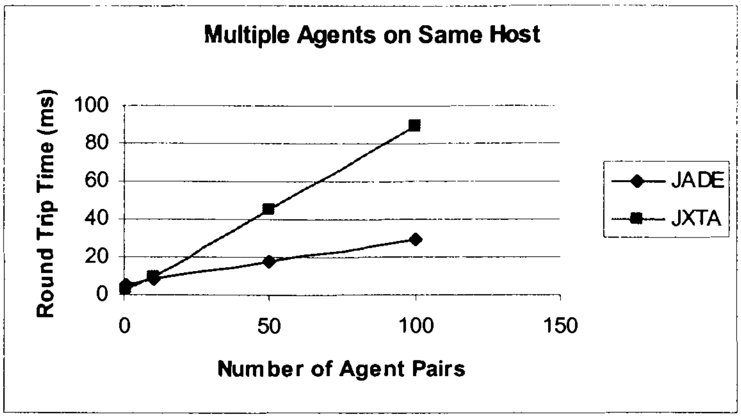


FIGURE 48. VARIABLE AGENT-PAIR ON SAME HOST COMPARISON [19]

From the results, we see that the *RTT* for JADE is very similar to Figure 47, which is expected. All agents in the extended JADE reside in the *main container,* thus creating the identical scenario to Figure 47.

When *RTT* of JXTA and JADE are compared, we see that the communication time rises linearly with increasing number of agent-pairs. The rate of increase for a JXTA agent-pair is significantly higher than that of a JADE agent-pair.

5.2.3 Multiple Agent-Pairs on Different Host

In this test, varying number of agent-pairs that reside on different hosts are used for the message exchange. The Sender agents exchange messages with Receiver agents that reside on the same host. This test will demonstrate the scalability of a particular distributed software platform when the Sender agent and the Receiver agent do not reside on the same host. The results are illustrated in Figure 49.

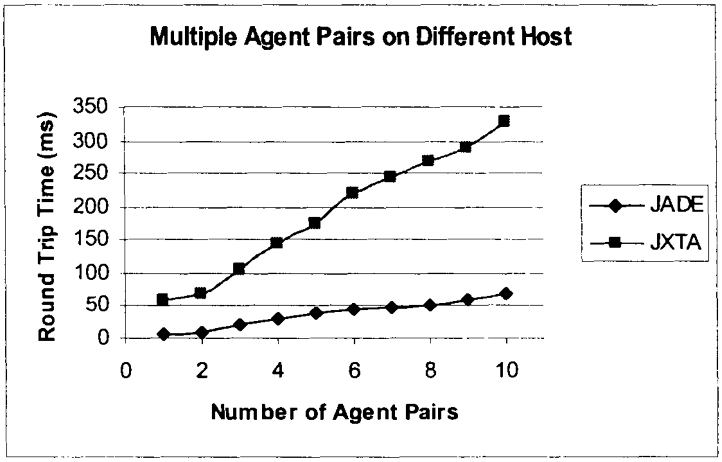


FIGURE 49. VARIABLE AGENT-PAIR ON DIFFERENT HOST COMPARISON [19]

From the results, we see that the communication time somewhat rises linearly with increasing number of agent-pairs. Again, the rate of increase for a JXTA agent-pair is significantly higher than that of a JADE agent-pair.

5.2.4 Multiple Message Size Comparison

Network efficiency under varying message load is also an important indication of the competency of a particular software platform. In a Distributed System, nodes are constantly exchanging messages and requests. The efficiency of the overall network depends heavily on the minimization of latency between message exchanges.

In this scenario, a sender-receiver pair residing on *different hosts* is setup for the message exchange of varying sizes. The results are illustrated in Figure 50.

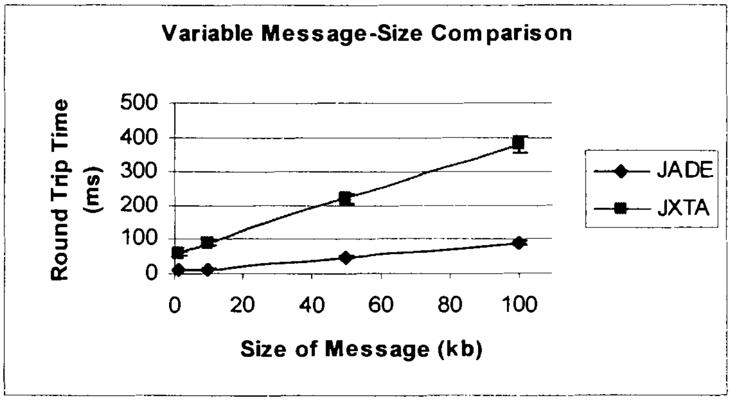


FIGURE 50. VARIABLE MESSAGE SIZE COMPARISON [19]

From the results, we see that again the communication time rises linearly for a linear increase in load for both platforms. However, the rates at which they rise differ significantly.

5.2.5 Quantitative Result Discussion

As the results of the three tests suggest, JADE seems to be a better distributed software platform when compared to JXTA under the specified conditions. In all three test scenarios, the performance of JADE is significantly better than that of JXTA. Not only is JADE more capable under varying message load, but it is also more efficient when the receiving agents reside both on the same and on different hosts.

However, one important advantage that JXTA has over JADE is its unrestricted scalability. The lack of a centralized management system enables a JXTA system to be highly scalable. Although the extensive use of Rendezvous peers in JXTA may hinder overall system performance, a JXTA network is built on the concept of unrestricted scalability.

JADE on the other hand relies heavily on the centralized *main container* to handle administrative issues for system expansion. Agents residing on *remote containers* rely critically on the continual operation of the AMS and DF of the *main container.* Scalability in JADE is *"the ability to keep up good performance when the load is increased"* [19].

Due to the JADE's central main container, agents are efficiently located by querying the AMS and the DF. In JXTA, extensive communication may be needed between querying agents and multiple Rendezvous peers to locate the receiving agent before a communication pipe can be established between the agent-pair. Also, the complex messaging architecture of JXTA that involves XML parser and several layers of abstraction adds significant overhead and affect the efficiency of the messaging framework.

5.3 Summary, Concluding Remarks and Future Research

5.3.1 Summary

Distributed systems offer a useful approach for resolving critical networking limitations that result from the use of centralized topologies. Scalability and fault-tolerance can be increased by utilizing a distributed system, however, the complexity of a distributed system grows exponentially as the number of nodes increase.

JADE and JXTA are distributed software platforms that facilitate the development of distributed systems. Both are Java-based software that serve as middleware to provide low-level communication transport and message encoding. Software developers can therefore concentrate on the development of complex models and reasoning that constitute the distributed system, rather than low-level communication.

This project examined the architectures of JADE and JXTA. We also noted their strength and weaknesses in a distributed environment, as shown in Table 23 and Table 24.

Table 20. Advantages and Disadvantages of JADE in a Distributed System

Advantages:	**Disadvantages**
• Open source, completely written in JAVA and FIPA-compliant • Serves as middleware to deal with communication transport and message encoding • Concise and efficient software architecture • All agent tasks modeled as *Behaviors* objects for simple implementation of complex tasks • Ability for agents to migrate from container to container, regardless of platform	• Cannot define specific path to receiving node • Dependence on the *main container* for communication • Unable to simulate different transmission scenarios

TABLE 21. ADVANTAGES AND DISADVANTAGES OF JXTA IN A DISTRIBUTED SYSTEM

Advantages:	**Disadvantages**
• No extensive knowledge of underlying distributed domain • Support large number of potential peers with no central management system • Network resources distributed among multiple machines • Automatic protocol translation for communication between peers with different protocols • Cached network information reduces search time for service requests	• Developers unaware of mechanisms and path used for message transport. • Sizeable XML messages, XML parser and several layers of abstraction may lead to network inefficiency. • Dependence on specific types peers for routing, messaging and requests between peers. • Increased memory overhead by caching network configuration for every peer

Both JADE and JXTA have limitations in their current form. In JADE, the over-reliance of the AMS and the DF of the *main container* restricts the scalability and the fault-tolerance of a JADE system. Agents residing on *remote containers* are critically dependent on the host on which the *main container* resides. In JXTA, although lacking a centralized management system, the extensive use of Rendezvous peers limits the efficiency of a JXTA system. Messages and requests are routed through Rendezvous peers and a localized network failure may occur should Rendezvous peers fail. Also, the use of XML message introduces large overhead into the JXTA messaging architecture.

This project then proposes extensions to the current JADE and JXTA. The JADE extensions and their descriptions are shown in Figure 51.

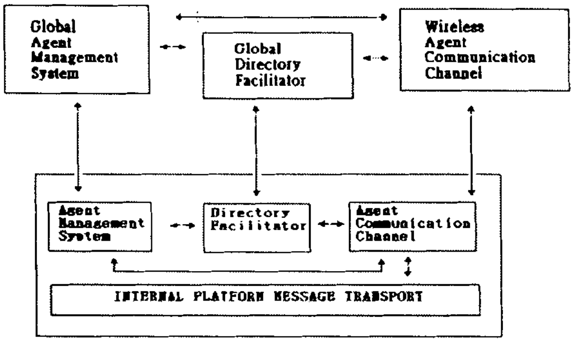


FIGURE 51. EXTENSIONS OF JADE AGENT MODEL

- The *Broadcast Agent* models the GDF and handles broadcasted messages to/from other nodes. It is responsible for maintaining a current list of all nodes currently available on the network.

- The *Sender Agent* models the GAMS and provides management service for the respective node. It is also responsible for the sending of messages.

- The *Receiver Agent* models the WACC and receives messages from other nodes. It internally determines the subsequent nodes that the message should traverse.

The JXTA extensions and their descriptions are shown in Figure 52.

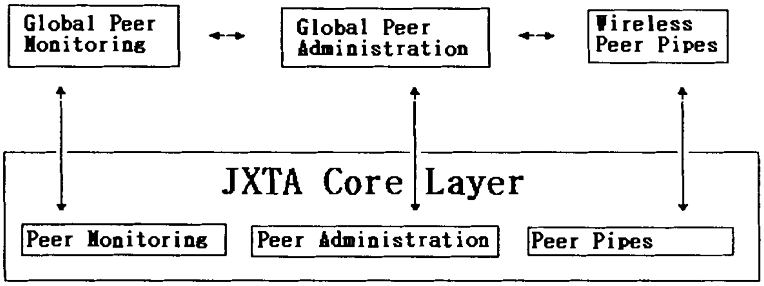


FIGURE 52. MODIFIED JXTA FRAMEWORK FOR AN IMPROVED DS

- The *Global Peer* Monitoring maintains a current list of all nodes currently available on the network. It also handles broadcasted messages to/from other nodes.
- The *Wireless Peer Pipes* extension is used to restrict the sending of message to only nodes available according to the user-defined scenario.
- The *Global Peer Administration* extension is used to handle the added administrative overhead. It also initializes and supervises the JXTA node. An entry point is contained in the *GPA* to allow developer to operate the JXTA node.

The extensions are accomplished by implementing four distinct Java Classes. The *PipeListener()* and *PipeSender()* classes are used in conjunction to model the *Wireless Peer Pipe.* The *PipeComm()* and the *PeerRoute()* classes are used to model the *Global Peer Administration* and *Global Peer Monitoring* respectively.

When JXTA and JADE are compared quantitatively, we found that JADE seems to be a better distributed software that is distributed in terms of performance and scalability. In all three test scenarios, the performance of JADE is significantly better than that of JXTA. Not only is JADE more efficient under varying message load, but it is also more efficient when the receiving agents reside both on the same and on different hosts.

The main reason for the apparent superiority of JADE over JXTA is the extensive use of the centralized management system by JADE. Agents are able to locate receiver agents by querying the AMS of the *main container*. However, this characteristic is not consistent with a standard distributed system: The system should not be critically dependent on any specific node.

JXTA on the other hand, does not use a centralized management system and relies heavily on Rendezvous peers scattered throughout the network to discover and route messages and requests. Although longer latency for message exchanges when compared with JADE, a JXTA system is not critically dependent on any node.

5.3.2 Concluding Remarks

Although JADE and JXTA are built with a common purpose, both have limitations in their present form. Extensions are needed to both platforms to achieve improved implementations of distributed systems.

Overall, we found that JADE outperformed JXTA both in terms of latency and scalability, mainly due to its partially centralized approach. JADE is also easier to understand and to deploy than JXTA. Numerous configurations and options are available in JXTA to customize a unique distributed system, thus creating a daunting task for beginners.

Agents in JADE are able to freely migrate among the different containers and hosts, while agents in JXTA are physically tied to the hardware that they reside on. This is an important feature that JXTA is lacking and would increase the robustness and scalability of a JXTA system.

We feel that both JADE and JXTA requires extensions to their existing architectures for better distributed systems. This project outlined and implemented the extensions needed for the improvements.

5.3.3 Future Research

Distributed networks represent a new and emerging technology. Although they appear to alleviate networking constraints that result from a centralized topology, further research is needed to deploy mature, robust and highly scalable distributed networks.

In this research, two distributed software agent platforms were analyzed and extensions were outlined and implemented. Future validation of the results requires implementation in a real-world environment where hundreds or perhaps thousands of nodes are communicating using wireline and wireless in real time. A variety of system hardware can be used as nodes in this real-world environment. We must also experiment with different intelligent routing algorithms to maximize efficiency and minimize latency. Network bottlenecks that result from the exponential growth of administrative overhead must be analyzed and tests can be performed to evaluate the robustness of the network. Gateways should also be developed to resolve interoperability between different software platforms.

Although this thesis compared two distributed software agent platforms, other products should be evaluated to ascertain their relative similarities and differences and compare them for specific applications. Their relative performances in a distributed network should also be quantitatively and qualitatively analyzed.

6 REFERENCES

[1] C. Ng, D. Sabaz, and W.A. Gruver, "Distributed algorithm simulator for wireless peer-to-peer networks," *Proc. of the IEEE International Conference on Systems, Man, and Cybernetics*, The Hague, Netherlands, 2004.

[2] E. Chen, D. Sabaz, and W.A. Gruver, "JADE and wireless distributed environments," *Proc. of the IEEE International Conference on Systems, Man, and Cybernetics*, The Hague, Netherlands, 2004.

[3] JADE, Java Agent Development Framework, http://jade.cselt.it

[4] JXTA, http://www.jxta.org/ accessed April 8, 2005

[5] Foundation for Intelligent Physical Agents (FIPA), http://www.fipa.org accessed March 2, 2005

[6] FIPA-OS, http://www.nortelnetworks.com/ accessed April 8, 2005

[7] Agent Oriented Software Group, http://www.agentsoftware.com

[8] E. Cortese, F. Ouarta, and G. Vitaglione, "Scalability and performance of the JADE message transport system," *Proc. of the AAMAS Workshop on AgentCities*, Bologna, Italy, July 2002

[9] Digital Equipment Corporation, "In Memoriam: J.C.R. Licklider 1915-1990," SRC Research Report 61, August 1990.

[10] L. Roberts, T. Merril "Toward a cooperative network of time-shared computers," *Proc. of the Fall AFIPS Conference*, Oct. 1966.

[11] V. Cerf and R. Kahn, "A protocol for packet network interconnection," *IEEE Trans. on Communications Technology*, Vol. COM-22, Number 5, May 1974 , pp. 627-641.

[12] F. Bellifemine, G. Caire, A. Poggi, G. Rimassa, "JADE – A white paper," *EXP – In Search Of Innovation*, Volume 3, Number 3, Telecom Italia Labs, Turin, Italy, 2003.

[13] J. F. Kurose and K. W. Ross, *Computer Networking*, AW Education Group, USA, 2002.

[14] S.I. Kumaran, *JINI Technology, An Overview*, Upper Saddle River, NJ, USA, 2002

[15] S. Li, *JXTA Peer-to-Peer Computing with Java*, Birmingham, UK, 2001

[16] M. Laukkanen, *Evaluation of FIPA-Compliant Agent Platforms,* Master's Thesis, Department of Information Technology, Lappeenranta University of Technology, Finland, 2002.

[17] F. Bellifemine, G.Caire, T. Trucco, G. Rimassa, *JADE's Programmer's Guide*, Telecom Italia Labs, Turin, Italy, 2003

[18] B. Wilson, *Projects: JXTA Book*, New Rider's Publishing Co., USA, 2003

[19] K. Burbeck, D. Garpe, and S. Nadjm-Tehrani, "Scale-up and performance studies of three agent platforms," *Proc. of International Performance, Communication and Computing Conference, Middleware Performance Workshop.*, Phoenix, AZ, USA, pp. 857-863, Apr. 2004

[20] The Developer, http://www.developer.com accessed March 23, 2005

[21] FIPA 97 Specification, Spec 2, "Agent Communication Language," *Introduction to Sequencing and Scheduling*, Durham, USA, 1974.

[22] GRASSHOPPER, http://www.fokus.gmd.de/ accessed October 25, 2004

[23] ZEUS, http://www.labs.bt.com/projects/agents/zeus accessed September 3, 2004

[24] Agent Development Kit, http://www.madkit.org/ accessed January 2, 2005

[25] Garpe, D., *Comparison of Three Agent Platforms – Performance, Scalability and Security,* Master's Thesis, LiTH-IDA-EX-03/070-SE, Department of Computer and Information Science, Linkoping University, Sweden, 2003.

[26] D. Sabaz, W. A. Gruver, and M. H. Smith, "Distributed systems with agents and holons," *Proc. of the 2004 IEEE International Conference on Systems, Man, and Cybernetics*, The Hague, Netherlands, October 2004.

APPENDIX A

This Appendix contains sample code listing for the three JADE agents,

- Sender Agent
- Receiver Agent
- Broadcast Agent

```java
// ----------------------------------------------------------------------------
// INCLUDED JADE FILES
// ----------------------------------------------------------------------------
package examples.receivers;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;

import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.DFService;
import jade.domain.FIPAException;

// ----------------------------------------------------------------------------
// INCLUDED JAVA FILES
// ----------------------------------------------------------------------------
import java.net.*;
import java.util.*;
import java.io.*;
import java.lang.Thread;
import java.lang.*;

// ----------------------------------------------------------------------------------------


public class AgentSender extends Agent {

  protected void setup() {

    // ----------------------------------------------------------------------------
    // Registration with the DF
    DFAgentDescription dfd = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    sd.setType("AgentSender");
    sd.setName(getName());
    sd.setOwnership("Edward");
    dfd.setName(getAID());
    dfd.addServices(sd);
    try {
        DFService.register(this,dfd);
    } catch (FIPAException e) {
        System.err.println(getLocalName()+" registration with DF unsucceeded. Reason: "+e.getM
        doDelete();
    }
    // ----------------------------------------------------------------------------
     String agent_name = this.getName();
     addBehaviour(new SimpleBehaviour(this) {
     private boolean finished = false;


//************************************************************************************
// Main Execution of the program
//************************************************************************************
    public void action()
    {
        try{
            // MAIN USER INTERFACE GUI
            console();
            }catch(Exception e){
        System.out.println(e);
        }
    } // end action

    public boolean done(){
                return finished;
            } // end done
    }); // end addbehavior
    } // end setup

    //************************************************************************************
```

```java
    // Function Body
    //****************************************************************************************
    public void console() throws IOException
    {
        boolean exitConsole = false;
        char userInput;

        // Class used to display information onto screen
        Display display = new Display();

        // Class used to send messages using different routing methods
        Route route = new Route();

        // Class contains functions of a J_Node
        J_Node node = new J_Node();

        // System initialization function
        node.initialize();

        while(!exitConsole)
        {
            userInput = node.main_menu();

            switch (userInput)
            {
                // ----------------------------------------------------------------------------
                case 'a':
                    display.host_info();
                break;
                // ----------------------------------------------------------------------------

                case 'b':
                    display.neighbour_nodes();
                break;
                // ----------------------------------------------------------------------------

                case 'c':
                    display.all_nodes();
                break;

                // ----------------------------------------------------------------------------

                case 'd':

                    char choice = route.route_menu();
                    switch(choice)
                    {
                        // -----------------------------------------
                        // Send the message directly to destination
                        // -----------------------------------------
                        case 'a':
                            route.direct();
                            break;

                        // -----------------------------------------
                        // Send message according to specific Hops
                        // -----------------------------------------
                        case 'b':
                            route.multi_hop();
                            break;
                        // -----------------------------------------
                        // Specify a path to destination
                        // -----------------------------------------
                        case 'c':
                            route.specific_path();
                            break;

                        default:
                        System.out.println("Invalid choice!!");

                    } // end switch
```

```

                break;

                // ---------------------------------------------------------------------------------

                case 'e':
                    node.establish_connection();
                break;

                // ---------------------------------------------------------------------------------

                case 'f':
                    node.direct_send();
                break;

                // ---------------------------------------------------------------------------------
                case 'g':
                    node.remote_setup();
                    break;

                // ---------------------------------------------------------------------------------
                case 'h':
                    node.broadcast();
                break;
                // ---------------------------------------------------------------------------------
                case 'i':
                    node.update_hop_list();
                break;

                // ---------------------------------------------------------------------------------
                case 'j':
                    node.hop_test();
                break;


                // ---------------------------------------------------------------------------------
                case 'x':

                    System.out.println("Exiting ...... Good-Bye!!");
                    exitConsole = true;
                    System.exit(1);
                break;
                // ---------------------------------------------------------------------------------
                default:
                System.out.println("Invalid Entry!! Try again");
            } // end switch

        } // end while

    } // end function console()


private class J_Node{

    J_Node(){ //Begin Constructor

    // initialize all global variable in this class

   } //End Constructor

    // -------------------------------------------------------------------------------------
    // This function initializes the JADE node, delete previous version of files, if any
    // -------------------------------------------------------------------------------------
    public void initialize() throws IOException
    {
        // --------------------------------------------------
        // Delete previous version of neighbour, global and hop list.
        // --------------------------------------------------
        File myFile = new File( "C:\\jade\\bin\\jade\\neighbour_list.txt" );
        myFile.delete();

```

```java
        File myFile2 = new File( "C:\\jade\\bin\\jade\\global_list.txt" );
        myFile2.delete();

        File myFile3 = new File( "C:\\jade\\bin\\jade\\hop_list.txt" );
        myFile3.delete();

        File myFile4 = new File( "C:\\jade\\bin\\jade\\temp_hop_list.txt" );
        myFile4.delete();

        //-----------------------------
        // Initialize global, neighbour, and hop list
        //-----------------------------
        String host = get_own_Inet().toString();
        BufferedWriter bufWriter = new BufferedWriter(new FileWriter("global_list.txt", true))

        // make everything lower case, just to be safe
        host = host.toLowerCase();

        bufWriter.write(host);
        bufWriter.newLine();
        bufWriter.close();

        bufWriter = new BufferedWriter(new FileWriter("neighbour_list.txt", true));
        bufWriter.write(host);
        bufWriter.newLine();
        bufWriter.close();

        bufWriter = new BufferedWriter(new FileWriter("hop_list.txt", true));
        int seperator = host.indexOf("/");
        host = host.substring(0, seperator);
        host = host.concat("#5");
        bufWriter.write(host);
        bufWriter.newLine();
        bufWriter.close();

        bufWriter = new BufferedWriter(new FileWriter("temp_hop_list.txt", true));
        bufWriter.write(host);
        bufWriter.newLine();
        bufWriter.close();
    } // end initialize()


    // -------------------------------------------------------------------------------
    // This is the main menu of a J_Node
    // -------------------------------------------------------------------------------
    public char main_menu() throws IOException
    {

            char userInput;

            System.out.println(" ");
            System.out.println(" ");
            System.out.println(" ");
            System.out.println(" ");
            System.out.println("             Welcome to J-Net\n");
            System.out.println("An Innovative Approach to Distributed Communication. ");
            System.out.println(" ");
            System.out.println("Please select one of the following options: ");
            System.out.println(" ");
            System.out.println("   a) Display Host Computer Name and IP Address");
            System.out.println("   b) Display 1st-tier Nodes Connected to Host");
            System.out.println("   c) Display ALL Nodes within J-Net");
            System.out.println("   d) Send Message to Specific Node");
            System.out.println("   e) Establish a Link Function with a specific Node");
            System.out.println("   f) Adminstrator send (Direct Send)");
            System.out.println("   g) Setup connection for other nodes ");
            System.out.println("   h) Broadcast existence to everyone ");
            System.out.println("   i) Update global hop list ");
            System.out.println("   j) Perform hop test!!! ");
            System.out.println("   x) Exit");

```

```
              System.out.println(" ");
              System.out.println(" ");
              System.out.println(" ");
              System.out.print("Please make your selection: ");

              try{
                    userInput = get_char();
                    return userInput;
              }
              catch(Exception e){
                   System.out.println(e);
              }

              // dummy return
              return 'x';
    } // end main_menu

    // ---------------------------------------------------------------------------------
    // This function is used to establish virtual connection with another J_Node
    // ---------------------------------------------------------------------------------
    // This function writes the Node into neighbour_list.txt
    public void establish_connection() throws IOException
    {
              System.out.println("Enter name of node: ");

              String node_name = getstring();

              InetAddress IP_address = InetAddress.getByName(node_name);

              String to_file = IP_address.toString();
              // make everything lower case, just to be safe
              to_file = to_file.toLowerCase();

              // check if content already exist
              if(!content_exist("neighbour_list.txt", to_file))
              {
                   // Open the neighbour_list.txt file to write to
                   BufferedWriter bufWriter = new BufferedWriter(new FileWriter("neighbour_list.t
                   // write to file
                   bufWriter.write(to_file);
                   bufWriter.newLine();
                   bufWriter.close();
                   System.out.println("New node: " + node_name + " is written to neighbour_list.t
              }
              else
              {
                   System.out.println("Node: " + node_name + " already a neighbour");
              }

              // Send Admin_Setup: message to this new neighbour node so both on neighbour_list

              InetAddress ownAddress = get_own_Inet();
              String host_name = ownAddress.getHostName();

              String message = "Admin_Setup: ".concat(host_name);
              String total_message = node_name.concat("*".concat(message));

              int seperator = total_message.indexOf("*");

              String to_node = total_message.substring(0,seperator);
              String to_message = total_message.substring(seperator+1,total_message.length());

              send_msg(to_node, to_message);
    } // end function


    // ---------------------------------------------------------------------------------
    // This function is used to send message DIRECTLY to another J_Node
    // ---------------------------------------------------------------------------------
    // This function writes the Node into neighbour_list.txt
    public void direct_send() throws IOException
```

```
    {
        System.out.println("Adminstrator Direct Send");
        System.out.println("Enter Node name: ");
        String node_name = getstring();

        System.out.println("Enter message: ");
        String message = getstring();

        // actually send the message
        send_msg(node_name, message);
    }


    // ---------------------------------------------------------------------------
    // This function is used to remotely establish virtual connection with two J_Nodes
    // ---------------------------------------------------------------------------
    // This function writes the Node into neighbour_list.txt
    public void remote_setup() throws IOException
    {
        System.out.println("Setting up connection for another node");
        System.out.println("Enter 1st Node name: ");
        String first_node = getstring();

        System.out.println("Enter 2nd Node name: ");
        String second_node = getstring();

        String message = "Admin_Setup: ".concat(first_node);
        send_msg(second_node, message);

        message = "Admin_Setup: ".concat(second_node);
        send_msg(first_node, message);
    }

    // ---------------------------------------------------------------------------
    // This function is broadcasts existence to every J_Node on network
    // ---------------------------------------------------------------------------
    public void broadcast() throws IOException
    {
        System.out.println("Broadcasting to J-Net");

        int MULTICAST_PORT =  7777;
        String MULTICAST_ADDR = "230.0.0.1";

        try
        {
            // get own Host Information
            //String host = InetAddress.getLocalHost().getHostName();
            String host = get_own_Inet().toString();
            byte[] temp = host.getBytes();

          InetAddress inetAddress = InetAddress.getByName(MULTICAST_ADDR);
          DatagramPacket Out_Packet = new DatagramPacket(temp, temp.length, inetAddress, MULTI
          MulticastSocket multicastSocket = new MulticastSocket();
          multicastSocket.send(Out_Packet);
        }
        catch (Exception exception)
        {
          exception.printStackTrace();
        }
    } // end broadcast()

    // ---------------------------------------------------------------------------
    // This function is used to Update global hop list
    // ---------------------------------------------------------------------------
    public void update_hop_list() throws IOException
    {
        //sender_node#HOP_COUNT#PREVIOUS_sender_node#Original_hop_Count
        System.out.println("Updating global hop list.... Please wait");

        // update from 2 hops to 5 hops.... TO BE CHANGED!!!!!!!!!!!!!!!!!
        String update_hop_header = "Update_Hop_Message_Header: ";
```

```
        String host_name = get_own_Inet().toString();
            int index = host_name.indexOf("/");
            host_name = host_name.substring(0,index);
        String hop_count;
        String neighbour_name;
        String current_line;
        String update_hop_message;

        for(int i=2; i<6; i++)
        {
            BufferedReader bufReader = new BufferedReader(new FileReader("neighbour_list.txt")

            hop_count = String.valueOf(i);
            update_hop_message = host_name.concat("#".concat(hop_count.concat("#")));
            update_hop_message = update_hop_header.concat(update_hop_message);

            // actually send the message to everyone on neighbor list, except itself
            while( (current_line = bufReader.readLine()) != null)
            {
                index = current_line.indexOf("/");
                neighbour_name = current_line.substring(0,index);

                // Don't send message to itself, to add the NOT "!"
                if((neighbour_name.equalsIgnoreCase(host_name)))
                {
                    update_hop_message = update_hop_message.concat(neighbour_name);
                    update_hop_message = update_hop_message.concat("#".concat(hop_count));
                    send_msg(neighbour_name, update_hop_message);
                    System.out.println(update_hop_message);
                }
            } // end while
            bufReader.close();
        }
    } // end update_hop_list


    // -----------------------------------------------------------------------------
    // This function is used to perform hop test
    // -----------------------------------------------------------------------------
    public void hop_test() throws IOException
    {

        String max_hop;

        System.out.println(" ");
        System.out.println(" ");
        System.out.println("Enter maximum hops to test");
        System.out.println(" ");

        // get input from user
        max_hop = getstring();

        // organize the hop_test message
        // Hop_Test_Header:  sent_time#original_sender#max_hop
        String Hop_Test_Header = "Hop_Test_Header: ";

        // Retrieve the number of milliseconds since 1/1/1970 GMT
        Date date = new Date();
        long start_milliseconds = date.getTime();
        // convert to string
        String start_time = String.valueOf(start_milliseconds);

        // get host information
        InetAddress ownAddress = get_own_Inet();
        String host_name = ownAddress.getHostName();

        // organize the hop_test message
        // Hop_Test_Header:  sent_time#max_hop#original_sender

        String hop_test_msg = Hop_Test_Header.concat(start_time);
        hop_test_msg = hop_test_msg.concat("#");
```

```
        hop_test_msg = hop_test_msg.concat(host_name);
        hop_test_msg = hop_test_msg.concat("#");
        hop_test_msg = hop_test_msg.concat(max_hop);

        // get number of 1st-tier neighbours
        int neighbours = number_of_neighbours();
        //int globals = number_of_globals();

        // randomly send out to a first-tier neighbour
        Random x = new Random();  // default seed is time in milliseconds
        //Random x = new Random(long seed); // for reproducible testing

       int  random = x.nextInt(neighbours); // returns random int >= 0 and < n

        // get destination node information
        BufferedReader bufReader = new BufferedReader(new FileReader("neighbour_list.txt"));

        // go to the line in the file
        for(int i=0; i<random; i++)
        {
            bufReader.readLine();
        }

        String current_line = bufReader.readLine();
        int index = current_line.indexOf("/");
        String dest_node = current_line.substring(0,index);
        bufReader.close();

        // send the message out
        send_msg(dest_node, hop_test_msg);

    } // end hop_test()

    // --------------------------------------------------------------------------------
    // This function returns the number of global nodes
    // --------------------------------------------------------------------------------
    private int number_of_globals() throws IOException
    {
        int count = 0;
        BufferedReader bufReader = new BufferedReader(new FileReader("global_list.txt"));

        while(bufReader.readLine() != null)
        {
            count++;
        }
        bufReader.close();

        return count;
    }   // end function
    // --------------------------------------------------------------------------------
    // This function returns the number of neighbours this node is connected to
    // --------------------------------------------------------------------------------
    private int number_of_neighbours() throws IOException
    {
        int count = 0;
        BufferedReader bufReader = new BufferedReader(new FileReader("neighbour_list.txt"));

        while(bufReader.readLine() != null)
        {
            count++;
        }
        bufReader.close();

        return count;
    }   // end function
    // --------------------------------------------------------------------------------
    // This is the JADE program that actually sends the message OUT
    // --------------------------------------------------------------------------------
    public void send_msg(String node_name, String message)
    {
        String responder = null;
```

```
        String dest = null;

        try     //trying to open socket for data going out
        {
            dest = "http://".concat(node_name).concat(":7778/acc");
            // Use String class manipulation to get responder address
            int end_index = dest.lastIndexOf(":");
            responder = "receiver@".concat(node_name).concat(":1099/JADE");

            // Setup JADE send variables to use JADE to send the message out
            AID r = new AID();

            r.setName(responder);
            r.addAddresses(dest);

            // create the ACL message and set specs, then send the msg according to
            // the user defined address

            ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
            msg.setSender(getAID());
            msg.addReceiver(r);

            msg.setContent(message);
            send(msg);

//              finished = false;
        }
        catch(Exception e)
        {
            System.out.println("JADE send failed");
        }
}

    // ------------------------------------------------------------------------------
    // This function returns the InetAddress of the current host computer
    // ------------------------------------------------------------------------------
    public InetAddress get_own_Inet(){

        try                                                    //trying to set own ip-address
        {
            InetAddress ownIP = InetAddress.getLocalHost();
            return ownIP;
        }
        catch(UnknownHostException e)
        {
            System.out.println(e);
        }
        return null;
    }// end get_own_Inet()

    // ---------------------------------------------------------------------------------------
    // This functions returns the character input from the user
    private char get_char() throws IOException
    {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String s = br.readLine();
        return s.charAt(0);

    } // end get_char()


    // ---------------------------------------------------------------------------------------
    // This function returns the entire line of String
    private String getstring() throws IOException
    {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String s = br.readLine();
        return s;

```

```
    } // end getString()

    // --------------------------------------------------------------------------
    // This functions checks if incoming content already exist in file

    private boolean content_exist(String filename, String content) throws IOException
    {
        boolean exist = false;
        String current_line;

        BufferedReader bufReader = new BufferedReader(new FileReader(filename));

        while( (current_line = bufReader.readLine()) != null)
        {
            if(current_line.equalsIgnoreCase(content))
            {
                exist = true;
                bufReader.close();
                return exist;
            }
        }
        return exist;
    }   // end function content_exist()

} // end class J_Node
// ----------------------------------------------------------------------------------



/*****************************************************************************************
This class defines the various routing algorithms to be used to route the packet to destinatio
*****************************************************************************************

private class Route{

    Route(){ //Begin Constructor

    // initialize all global variable in this class

   } //End Constructor


    // --------------------------------------------------------------------------
    // This function sends message through JADE directly to destination
    // --------------------------------------------------------------------------
    public char route_menu() throws IOException
    {
        char userInput;

        System.out.println(" ");
        System.out.println(" ");
        System.out.println("Choose how you like to send the message");
        System.out.println(" ");
        System.out.println("    a) Send Directly to Destination");
        System.out.println("    b) Specify maximum number of HOPS allowed");
        System.out.println("    c) Specify a specific path to Destination");
        System.out.println(" ");
        System.out.println(" ");
        System.out.print("Please make your selection: ");

        try{
             userInput = get_char();
             return userInput;
        }
        catch(Exception e){
            System.out.println(e);
        }

        // dummy return
        return 'x';
    } // end route_menu
```

```java

     // ---------------------------------------------------------------------------
     // This function sends message through JADE directly to destination
     // ---------------------------------------------------------------------------
     public void direct() throws IOException
     {

         System.out.println("Enter Node name: ");
         String node_name = getstring();

         System.out.println("Enter message: ");
         String message = getstring();
         // actually send the message
         send_msg(node_name, message);
     }

     // ---------------------------------------------------------------------------
     // This function sends message to a node up to user-defined MAX HOPS
     // ---------------------------------------------------------------------------
    public void multi_hop() throws IOException
    {
        String Multi_Hop_Header = "Multi_Hop_Message_Header: ";
        String Multi_Hop_Message;
        String message;
        String dest_node;
        String MAX_HOP;
         String node_name;

        System.out.println("Enter destination");
         dest_node = getstring();

        System.out.println("Enter message");
         message = getstring();

         System.out.println("Enter Maximum number of hops allowed");
         MAX_HOP = getstring();

         // Multi_Hop_Message_Header: 3#destination$msg_body
         Multi_Hop_Message = Multi_Hop_Header.concat(MAX_HOP.concat("#".concat(dest_node.concat

         System.out.println("multi_hop_message: " + Multi_Hop_Message);

         String current_line;

         // get own host name
         InetAddress ownAddress = get_own_Inet();
         String host_name = ownAddress.getHostName();

         // check if destination is already a neighbour_node
         int front = Multi_Hop_Message.indexOf("#");
         int back = Multi_Hop_Message.indexOf("$");

         // if already in neighbour list
         if(content_exist("neighbour_list.txt", Multi_Hop_Message.substring(front+1, back)))
         {
             // Extract the message
             message = Multi_Hop_Message.substring(back+1, Multi_Hop_Message.length());
             send_msg(Multi_Hop_Message.substring(front+1, back), message);
         }
         // send to everyone on neighbour list
         else
         {
             // actually send the message to everyone on neighbor list
             BufferedReader bufReader = new BufferedReader(new FileReader("neighbour_list.txt")

             while( (current_line = bufReader.readLine()) != null)
             {
                 int index = current_line.indexOf("/");
                 node_name = current_line.substring(0,index);

                 // Don't send message to itself
```

```
                if(!(node_name.equalsIgnoreCase(host_name)))
                {
                    send_msg(node_name, Multi_Hop_Message);
                }
            }
        } // end else

    } // end route_multi_hop()

    // ----------------------------------------------------------------------------------------
    public void specific_path() throws IOException
    {
        String Specific_Path_Header = "Specific_Path_Message_Header:";
        String message;
        String next_node;
        String temp_header = " ";
        String Specific_Path_Message;
        char another;
        int back;

        boolean next = true;

        System.out.println("Enter message");
        message = getstring();

        //  Specific_Path_Message_Header: #next_destination#next_next_destination$msg_body
        while(next)
        {
            System.out.println("Enter Next Node for routing");
            next_node = getstring();
            temp_header = temp_header.concat("#".concat(next_node));

                System.out.println("Attach another Hop??   Y or N");
                another = get_char();

                if(another == 'N' || another == 'n')
                {
                    next = false;
                }
        } // end while

        Specific_Path_Message = Specific_Path_Header.concat(temp_header.concat("$".concat(mess

        //System.out.println("Specific_Path_Message " + Specific_Path_Message);

        // Extract out the 1st hop as destination for this send
        int last = Specific_Path_Message.lastIndexOf("#");
        int front = Specific_Path_Message.indexOf("#");

        if(last == front)
        {
            back = Specific_Path_Message.indexOf("$", front+1);
        }
        else
        {
            back = Specific_Path_Message.indexOf("#", front+1);
        }

        String node_name = Specific_Path_Message.substring(front+1, back);

        // if 1st path is already a neighbour node
        if(content_exist("neighbour_list.txt", node_name))
        {
            int msg_start = Specific_Path_Message.indexOf("$");
            Specific_Path_Message = Specific_Path_Message.substring(msg_start+1, Specific_Path_
        }
        // actually send the message
        send_msg(node_name, Specific_Path_Message);
    } // end route_specific_path()

    // ---------------------------------------------------------------------------------
```

```
    // This functions checks if incoming content already exist in file
    // ---------------------------------------------------------------------------------
    private boolean content_exist(String filename, String content) throws IOException
    {
        boolean exist = false;
        String current_line;

        BufferedReader bufReader = new BufferedReader(new FileReader(filename));

        while( (current_line = bufReader.readLine()) != null)
        {
            if(current_line.startsWith(content))
            {
                exist = true;
                bufReader.close();
                return exist;
            }
        }
        return exist;
    }
    // ------------------------------------------------------------------------------
    // This function returns the InetAddress of the current host computer
    // ------------------------------------------------------------------------------
    public InetAddress get_own_Inet(){

        try                                                        //trying to set own ip-address
        {
            InetAddress ownIP = InetAddress.getLocalHost();
            return ownIP;
        }
        catch(UnknownHostException e)
        {
            System.out.println(e);
        }
        return null;
    }// end get_own_Inet()
// ----------------------------------------------------------------------------------
// This is the JADE program that actually sends the message OUT
// ----------------------------------------------------------------------------------
    public void send_msg(String node_name, String message)
    {
        String responder = null;
        String dest = null;

                try     //trying to open socket for data going out
                {

                    dest = "http://".concat(node_name).concat(":7778/acc");
                    // Use String class manipulation to get responder address
                    int end_index = dest.lastIndexOf(":");
                    responder = "receiver@".concat(node_name).concat(":1099/JADE");

            //      System.out.println("responder: " + responder);
            //      System.out.println("dest: " + dest);
            //      System.out.println("message " + message);

                    // Setup JADE send variables to use JADE to send the message out
                    AID r = new AID();

                    r.setName(responder);
                    r.addAddresses(dest);

                    // create the ACL message and set specs, then send the msg according to
                    // the user defined address

                    ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
                    msg.setSender(getAID());
                    msg.addReceiver(r);

                    msg.setContent(message);
                    send(msg);
```

```
//              finished = false;
                }
                catch(Exception e)
                {
                    System.out.println("JADE send failed");
                }

    } // end function send_msg

    // -----------------------------------------------------------------------------------------
    // This function returns the entire line of String
    private String getstring() throws IOException
    {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String s = br.readLine();
        return s;

    } // end getString()

    // -----------------------------------------------------------------------------------------
    // This functions returns the character input from the user
    private char get_char() throws IOException
    {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String s = br.readLine();
        return s.charAt(0);

    } // end get_char()

    } // end class Route
} // end class AgentSender









/*************************************************************************************
This class defines the various display functions to output information to screen
**************************************************************************************/

class Display{

Display(){ //Begin Constructor

    // initialize all global variable in this class
} //End Constructor


    // ---------------------------------------------------------------------------------
    // This function returns the InetAddress of the current host computer
    // ---------------------------------------------------------------------------------
    public InetAddress get_own_Inet(){

        try                                                     //trying to set own ip-address
        {
            InetAddress ownIP = InetAddress.getLocalHost();
            return ownIP;
        }
        catch(UnknownHostException e)
        {
            System.out.println(e);
        }
        return null;
    }// end get_own_Inet()
```

```java

    // ---------------------------------------------------------------------------
    // This function displays host information
    // ---------------------------------------------------------------------------
    public void host_info(){

        InetAddress ownAddress = get_own_Inet();
        String host_name = ownAddress.getHostName();
        String host_IP = ownAddress.getHostAddress();

        System.out.println(" ");
        System.out.println("Display Host Information");
        System.out.println(" ");
        System.out.println("Host Computer is: " + host_name);
        System.out.println("Host IP is: " + host_IP);
    }

    // ---------------------------------------------------------------------------
    // This function displays all neighbour node information
    // ---------------------------------------------------------------------------
    public void neighbour_nodes() throws IOException
    {
        try{
            BufferedReader bufReader = new BufferedReader(new FileReader("neighbour_list.txt")
            String current_line;

            System.out.println("Displaying all 1st neighbour nodes");
            System.out.println(" ");

            while( (current_line = bufReader.readLine()) != null)
            {
                int index = current_line.indexOf("/");

                System.out.println("Name: " + current_line.substring(0,index));
                System.out.println("IP: " + current_line.substring(index+1,current_line.length
                System.out.println(" ");
            }
        }catch(Exception e){
            System.out.println("No nodes are currently connected");
        }

    } // end display_neighbour_nodes()

    // ---------------------------------------------------------------------------
    // This function displays information on all nodes
    // ---------------------------------------------------------------------------
    // This function display ALL nodes currently available on J-Net
    public void all_nodes() throws IOException
    {
        System.out.println("Displaying ALL Nodes within J-Net");
        System.out.println(" ");

        try{
            String current_line;
            int count=1;
            BufferedReader bufReader = new BufferedReader(new FileReader("global_list.txt"));

            while( (current_line = bufReader.readLine()) != null)
            {
                int index = current_line.indexOf("/");

                System.out.println("Name: " + current_line.substring(0,index));
                System.out.println("IP: " + current_line.substring(index+1,current_line.length
                System.out.println(" ");
                count++;
            }
            System.out.println("Total of " + (count-1) + " nodes are available on J-Net");
            System.out.println(" ");
        }catch(Exception e){
            System.out.println("No nodes on J-Net");
        }
```

```
    }   // end all_nodes()

} // end class Display


```

```
// ------------------------------------------------------------------------------
// INCLUDED JAVA FILES
// ------------------------------------------------------------------------------
package examples.receivers;

import java.util.*;
import java.io.*;
import java.net.*;
import java.lang.*;
import java.lang.Thread;

// ------------------------------------------------------------------------------
// INCLUDED JADE FILES
// ------------------------------------------------------------------------------
import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.DFService;
import jade.domain.FIPAException;

// ------------------------------------------------------------------------------


public class AgentReceiver extends Agent {


    class WaitPingAndReplyBehaviour extends SimpleBehaviour {

    private boolean finished = false;

    public WaitPingAndReplyBehaviour(Agent a) {
        super(a);
    }

    public void action(){

        final String admin_header = "Admin_Setup: ";                            // header used
        final String broadcast_header = "Broadcast_Setup: ";                    // header used
        final String multi_hop_header = "Multi_Hop_Message_Header: ";           // header used
        final String specific_path_header = "Specific_Path_Message_Header: ";   // header used
        final String update_hop_header = "Update_Hop_Message_Header: ";         // header used
        final String update_hop_list_header = "Update_Hop_List_Header: ";       // header to wr
        final String Hop_Test_Header = "Hop_Test_Header: ";                     // header used
        final String End_Hop_Test_Header = "End_Hop_Test_Header: ";             // header used

        // wait here until a msg is received, since this is a one-behaviour function.
        ACLMessage  msg = myAgent.receive();//blockingReceive();

        if(msg != null)
        {
            try
            {
                // retrieve the message
                String content = msg.getContent();

                // class to handle incoming messages
                Receive receive = new Receive();

                // *******************************************************************
                // RESPONDING TO REMOTE REQUEST CONNECTION BY ADMINISTRATOR
                // *******************************************************************
                // test if message is adminstrative message
                // store into neighbour_list if not already exists
                if(content.startsWith(admin_header))
                {
                    receive.ADMIN_HEADER(content);
                }   // end if(admin_header)

                // *******************************************************************
```

```
                // REPLYING TO BROADCASTING MESSAGES
                // *******************************************************************
                else if(content.startsWith(broadcast_header))
                {
                    receive.BROADCAST_HEADER(content);
                }   // end if(broadcast_header)

                // *******************************************************************
                // Relay message to destination (Multi_hop)
                // *******************************************************************
                // test if message is broadcast message header
                // store into global_list
                else if(content.startsWith(multi_hop_header))
                {
                    receive.MULTI_HOP_HEADER(content);
                }

                // *******************************************************************
                // Update_Hop_List message
                // *******************************************************************
                else if(content.startsWith(update_hop_header))
                {
                    receive.UPDATE_HOP_HEADER(content);

                } // end else if

                // *******************************************************************
                // Update hop_List.txt
                // *******************************************************************
                else if(content.startsWith(update_hop_list_header))
                {
                    receive.UPDATE_HOP_LIST(content);
                } // end else if

                // *******************************************************************
                // Hop_Test_Header
                // *******************************************************************
                else if(content.startsWith(Hop_Test_Header))
                {
                    receive.HOP_TEST_HEADER(content);
                }
                // *******************************************************************
                // End_Hop_Test_Header (Get time difference)
                // *******************************************************************
                else if(content.startsWith(End_Hop_Test_Header))
                {
                    receive.END_HOP_TEST_HEADER(content);
                }

                // No header, so must be message received
                else
                {
                    System.out.println("RECEIVED: " + content);
                }

/*
                // *************************************************
                // Send to JAVA program
                int Jade_Java_port = 4801;
                byte[] temp = new byte[1024];
                temp = content.getBytes();  // convert to byte array

                // Actually send the packet out
                DatagramPacket data_out_packet = new DatagramPacket(temp, temp.length, ownIP, J
                DatagramSocket Out_socket = new DatagramSocket();
                Out_socket.send(data_out_packet);
                // **************************************************
*/


            }
            catch(Exception e)
```

```
            {
                System.out.println(e);
            }


        /*
        // create a reply message to the Sender Agent
        ACLMessage reply = msg.createReply();

        // set message type
        reply.setPerformative(ACLMessage.INFORM);
        // set content
        reply.setContent("ACK: Message Received");

        send(reply);*/

        } // end if msg!=null
        else
        {
            block();
        }
    }   // end action

    public boolean done() {
        return finished;
    }

    // ---------------------------------------------------------------------------------
    // This functions checks if incoming content already exist in file
    // ---------------------------------------------------------------------------------
    private boolean content_exist(String filename, String content) throws IOException
    {
        boolean exist = false;
        String current_line;

        BufferedReader bufReader = new BufferedReader(new FileReader(filename));

        while( (current_line = bufReader.readLine()) != null)
        {
            if(current_line.startsWith(content))
            {
                exist = true;
                bufReader.close();
                return exist;
            }
        }
        return exist;
    }

    } //End class WaitPingAndReplyBehaviour


    protected void setup() {

    // Registration with the DF
    DFAgentDescription dfd = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    sd.setType("AgentReceiver");
    sd.setName(getName());
    sd.setOwnership("Edward");
    //sd.addOntologies("PingAgent");
    dfd.setName(getAID());
    dfd.addServices(sd);
    try {
        DFService.register(this,dfd);
    } catch (FIPAException e) {
        System.err.println(getLocalName()+" registration with DF unsucceeded. Reason: "+e.getMe
        doDelete();
    }

    WaitPingAndReplyBehaviour PingBehaviour = new  WaitPingAndReplyBehaviour(this);
```

```
    addBehaviour(PingBehaviour);

    }

private class Receive{

Receive(){ //Begin Constructor

    // initialize all global variable in this class
} //End Constructor

    // ----------------------------------------------------------------------------
    // This function returns the InetAddress of the current host computer
    // ----------------------------------------------------------------------------
    public InetAddress get_own_Inet(){

        try                                                        //trying to set own ip-address
        {
            InetAddress ownIP = InetAddress.getLocalHost();
            return ownIP;
        }
        catch(UnknownHostException e)
        {
            System.out.println(e);
        }
        return null;
    }// end get_own_Inet()

    // ---------------------------------------------------------------------------------------
    // This function process ADMIN_SETUP messages (sets up connection with specificed neighbour
    // ---------------------------------------------------------------------------------------
    public void ADMIN_HEADER(String content) throws IOException
    {

        int index = content.indexOf(":");
        int length = content.length();

        String node_to_add = content.substring(index+2, length);
        InetAddress IP_Addr = InetAddress.getByName(node_to_add);
        String tofile = IP_Addr.toString();
        // make everything lower case, just to be safe
        tofile = tofile.toLowerCase();

        if(!(content_exist("neighbour_list.txt", tofile)))
        {
            // Open the neighbour_list.txt file to write to
            BufferedWriter bufWriter = new BufferedWriter(new FileWriter("neighbour_list.txt",
            // write to file
            bufWriter.write(tofile);
            bufWriter.newLine();
            bufWriter.close();
            System.out.println("Node: " + tofile + " is added remotely by Administrator");
        }

    }

    // ---------------------------------------------------------------------------------------
    // This function process BROADCAST_SETUP messages (handles broadcast messages, writes to gl
    // ---------------------------------------------------------------------------------------
    public void BROADCAST_HEADER(String content) throws IOException
    {

        int index = content.indexOf(":");
        int length = content.length();

        String node_to_add = content.substring(index+2, length);
        InetAddress IP_Addr = InetAddress.getByName(node_to_add);
        String tofile = IP_Addr.toString();
        // make everything lower case, just to be safe
        tofile = tofile.toLowerCase();

```

```
        // check if content already exist
        if(!(content_exist("global_list.txt", tofile)))
        {
            // Open the neighbour_list.txt file to write to
            BufferedWriter bufWriter = new BufferedWriter(new FileWriter("global_list.txt", tru
            // write to file
            bufWriter.write(tofile);
            bufWriter.newLine();
            bufWriter.close();
            System.out.println("Node: " + tofile + " is written to global_list.txt");
        }

    }

    // ----------------------------------------------------------------------------------
    // This function process MULTI_HOP_HEADER messages (user-defined maximum hops)
    // ----------------------------------------------------------------------------------
    public void MULTI_HOP_HEADER(String content) throws IOException
    {

        /*
            Check if dest_node is a neighbour node, if is, send directly
            if not, decrement hop count and send to all neighbour node
            if hop_count==0, discard (send msg failed??)
        */
        // Extract destination node to see if neighbour node

        int front = content.indexOf("#");
        int back = content.indexOf("$");
        int index;

        String dest_node = content.substring(front+1, back);
        String new_content;              // new content of message, sent to all neighbour no
        String multi_hop_header = "Multi_Hop_Message_Header: "; // header used to route packet

        // if already in neighbour list, send directly
        if(content_exist("neighbour_list.txt", dest_node))
        {
            // Multi_Hop_Message_Header: 3#destination$msg_body
            // Extract the message and send to destination
            String msg_node = content.substring(back+1, content.length());
            send_msg(dest_node, msg_node);
        }
        // decrement Hop count and send to all neighbour
        else
        {
            // extract hop count
            // Multi_Hop_Message_header: 3#destination$msg_body
            int start = content.indexOf(":");
            String hop = content.substring(start+2, front);
            int temp_hop = Integer.parseInt(hop);
            temp_hop--;
            hop = String.valueOf(temp_hop);

            // get own host_name
            InetAddress ownAddress = get_own_Inet();
            String host_name = ownAddress.getHostName();
            // make everything lower case, just to be safe
            host_name = host_name.toLowerCase();

            // go through neighbour list and send to all neighbours
            if(temp_hop>0)
            {
                content = content.substring(front, content.length());

                new_content = multi_hop_header.concat(hop.concat(content));
                System.out.println("new_content: " + new_content);

                // actually send the message to everyone on neighbor list
                BufferedReader bufReader = new BufferedReader(new FileReader("neighbour_list.tx
                String current_line;
```

```

            while((current_line = bufReader.readLine()) != null )
            {
                index = current_line.indexOf("/");
                String to_node = current_line.substring(0,index);

                if(!(to_node.equalsIgnoreCase(host_name)))
                {
                    send_msg(to_node, new_content);
                }
            }
                bufReader.close();
            }   // end if(hop!=)
    } // end else
} // end function

// ------------------------------------------------------------------------------------------
// This function process MULTI_HOP messages (decrement hops and send to others)
// ------------------------------------------------------------------------------------------
public void UPDATE_HOP_HEADER(String content) throws IOException
{

        //sender_node#HOP_COUNT#PREVIOUS_sender_node#Original_hop_Count
        final String update_hop_list_header = "Update_Hop_List_Header: "; // header to writ
        int first = content.indexOf("#");
        int second = content.indexOf("#", first+1);
        int end = content.lastIndexOf("#");

        // convert to INT
        int hop_count = Integer.parseInt(content.substring(first+1, second));

        String final_hop_count=null;
        String original_sender;
        String current_line;
        String new_hop_count;

        // get current host name
        String local_host = get_own_Inet().toString();
        int host_index = local_host.indexOf("/");
        local_host = local_host.substring(0,host_index);

        if(hop_count>0)
        {
            hop_count = hop_count-1;
        }

        // hop_count==0!!!!!!!!!!!!!!!!!!!!!!!!!!
        if(hop_count==0)
        {
            // end of hop reached, send back to sender with hop info

            //Update_Hop_List_Header: current_node#original_hop_count
            int space = content.indexOf(" ");
            String update_hop_list = update_hop_list_header.concat(local_host);
            update_hop_list = update_hop_list.concat(content.substring(end, content.length(

            original_sender = content.substring(space+1, first);
            System.out.println("original: " + original_sender);
            System.out.println("update_hop_list: " + update_hop_list);
            // send to original sender TO BE MODIFIED!!!!!!!
            send_msg(original_sender, update_hop_list);
        }
        else
        {
            // Update_Hop_Message_Header: sender_node#HOP_COUNT#PREVIOUS_sender_node#Origin

            //Replace the hop_count and send to everyone on the list, except to itself
            new_hop_count = String.valueOf(hop_count);
            System.out.println("hop_count: "  + hop_count);

            // get message header
```

```
                String temp_content = content.substring(0,first);

                // attach new hop_count
                temp_content = temp_content.concat("#".concat(new_hop_count));

                // get original hop count
                temp_content = temp_content.concat(content.substring(second, content.length()))

                // send to everyone on neighbour list with new hop_count
                BufferedReader bufReader = new BufferedReader(new FileReader("neighbour_list.tx

                // actually send the message to everyone on neighbor list, except itself AND pr
                String previous_sender = content.substring(second+1,end);

                while( (current_line = bufReader.readLine()) != null)
                {
                    host_index = current_line.indexOf("/");
                    String neighbour_name = current_line.substring(0,host_index);

                    // Don't send message to itself
                    if(!(neighbour_name.equalsIgnoreCase(local_host)))// || !(neighbour_name.eq
                    {
                        System.out.println("Resend: " + neighbour_name + " " + temp_content);
                        send_msg(neighbour_name, temp_content);
                    }
                } // end while
                bufReader.close();
            } // end else
    } // end UPDATE_HOP_HEADER

    // ---------------------------------------------------------------------------------------
    // This function process MULTI_HOP messages (decrement hops and send to others)
    // ---------------------------------------------------------------------------------------
    public void UPDATE_HOP_LIST(String content) throws IOException
    {
        System.out.println("rewrite: " + content);

        // array used to hole hop_list count
        String[] hop_list = new String [500];
        //Update_Hop_List_Header: end_node#original_hop_count
        int space = content.indexOf(" ");
        int seperator = content.indexOf("#");

        String end_node = content.substring(space+1, seperator);
        String final_count = content.substring(seperator+1, content.length());

        // loop through hop_list to record hop info, only take the info with least hops!!
        BufferedReader bufReader = new BufferedReader(new FileReader("hop_list.txt"));

        //copy file into hop_list array, then delete file
        String current_line;
        int counter = 1;
        while( (current_line = bufReader.readLine()) != null)
        {
            hop_list[counter] = current_line;
            counter++;
        } // end while

        // delete the file
        bufReader.close();
        File myFile = new File( "C:\\jade\\bin\\jade\\hop_list.txt" );
        myFile.delete();

        counter = 1;
        String node_in_file;
        String old_node_count;
        String replacement;
        String node_name_in_file;
        String received_node;

        int message_count;
```

```
        int array_count;
        boolean node_exist = false;

        int sept;
        while( hop_list[counter] != null)
        {
            node_in_file = hop_list[counter];
            sept = node_in_file.indexOf("#");
            node_name_in_file = node_in_file.substring(0,sept);

            // if node exist
            if( end_node.equalsIgnoreCase(node_name_in_file))
            {
                node_exist = true;
                // get node_count from string array (File)
                array_count = Integer.parseInt(node_in_file.substring(sept+1, node_in_file.leng

                // get node_count from message
                message_count = Integer.parseInt(content.substring(seperator+1, content.length(

                // replace array if hop is now smaller
                if(message_count < array_count)
                {
                    replacement = node_in_file.substring(0,sept+1);
                    replacement = replacement.concat(String.valueOf(message_count));
                    hop_list[counter] = replacement;
                    System.out.println("UPDATED HOP_LIST: " + replacement);
                }
                } // end if

                counter++;
        }   // end while

        // new node, write to file
        if(!node_exist)
        {
            hop_list[counter++] = content.substring(space+1, content.length());
        }

        // open up new hop_list file and write
        BufferedWriter bufWriter = new BufferedWriter(new FileWriter("hop_list.txt", true));

        int i=1;
        while( i<counter )
        {
            bufWriter.write(hop_list[i]);
            bufWriter.newLine();
            i++;
        }
        bufWriter.close();

    } // end UPDATE_HOP_LIST

    // ---------------------------------------------------------------------------------------
    // This function process HOP_TEST_HEADER messages (decrement hops and randomly send to othe
    // ---------------------------------------------------------------------------------------
    public void HOP_TEST_HEADER(String content) throws IOException
    {
        // Hop_Test_Header:  sent_time#original_sender#max_hop
        // extract hop count and decrement and randomly send to peers again
        int last = content.lastIndexOf("#");
        String max_hop = content.substring(last+1, content.length());
        int new_max = (Integer.parseInt(max_hop)) - 1;
        max_hop = String.valueOf(new_max);

        // construct the new message with decremented max_hop
        String temp = content.substring(0,last+1);
        String hop_test_msg = temp.concat(max_hop);

//      System.out.println("hop_test: " + hop_test_msg);

```

```
        // randomly send to neighbours again
        if(new_max > 0)
        {
            // get number of 1st-tier neighbours
            int neighbours = number_of_neighbours();
            //int globals = number_of_globals();

            // randomly send out to a first-tier neighbour
            Random x = new Random();  // default seed is time in milliseconds

            int  random = x.nextInt(neighbours); // returns random int >= 0 and < n

            // get destination node information
            BufferedReader bufReader = new BufferedReader(new FileReader("neighbour_list.txt"))

            // go to the line in the file
            for(int i=0; i<random; i++)
            {
                bufReader.readLine();
            }

            String current_line = bufReader.readLine();
            int index = current_line.indexOf("/");
            String dest_node = current_line.substring(0,index);
            bufReader.close();

            // send the message out
            send_msg(dest_node, hop_test_msg);

        }
        // reached the end, send back to original sender
        else
        {
            int index = content.indexOf("#");
            String End_Hop_Test_Msg = content.substring(0,index);
            End_Hop_Test_Msg = "End_".concat(End_Hop_Test_Msg);

            String dest_node = content.substring(index+1, last);

            send_msg(dest_node, End_Hop_Test_Msg);
            //System.out.println("dest_node: " + dest_node);
            //System.out.println("End_Hop_Test_Msg: " + End_Hop_Test_Msg);
        }
    } // end HOP_TEST_HEADER


    // ---------------------------------------------------------------------------------
    // This function process END_HOP_TEST_HEADER messages (outputs time spend and analysis)
    // ---------------------------------------------------------------------------------
    public void END_HOP_TEST_HEADER(String content) throws IOException
    {
         int index = content.indexOf(":");

         String orig_time = content.substring(index+2, content.length());
         long start_time = Long.parseLong(orig_time);

         // Retrieve the number of milliseconds since 1/1/1970 GMT
        Date date = new Date();
        long end_time = date.getTime();

        long elasped_time = end_time - start_time;

        System.out.println("Total elasped time is: " + elasped_time + "milliseconds");


    } // end END_HOP_TEST_HEADER



    // ------------------------------------------------------------------------
    // This function returns the number of neighbours this node is connected to
```

```
    // --------------------------------------------------------------------------------
    private int number_of_neighbours() throws IOException
    {
        int count = 0;
        BufferedReader bufReader = new BufferedReader(new FileReader("neighbour_list.txt"));

        while(bufReader.readLine() != null)
        {
            count++;
        }
        bufReader.close();

        return count;
    }   // end function
    // --------------------------------------------------------------------------------
    // This function returns the number of global nodes
    // --------------------------------------------------------------------------------
    private int number_of_globals() throws IOException
    {
        int count = 0;
        BufferedReader bufReader = new BufferedReader(new FileReader("global_list.txt"));

        while(bufReader.readLine() != null)
        {
            count++;
        }
        bufReader.close();

        return count;
    }   // end function
    // --------------------------------------------------------------------------------
    // This functions checks if incoming content already exist in file
    // --------------------------------------------------------------------------------
    private boolean content_exist(String filename, String content) throws IOException
    {
        boolean exist = false;
        String current_line;

        BufferedReader bufReader = new BufferedReader(new FileReader(filename));

        while( (current_line = bufReader.readLine()) != null)
        {
            if(current_line.startsWith(content))
            {
                exist = true;
                bufReader.close();
                return exist;
            }
        }
        return exist;
    }


    // --------------------------------------------------------------------------------
    // This is the JADE program that actually sends the message OUT
    // --------------------------------------------------------------------------------
    public void send_msg(String node_name, String message)
    {
        String responder = null;
        String dest = null;

                try     //trying to open socket for data going out
                {

                    dest = "http://".concat(node_name).concat(":7778/acc");
                    // Use String class manipulation to get responder address
                    int end_index = dest.lastIndexOf(":");
                    responder = "receiver@".concat(node_name).concat(":1099/JADE");

                    // Setup JADE send variables to use JADE to send the message out
                    AID r = new AID();
```

```

                        r.setName(responder);
                        r.addAddresses(dest);

                        // create the ACL message and set specs, then send the msg according to
                        // the user defined address

                        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
                        msg.setSender(getAID());
                        msg.addReceiver(r);

                        msg.setContent(message);
                        send(msg);

//                  finished = false;
                    }
                    catch(Exception e)
                    {
                        System.out.println("JADE send failed");
                    }

        }   // end send_msg()

        } // end class Receive

}//end class AgentReceiver

```

```
// ---------------------------------------------------------------------------
// INCLUDED JAVA FILES
// ---------------------------------------------------------------------------
package examples.receivers;

import java.net.*;
import java.util.*;
import java.io.*;
import java.lang.Thread;
// ---------------------------------------------------------------------------

// ---------------------------------------------------------------------------
// INCLUDED JADE FILES
// ---------------------------------------------------------------------------
import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.DFService;
import jade.domain.FIPAException;
// ---------------------------------------------------------------------------


public class Broadcast_receive extends Agent {

    class WaitPingAndReplyBehaviour extends SimpleBehaviour {
    private boolean finished = false;
    public WaitPingAndReplyBehaviour(Agent a) {
        super(a);
    }
    public void action() {
        // empty function, never gets here

    }   // end action
    public boolean done() {
        return finished;
    }
    } //End class WaitPingAndReplyBehaviour

    protected void setup() {


    try
    {
      Broadcast broadcast = new Broadcast();

      // Loop forever and receive host information from clients.
      //  the received messages.
      while (true)
      {

        MulticastSocket multicastSocket = new MulticastSocket();
        multicastSocket = broadcast.multicast_setup("230.0.0.1", 7777);

        // blocks here indefinitely until a message is received
        String message = broadcast.receive(multicastSocket);

        // determine if node already exists in global_list.txt
        if(!(broadcast.content_exist("global_list.txt", message)))
        {
            // write to global_list file
            broadcast.write("global_list.txt", message);
        }

        // Create an reply to tell the new Node that this current node is ON
        broadcast.reply(message);

    }   // end while
    }
    catch (Exception exception)
```

```
    {
      exception.printStackTrace();
    }

        // ---------------------------------------------------------------------------
        // REGISTRATION WITH DIRECTORY FACILITATOR (DF)
        // ---------------------------------------------------------------------------
        DFAgentDescription dfd = new DFAgentDescription();
        ServiceDescription sd = new ServiceDescription();
        sd.setType("Broadcast_receive Agent");
        sd.setName(getName());
        sd.setOwnership("Edward");
        //sd.addOntologies("PingAgent");
        dfd.setName(getAID());
        dfd.addServices(sd);
        try {
            DFService.register(this,dfd);
        } catch (FIPAException e) {
            System.err.println(getLocalName()+" registration with DF unsucceeded. Reason: "+e.g
            doDelete();
        }

        WaitPingAndReplyBehaviour PingBehaviour = new  WaitPingAndReplyBehaviour(this);
        addBehaviour(PingBehaviour);
        // ---------------------------------------------------------------------------

    } // END SETUP



//}//end class Broadcast_receive


private class Broadcast{

    Broadcast(){ //Begin Constructor

    // initialize all global variable in this class

   } //End Constructor


    // ---------------------------------------------------------------------------
    // This function sets up the multicast address and joins the group
    // ---------------------------------------------------------------------------
     public MulticastSocket multicast_setup(String MULTICAST_ADDR, int MULTICAST_PORT) throws I
    {
        MulticastSocket multicastSocket = new MulticastSocket(MULTICAST_PORT);
        InetAddress inetAddress = InetAddress.getByName(MULTICAST_ADDR);
        multicastSocket.joinGroup(inetAddress);

        return multicastSocket;
    }

    // ---------------------------------------------------------------------------
    // This function blocks indefinitely until a message is received on Multicast Port
    // ---------------------------------------------------------------------------
     public String receive(MulticastSocket multicastSocket) throws IOException
    {
        byte [] temp = new byte [1024];
        DatagramPacket datagramPacket = new DatagramPacket(temp, temp.length);

        // infinitely stuck here until receive a packet
        multicastSocket.receive(datagramPacket);
        String message = new String(datagramPacket.getData(), 0, datagramPacket.getLength());

        return message;
    }

    // ---------------------------------------------------------------------------
    // This function replies to the sender of the broadcast message
```

```
    // ------------------------------------------------------------------------------
     public void reply(String message) throws IOException
    {
        int index = message.indexOf("/");
        String node_name = message.substring(0,index);

        InetAddress ownAddress = get_own_Inet();
        String host_name = ownAddress.getHostName();
        String msg = "Broadcast_Setup: ".concat(host_name);
        send_msg(node_name, msg);
    }

    // ---------------------------------------------------------------------------------
    // This functions actually sends the message out to destination node
    // ---------------------------------------------------------------------------------
    private void send_msg(String node_name, String message)
    {
        String responder = null;
        String dest = null;
        try
        {
            dest = "http://".concat(node_name).concat(":7778/acc");
            // Use String class manipulation to get responder address
            int end_index = dest.lastIndexOf(":");
            responder = "receiver@".concat(node_name).concat(":1099/JADE");

//          System.out.println("responder: " + responder);
//          System.out.println("dest: " + dest);

            // Setup JADE send variables to use JADE to send the message out
            AID r = new AID();

            r.setName(responder);
            r.addAddresses(dest);

            // create the ACL message and set specs, then send the msg according to
            // the user defined address
            ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
            msg.setSender(getAID());
            msg.addReceiver(r);

            msg.setContent(message);
            send(msg);
            }
        catch(Exception e)
        {
            System.out.println("JADE send failed");
        }
    }   // end function

//-----------------------------------------------------------------------------------------
    // ------------------------------------------------------------------------------
    // This function writes the message to the specified file
    // ------------------------------------------------------------------------------
     public void write(String filename, String message) throws IOException
    {
        BufferedWriter bufWriter = new BufferedWriter(new FileWriter(filename, true));
        bufWriter.write(message);
        bufWriter.newLine();
        bufWriter.close();
        System.out.println("New node: " + message + " written to " + filename);
    }

    // ------------------------------------------------------------------------------
    // This function returns the InetAddress of the current host computer
    // ------------------------------------------------------------------------------
     public InetAddress get_own_Inet(){

        try                                                     //trying to set own ip-address
        {
            InetAddress ownIP = InetAddress.getLocalHost();
```

```
            return ownIP;
        }
        catch(UnknownHostException e)
        {
            System.out.println(e);
        }
        return null;
    }// end get_own_Inet()

    // ---------------------------------------------------------------------------------
    // This functions checks if incoming content already exist in file
    // ---------------------------------------------------------------------------------
    public boolean content_exist(String filename, String content) throws IOException
    {
        boolean exist = false;
        String current_line;

        BufferedReader bufReader = new BufferedReader(new FileReader(filename));


        while( (current_line = bufReader.readLine()) != null)
        {
            if(current_line.equalsIgnoreCase(content))
            {
                exist = true;
                bufReader.close();
                return exist;
            }
        }
        return exist;
    }

} // end class Broadcast


}//end class Broadcast_receive
```

APPENDIX B

This appendix contains the sample code listing for extending the JXTA distributed software platform.

```
//------------------------------------------------------------------------------------------
// INCLUDE JAVA FILES
//------------------------------------------------------------------------------------------
    import java.io.FileInputStream;
    import java.util.Date;
    import java.util.Enumeration;
    import java.io.FileWriter;
    import java.io.IOException;
    import java.net.*;
    import java.util.*;
    import java.io.*;
    import java.lang.Thread;
    import java.lang.*;
//------------------------------------------------------------------------------------------
// INCLUDE JXTA FILES
//------------------------------------------------------------------------------------------

import java.util.Enumeration;

    // DISCOVERY FILES
    import net.jxta.discovery.DiscoveryEvent;
    import net.jxta.discovery.DiscoveryListener;
    import net.jxta.discovery.DiscoveryService;
    import net.jxta.protocol.DiscoveryResponseMsg;
    import net.jxta.protocol.PeerAdvertisement;


    import net.jxta.endpoint.StringMessageElement;

    // RENDEZVOUS FILES
    import net.jxta.rendezvous.RendezvousEvent;
    import net.jxta.rendezvous.RendezvousListener;
    import net.jxta.rendezvous.RendezVousService;

    // DOCUMENT FILES
    import net.jxta.document.StructuredTextDocument;
    import net.jxta.document.AdvertisementFactory;
    import net.jxta.document.MimeMediaType;

    // ENDPOINT FILES
    import net.jxta.endpoint.Message;
    import net.jxta.endpoint.MessageElement;
    import net.jxta.endpoint.Message.ElementIterator;

    // PEERGROUP FILES
    import net.jxta.exception.PeerGroupException;
    import net.jxta.peergroup.PeerGroup;
    import net.jxta.peergroup.PeerGroupFactory;
    import net.jxta.impl.peergroup.StdPeerGroup;

    // PIPE FILES
    import net.jxta.pipe.InputPipe;
    import net.jxta.pipe.PipeMsgEvent;
    import net.jxta.pipe.PipeMsgListener;
    import net.jxta.pipe.PipeService;
    import net.jxta.pipe.OutputPipe;
    import net.jxta.pipe.OutputPipeEvent;
    import net.jxta.pipe.OutputPipeListener;
    import net.jxta.protocol.PipeAdvertisement;

    // ID FILES
    import net.jxta.id.ID;
    import net.jxta.id.IDFactory;

    // MISC
    import net.jxta.impl.endpoint.WireFormatMessage;
    import net.jxta.impl.endpoint.WireFormatMessageFactory;
    import net.jxta.util.CountingOutputStream;
    import net.jxta.util.DevNullOutputStream;
//------------------------------------------------------------------------------------------
// END INCLUDE FILES
```

```
//-------------------------------------------------------------------------------------



/*
Have an array of PipeAdv[] and using a while loop, bind all .XML (also in array) that is not N
remove line1 of .XML file
match array position with irms-client##
- still broadcast the .XML file to everyone on the list
- client1 goes online, sends to everyone, including client2
    - when client2 wants to send to client1, checks directory for client1.xml, if its exist
        bind to it
- broadcast --> also send back own .XML file


*/
public class PipeComm
{
        ///////////////////////////////////////////////////////////////////
        // GLOBAL VARIABLES                                              //
        ///////////////////////////////////////////////////////////////////


     public static void main (String[] args) throws IOException{

        PeerGroup netPeerGroup = null;
        boolean exitConsole = false;
        char userInput;

        InetAddress ownIP = InetAddress.getLocalHost();
        String host_name = ownIP.getHostName();

        ///////////////////////////////////////////////////////////////////
        // DELETE PREVIOUS FILES                                         //
        ///////////////////////////////////////////////////////////////////
        delete_previous();

        ///////////////////////////////////////////////////////////////////


        ///////////////////////////////////////////////////////////////////
        // CREATE THE DEFAULT JXTA NETPEERGROUP                          //
        ///////////////////////////////////////////////////////////////////
       try {
            // create, and Start the default jxta NetPeerGroup
            netPeerGroup = PeerGroupFactory.newNetPeerGroup();
        }
        catch (PeerGroupException e) {
            // could not instantiate the group, print the stack and exit
            System.out.println("fatal error : group creation failure");
            e.printStackTrace();
            System.exit(1);
        }

        ///////////////////////////////////////////////////////////////////
        // GENERATE PIPE ADVERTISMENT AND BROADCAST TO EVERYONE          //
        ///////////////////////////////////////////////////////////////////
        generatePipeAdv(netPeerGroup);
        ///////////////////////////////////////////////////////////////////

        ///////////////////////////////////////////////////////////////////
        // INITIALIZE LISTENER/SENDER TO READY TO RECEIVING AND SENDING  //
        ///////////////////////////////////////////////////////////////////
        PipeListener listener = new PipeListener();
        PipeExample example = new PipeExample();
        listener.peergroup(netPeerGroup, example);
        example.peergroup(netPeerGroup);
        // start the listener
        listener.run();
        ///////////////////////////////////////////////////////////////////

```

```
        ///////////////////////////////////////////////////////////////////
        // READ IN ALL .XML FILES IN THIS DIRECTORY AND SEND FOR BINDING
        ///////////////////////////////////////////////////////////////////
        File homedir = new File( "C:\\jxta_devguide\\pipeservice" );
        //File homedir = new File(System.getProperty("user.home"));
        String[] XML_filename = homedir.list(new FilenameFilter() {
          public boolean accept(File d, String name) { return name.endsWith(".XML");
          }
        });
        for(int i=0; i< XML_filename.length; i++)
        {
            System.out.println(XML_filename[i]);
        }
        // bind to all input pipes
        listener.bind_input_pipe(XML_filename);
        ////////////////////////////////////////////////////////////////////////


        ///////////////////////////////////////////////////////////////////
        // MAIN EXECUTION OF THE MENU SYSTEM                             //
        ///////////////////////////////////////////////////////////////////

        while(!exitConsole)
         {
            userInput = main_menu();

                switch(userInput)
                 {
                    // send to specific node
                    case 'a':
                            send(example);
                    break;

                    // display all peers
                    case 'b':
                            display_all();
                    break;

                    // display all neighbour peers
                    case 'c':
                            display_neighbour_peers();
                        break;

                    // Add a neighbour peer
                    case 'd':
                            add_neighbour_peer(example, host_name);
                    break;

                    // Add a neighbour peer
                    case 'e':
                            send_multi_hop(example);
                    break;

                    // Update hop peer
                    case 'f':
                            update_hop_peer(example);
                    break;
                    // exit
                    case 'x':
                        System.exit(0);
                    break;


                    default:
                        System.out.println("Error input!!");
                    break;
                 } // end switch
         }
    } // end main

public static void send(PipeExample example) throws IOException
```

```java
{
    System.out.println("Enter peer name: ");
    String node_name = getstring();
    System.out.println("Enter message: ");
    String message = getstring();

    example.set_message(message);

    String file_path = "C:\\jxta_devguide\\pipeservice\\".concat(node_name);
    file_path = file_path.concat(".XML");
    File myFile = new File( file_path );

    // only attempt to send when a valid node
    if(myFile.exists())
    {
        example.send_name(node_name);
        example.run();
    }
    else
     {
    System.out.println("INVALID NODE -- DOES NOT EXIST");
     }
} // end send()

public static void  display_all() throws IOException
{
    File homedir = new File( "C:\\jxta_devguide\\pipeservice" );
    //File homedir = new File(System.getProperty("user.home"));
    String[] XML_filename = homedir.list(new FilenameFilter() {
      public boolean accept(File d, String name) { return name.endsWith(".XML");
      }
    });

    System.out.println("");
    System.out.println("All Peers available");

    for(int i=0; i< XML_filename.length; i++)
    {
        String temp = XML_filename[i].substring(0, XML_filename[i].indexOf("."));
        System.out.println("Peer_" + i + ": " + temp);
    }
}

// ----------------------------------------------------------------------------------------
    public static void display_neighbour_peers() throws IOException
    {
        try{
            BufferedReader bufReader = new BufferedReader(new FileReader("neighbour_peer.txt")
            String current_line;

            System.out.println("Displaying all 1st neighbour peer");
            System.out.println(" ");
            int i=0;

            while( (current_line = bufReader.readLine()) != null)
            {
                System.out.println("Neighbour Peers#" + i + " " + current_line);
                i++;
            }
        }catch(Exception e){
            System.out.println("No nodes are currently connected");
        }

    } // end display_neighbour_nodes()
// --------------------------------------------------------------------------------

public static void add_neighbour_peer(PipeExample Sender, String host_name) throws IOException
{
        System.out.println("Enter name of peer: ");
        String peer_name = getstring();

```

```
        String file_path = "C:\\jxta_devguide\\pipeservice\\".concat(peer_name);
        file_path = file_path.concat(".XML");
        File myFile = new File( file_path );

        // check if content already exist and is a valid peer
        if((!content_exist("neighbour_peer.txt", peer_name)) && (myFile.exists()))
        {
            // Open the neighbour_peer.txt file to write to
            BufferedWriter bufWriter = new BufferedWriter(new FileWriter("neighbour_peer.txt",
            // write to file
            bufWriter.write(peer_name);
            bufWriter.newLine();
            bufWriter.close();
            System.out.println("New peer: " + peer_name + " is written to neighbour_peer.txt")

            //////////////////////////////////////////////////////////////////////////////
            // NOW SEND THIS INFORMATION TO THE OTHER PEER FOR SETUP AS WELL           //
            //////////////////////////////////////////////////////////////////////////////
            String message = "ADMIN_SETUP: ".concat(host_name);
            Sender.set_message(message);
            Sender.send_name(peer_name);
            Sender.run();
        }
        else
        {
            System.out.println("Peer: " + peer_name + " is not a valid peer");
        }

} // end function

public static void send_multi_hop(PipeExample Sender) throws IOException
{
       String Multi_Hop_Header = "MULTI_HOP_MESSAGE_HEADER: ";
       String Multi_Hop_Message;
       String message;
       String dest_peer;
       String MAX_HOP;

       System.out.println("Enter destination peer");
       dest_peer = getstring();

    if(peer_exist(dest_peer))
    {
       System.out.println("Enter message");
        message = getstring();

        System.out.println("Enter Maximum number of hops allowed");
        MAX_HOP = getstring();

        // Multi_Hop_Message_Header: 3#destination$msg_body
        Multi_Hop_Message = Multi_Hop_Header.concat(MAX_HOP.concat("#".concat(dest_peer.concat

        System.out.println("multi_hop_message: " + Multi_Hop_Message);

        // check if destination is already a neighbour_node
        int front = Multi_Hop_Message.indexOf("#");
        int back = Multi_Hop_Message.indexOf("$");


        // if already in neighbour list
        if(content_exist("neighbour_peer.txt", Multi_Hop_Message.substring(front+1, back)))
        {
            message = Multi_Hop_Message.substring(back+1, Multi_Hop_Message.length());
            Sender.set_message(message);
            Sender.send_name(dest_peer);
            Sender.run();
            // Extract the message
        }
        else
        {
            // actually send the message to everyone on neighbor list
```

```
            BufferedReader bufReader = new BufferedReader(new FileReader("neighbour_peer.txt")
            String current_line;
            InetAddress ownIP = InetAddress.getLocalHost();
            String host_name = ownIP.getHostName();

            while( (current_line = bufReader.readLine()) != null)
            {
                // Don't send message to itself
                if(!(current_line.startsWith(host_name)))
                {
                    Sender.set_message(Multi_Hop_Message);
                    Sender.send_name(current_line);
                    Sender.run();
                }
            } // end while
        } // end else

    } // end if
    else
    {
        System.out.println("INVALID PEER NAME");

    }
} // end function

public static void update_hop_peer(PipeExample Sender) throws IOException
{
        //Update_Hop_Message_Header: sender_node#HOP_COUNT#PREVIOUS_sender_node#Original_hop_C
        System.out.println("Updating global hop peer.... Please wait");

        // update from 2 hops to 5 hops.... TO BE CHANGED!!!!!!!!!!!!!!!!!!
        String UPDATE_HOP_HEADER = "UPDATE_HOP_MESSAGE_HEADER: ";

        InetAddress ownIP = InetAddress.getLocalHost();
        String host_name = ownIP.getHostName();

        String hop_count;
        String neighbour_name;
        String current_line;
        String UPDATE_HOP_MESSAGE;

        for(int i=1; i<2; i++)
        {
            // i is hop count

            hop_count = String.valueOf(i);
            UPDATE_HOP_MESSAGE = host_name.concat("#".concat(hop_count.concat("#")));
            UPDATE_HOP_MESSAGE = UPDATE_HOP_HEADER.concat(UPDATE_HOP_MESSAGE);

            // actually send the message to everyone on neighbor list
            BufferedReader bufReader = new BufferedReader(new FileReader("neighbour_peer.txt")

            while( (current_line = bufReader.readLine()) != null)
            {
                // Don't send message to itself
                if(!(current_line.startsWith(host_name)))
                {
                    if(current_line.endsWith("#"))
                    {
                        UPDATE_HOP_MESSAGE = UPDATE_HOP_MESSAGE.concat(hop_count);
                    }
                    else
                    {
                        UPDATE_HOP_MESSAGE = UPDATE_HOP_MESSAGE.substring(0,UPDATE_HOP_MESSAGE
                        UPDATE_HOP_MESSAGE = UPDATE_HOP_MESSAGE.concat(hop_count);
                    }
                    System.out.println(UPDATE_HOP_MESSAGE);
                    Sender.set_message(UPDATE_HOP_MESSAGE);
                    Sender.send_name(current_line);
                    Sender.run();
                }
```

```
            } // end while
            bufReader.close();
        } // end for
} // end function


public static char main_menu() throws IOException
{

            char userInput;

            System.out.println(" ");
            System.out.println(" ");
            System.out.println("             Welcome to JXTA\n");
            System.out.println("An Innovative Approach to Distributed Communication. ");
            System.out.println(" ");
            System.out.println("Please select one of the following options: ");
            System.out.println(" ");
            System.out.println("   a) Send");
            System.out.println("   b) Dispaly ALL peers");
            System.out.println("   c) Dispaly ALL neighbour peers");
            System.out.println("   d) Add neighbour peers");
            System.out.println("   e) Send by Multi-Hop");
            System.out.println("   f) Update HOP list");
            System.out.println("   x) Exit");

            System.out.println(" ");
            System.out.println(" ");
            System.out.println(" ");
            System.out.print("Please make your selection: ");

            try{
                 userInput = get_char();
                 return userInput;
            }
            catch(Exception e){
                System.out.println(e);
            }

            // dummy return
            return 'x';


} // end main_menu

// -------------------------------------------------------------------------------
// Generate a pipe advertisement
public static void generatePipeAdv(PeerGroup netPeerGroup) throws IOException
{
    DiscoveryService discovery = netPeerGroup.getDiscoveryService();
   // Create a new Pipe Advertisement object instance.
    PipeAdvertisement pipeAdv =
        (PipeAdvertisement) AdvertisementFactory.newAdvertisement(
            PipeAdvertisement.getAdvertisementType());
       // Create a unicast Pipe Advertisement.
    pipeAdv.setName("IRMS COMMUNICATION PIPE");
    pipeAdv.setPipeID((ID) IDFactory.newPipeID(netPeerGroup.getPeerGroupID()));
    pipeAdv.setType(PipeService.UnicastType);

     // Save the document into the public folder
 //     discovery.publish(pipeAdv, DiscoveryService.ADV);
   //  discovery.remotePublish(pipeAdv, DiscoveryService.ADV);

    writePipeAdv(pipeAdv);
}

// -------------------------------------------------------------------------------
// Write the advertisement to file, and broadcast to everybody
private static void writePipeAdv(PipeAdvertisement pipeAdv)
    {
        // Create an XML formatted version of the Pipe Advertisement.
```

```
        try
        {
            // get local host_name
            InetAddress ownIP = InetAddress.getLocalHost();
            String host_name = ownIP.getHostName();
            host_name = host_name.concat(".XML");

            FileWriter file = new FileWriter(host_name);
            MimeMediaType mimeType = new MimeMediaType("text/xml");
            StructuredTextDocument document =
                (StructuredTextDocument) pipeAdv.getDocument(mimeType);

            // Output the XML for the advertisement to the file.
            document.sendToWriter(file);
            file.close();
            broadcast();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }

}

// ---------------------------------------------------------------------------------
public static void broadcast()
{
    int MULTICAST_PORT =  7777;
    String MULTICAST_ADDR = "230.0.0.1";
    String current_line = "";
    String broadcast_file = "";
        try
    {
        // get local host_name
        InetAddress ownIP = InetAddress.getLocalHost();
        String host_file = ownIP.getHostName();
        broadcast_file = host_file.concat("#");
        host_file = host_file.concat(".XML");

        BufferedReader bufReader = new BufferedReader(new FileReader(host_file));

        while( (current_line = bufReader.readLine()) != null)
        {
            broadcast_file = broadcast_file.concat(current_line);
            broadcast_file = broadcast_file.concat("#");
        }
        bufReader.close();

      byte[] temp = broadcast_file.getBytes();
      InetAddress inetAddress = InetAddress.getByName(MULTICAST_ADDR);
      DatagramPacket Out_Packet = new DatagramPacket(temp, temp.length, inetAddress, MULTICAST_
      MulticastSocket multicastSocket = new MulticastSocket();
      multicastSocket.send(Out_Packet);
    }
    catch (Exception exception)
    {
      exception.printStackTrace();
    }

} // end broadcast

public static boolean peer_exist(String peer_name)
{
        String file_path = "C:\\jxta_devguide\\pipeservice\\".concat(peer_name);
        file_path = file_path.concat(".XML");
        File myFile = new File( file_path );

        if(myFile.exists())
        {
            return true;
        }
```

```
        else
        {
            return false;
        }
}
// -------------------------------------------------------------------------------
    public static void delete_previous() throws IOException
    {
        // get all *.XML files within directory
        File homedir2 = new File( "C:\\jxta_devguide\\pipeservice" );
        //File homedir = new File(System.getProperty("user.home"));
        String[] XML_filename2 = homedir2.list(new FilenameFilter() {
      public boolean accept(File d, String name) { return name.endsWith(".XML");
          }
        });
        for(int i=0; i< XML_filename2.length; i++)
        {
            System.out.println(XML_filename2[i]);
            File delete_file = new File(XML_filename2[i]);
            delete_file.delete();

        }

        InetAddress ownIP = InetAddress.getLocalHost();
        String host_name = ownIP.getHostName();

        File myFile = new File( "C:\\jxta_devguide\\PipeService\\neighbour_peer.txt" );
        myFile.delete();
        File myFile2 = new File( "C:\\jxta_devguide\\PipeService\\hop_peer.txt" );
        myFile2.delete();

        BufferedWriter bufWriter = new BufferedWriter(new FileWriter("neighbour_peer.txt", tru
        bufWriter.write(host_name);
        bufWriter.newLine();
        bufWriter.close();

        BufferedWriter bufWriter2 = new BufferedWriter(new FileWriter("hop_peer.txt", true));
        String host = host_name.concat("#5");
        bufWriter2.write(host);
        bufWriter2.newLine();
        bufWriter2.close();

    } // end function

// -------------------------------------------------------------------------------
// This functions checks if incoming content already exist in file

public static boolean content_exist(String filename, String content) throws IOException
{
        boolean exist = false;
        String current_line;

        BufferedReader bufReader = new BufferedReader(new FileReader(filename)};

        while( (current_line = bufReader.readLine()) != null)
        {
            if(current_line.equalsIgnoreCase(content))
            {
                exist = true;
                bufReader.close();
                return exist;
            }
        }
        return exist;
}   // end function content_exist()
// -------------------------------------------------------------------------------

// -------------------------------------------------------------------------------
// This functions returns the character input from the user
    public static char get_char() throws IOException
    {
```

```
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String s = br.readLine();
        return s.charAt(0);

    } // end get_char()


// -----------------------------------------------------------------------------------------

// This function returns the entire line of String
    public static String getstring() throws IOException
    {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String s = br.readLine();
        return s;

    } // end getString()

// --------------------------------------------------------------------------------

} // end class PipeComm



class PipeListener implements PipeMsgListener {

    static PeerGroup netPeerGroup = null;
    private final static String SenderMessage = "PipeListenerMsg";

    String[] hop_peer = new String [100];
    private PipeService pipe;
    private PipeAdvertisement pipeAdv;
    private InputPipe pipeIn1 = null;
    private InputPipe pipeIn2 = null;
    InputPipe pipeIn[] = new InputPipe[20];//ull;
    PipeExample Sender = new PipeExample(); // get netPeerGroup from MAIN

    public void peergroup(PeerGroup group, PipeExample example)
    {
        netPeerGroup = group;
        pipe = netPeerGroup.getPipeService();
        Sender = example;
  /*      System.out.println("Reading in pipexample.adv");
        try {
            FileInputStream is = new FileInputStream("era-pj57q9emaot.XML");
            pipeAdv = (PipeAdvertisement) AdvertisementFactory.newAdvertisement(MimeMediaType.
            is.close();
        } catch (Exception e) {
            System.out.println("failed to read/parse pipe advertisement");
            e.printStackTrace();
            System.exit(-1);
        }
        */
    }

    // bind to specified input pipe
    public void bind_input_pipe(String[] XML_filename) throws IOException
    {
            InetAddress ownIP = InetAddress.getLocalHost();
            String host_name = ownIP.getHostName();

        for(int i=0; i< XML_filename.length; i++)
        {
            try{
            pipe = netPeerGroup.getPipeService();
            System.out.println("Reading in "  + XML_filename[i]);

            if(XML_filename[i].startsWith(host_name))
                {
```

```
                FileInputStream is = new FileInputStream(XML_filename[i]);
                pipeAdv = (PipeAdvertisement) AdvertisementFactory.newAdvertisement(MimeMediaT
                is.close();

                 pipeIn[i] = pipe.createInputPipe(pipeAdv, this);
                 System.out.println("written");
                }
            } catch (Exception e) {
                System.out.println("failed to read/parse pipe advertisement");
                e.printStackTrace();
                System.exit(-1);
            }

        }
    } // end bind_input_pipe

    public static void printMessageStats(Message msg, boolean verbose) {
        try {
            CountingOutputStream cnt;
            ElementIterator it = msg.getMessageElements();
            System.out.println("------------------Begin Message---------------------");
            WireFormatMessage serialed = WireFormatMessageFactory.toWire(
                                            msg,
                                            new MimeMediaType("application/x-jxta-msg"), (Mim
            System.out.println("Message Size :" + serialed.getByteLength());
            while (it.hasNext()) {
                MessageElement el = (MessageElement) it.next();
                String eName = el.getElementName();
                cnt = new CountingOutputStream(new DevNullOutputStream());
                el.sendToStream(cnt);
                long size = cnt.getBytesWritten();
                System.out.println("Element " + eName + " : " + size);
                if (verbose) {
                    System.out.println("["+el+"]");
                }
            }
            System.out.println("-------------------End Message----------------------");
        } catch (Exception e) {
            e.printStackTrace();

        }
    }

    /**
     * wait for msgs
     *
     */

    public void run() {

        try {
            // the following creates the inputpipe, and registers "this"
            // as the PipeMsgListener, when a message arrives pipeMsgEvent is called
            System.out.println("Creating input pipe");
          //  pipeIn = pipe.createInputPipe(pipeAdv, this);
        } catch (Exception e) {
            return;
        }
     //   if (pipeIn == null) {
       //     System.out.println(" cannot open InputPipe");
         //   System.exit(-1);
       // }
        System.out.println("Waiting for msgs on input pipe");
    }


    /**
     * By implementing PipeMsgListener, define this method to deal with
     * messages as they arrive
     */

```

```
    public void pipeMsgEvent(PipeMsgEvent event){

        String ADMIN_HEADER = "ADMIN_SETUP: ";
        String MULTI_HOP_HEADER = "MULTI_HOP_MESSAGE_HEADER: ";
        String UPDATE_HOP_HEADER = "UPDATE_HOP_MESSAGE_HEADER: ";
        String UPDATE_HOP_LIST_HEADER = "UPDATE_HOP_LIST_HEADER: "; // header to write final h

        Message msg=null;
        try {
            // grab the message from the event
            msg = event.getMessage();
            if (msg == null) {
                return;
            }
//            printMessageStats(msg, true);
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }

        // get all the message elements
        Message.ElementIterator enum = msg.getMessageElements();
        if (!enum.hasNext()) {
            return;
        }

        // get the message element named SenderMessage
        MessageElement msgElement = msg.getMessageElement(null, SenderMessage);
        String received = msgElement.toString();
        // Get message
/*        if (msgElement.toString() == null) {
            System.out.println("null msg received");
        } else {
             System.out.println("Message received: "+ msgElement.toString());
        }
*/
        //ADMIN RECEIVED, SETUP NEIGHBOUR LIST
        if(received.startsWith(ADMIN_HEADER))
        {
            try{
                received = received.substring(received.indexOf(":")+2, received.length());
                boolean exist = content_exist("neighbour_peer.txt", received);
                if(!exist)
                {
                    BufferedWriter bufWriter = new BufferedWriter(new FileWriter("neighbour_pe
                    // write to file
                    bufWriter.write(received);
                    bufWriter.newLine();
                    bufWriter.close();
                    System.out.println("Peer: " + received + " is added remotely by Administra
                } // end if
            } // end try
            catch (Exception exception)
            {
                 exception.printStackTrace();
            }
        } // end if

        // MULTI_HOP MESSAGE RECEIVED, DECREMENT COUNT AND FORWARD
        else if(received.startsWith(MULTI_HOP_HEADER))
        {
        /*
            Check if dest_node is a neighbour node, if yes, send directly
            if not, decrement hop count and send to all neighbour node
            if hop_count==0, discard (send msg failed??)
        */
            // Extract destination node to see if neighbour node

        try{
            int front = received.indexOf("#");
            int back = received.indexOf("$");
```

```

            // if already in neighbour list, send directly
            if(content_exist("neighbour_peer.txt", received.substring(front+1, back)))
            {
                // Multi_Hop_Message_Header: 3#destination$msg_body

                // Extract the message and send to destination
                String temp = received.substring(back+1, received.length());
                Sender.set_message(temp);
                Sender.send_name(received.substring(front+1, back));
                Sender.run();
                //send_msg(content.substring(front+1, back), temp);
                System.out.println("to neighbour: " + received.substring(front+1, back));
            }
                // decrement Hop count and send to all neighbour
            else
            {
                // extract hop count
                // Multi_Hop_Message_header: 3#destination$msg_body
                    int start = received.indexOf(":");
                    String hop = received.substring(start+2, front);

                    // decrement hop_count
                    int temp_hop = Integer.parseInt(hop);
                    temp_hop--;
                    hop = String.valueOf(temp_hop);

                    InetAddress ownIP = InetAddress.getLocalHost();
                    String host_name = ownIP.getHostName();
                    // make everything lower case, just to be safe
                    host_name = host_name.toLowerCase();

                    // go through neighbour list and send to all neighbours
                    if(temp_hop>0)
                    {
                        received = received.substring(front, received.length());

                        // make new MULTI_HOP String
                        String NEW_MULTI_HOP_MESSAGE = MULTI_HOP_HEADER.concat(hop.concat(rece
                        System.out.println("new_received: " + NEW_MULTI_HOP_MESSAGE);

                        // actually send the message to everyone on neighbor list
                        BufferedReader bufReader = new BufferedReader(new FileReader("neighbou
                        String current_line;
                            // don't send to itself

                        while( (current_line = bufReader.readLine()) != null)
                        {
                            // Don't send message to itself
                            if(!(current_line.startsWith(host_name)))
                            {
                                Sender.set_message(NEW_MULTI_HOP_MESSAGE);
                                Sender.send_name(current_line);
                                Sender.run();
                            } //end if
                        } // end while
                        bufReader.close();
                    }   // end if(temp_hop>0)
            } // end else
            } // end try
            catch (Exception exception)
            {
                 exception.printStackTrace();
            }
        }   // end else if

        // *****************************************************************
        // Update_Hop_List message
        // *****************************************************************
        // test if message is to update_hop_list
        else if(received.startsWith(UPDATE_HOP_HEADER))
```

```
        {
            try{
            //sender_node#HOP_COUNT#PREVIOUS_sender_node#Original_hop_Count

            int first = received.indexOf("#");
            int second = received.indexOf("#", first+1);
            int end = received.lastIndexOf("#");
            // convert to INT
            int hop_count = Integer.parseInt(received.substring(first+1, second));

            String final_hop_count="";
            String original_sender;
            String current_line;
            String new_hop_count;

            // get current host name
            InetAddress ownIP = InetAddress.getLocalHost();
            String host_name = ownIP.getHostName();

        if(hop_count>0)
        {
                hop_count = hop_count-1;
        }

        // hop_count==0!!!!!!!!!!!!!!!!!!!!!!!!!!!
        if(hop_count==0)
        {
            // end of hop reached, send back to sender with hop info

            //Update_Hop_List_Header: current_node#original_hop_count
            int space = received.indexOf(" ");
                String UPDATE_HOP_LIST_MESSAGE = UPDATE_HOP_LIST_HEADER.concat(host_name);
                UPDATE_HOP_LIST_MESSAGE = UPDATE_HOP_LIST_MESSAGE.concat(received.substring(en

                original_sender = received.substring(space+1, first);
                System.out.println("original: " + original_sender);
                System.out.println("UPDATE_HOP_LIST_MESSAGE: " + UPDATE_HOP_LIST_MESSAGE);
                // send to original sender TO BE MODIFIED!!!!!!!!
                Sender.set_message(UPDATE_HOP_LIST_MESSAGE);
                Sender.send_name(original_sender);
                Sender.run();
        }

        // end of hop NOT reached, send out
        else
        {
            // Update_Hop_Message_Header: sender_node#HOP_COUNT#PREVIOUS_sender_node#Original_

            //Replace the hop_count and send to everyone on the list, except to itself
            new_hop_count = String.valueOf(hop_count);
            System.out.println("hop_count: "  + hop_count);

            // get message header
            String temp_content = received.substring(0,first);

            // attach new hop_count
            temp_content = temp_content.concat("#".concat(new_hop_count));

            // get original hop count
            temp_content = temp_content.concat(received.substring(second, received.length()));

            // send to everyone on neighbour list with new hop_count
            BufferedReader bufReader = new BufferedReader(new FileReader("neighbour_peer.txt")

            // actually send the message to everyone on neighbor list, except itself AND previ
            String previous_sender = received.substring(second+1,end);

            while( (current_line = bufReader.readLine()) != null)
            {
                    // Don't send message to itself
                    if(!(current_line.startsWith(host_name)) && !(current_line.equalsIgnoreCa
```

```
                    {
                        Sender.set_message(temp_content);
                        Sender.send_name(current_line);
                        Sender.run();
                    } //end if
            } // end while
            bufReader.close();
        } // end else
            } // end try
            catch (Exception exception)
            {
                 exception.printStackTrace();
            }
        } // end else if

        // ******************************************************************
        // Update_Hop_List message
        // ******************************************************************
        // Update hop_list.txt, get only the shortest hops away
        else if(received.startsWith(UPDATE_HOP_LIST_HEADER))
        {
        try{
            System.out.println("rewrite: " + received);

            //Update_Hop_List_Header: end_node#original_hop_count
            int space = received.indexOf(" ");
            int seperator = received.indexOf("#");

            String end_node = received.substring(space+1, seperator);
            String final_count = received.substring(seperator+1, received.length());

            // loop through hop_list to record hop info, only take the info with least hops!!
            BufferedReader bufReader = new BufferedReader(new FileReader("hop_peer.txt"));

            //copy file into hop_list array, then delete file
            String current_line;
            int counter = 1;
            while( (current_line = bufReader.readLine()) != null)
            {
                    hop_peer[counter] = current_line;
                    counter++;
            } // end while

            // delete the file
            bufReader.close();
            File myFile = new File( "C:\\jxta_devguide\\PipeService\\hop_peer.txt" );
            myFile.delete();


                    counter = 1;
                    String node_in_file;
                    String old_node_count;
                    String replacement;
                    String node_name_in_file;
                    String received_node;

                    int message_count;
                    int array_count;
                    boolean node_exist = false;

                    int sept;
                    while( hop_peer[counter] != null)
                    {
                        node_in_file = hop_peer[counter];
                        sept = node_in_file.indexOf("#");
                        node_name_in_file = node_in_file.substring(0,sept);

                        // if node exist
                        if( end_node.equalsIgnoreCase(node_name_in_file))
                        {
                            node_exist = true;
```

```
                            // get node_count from string array (File)
                            array_count = Integer.parseInt(node_in_file.substring(sept+1, node_

                            // get node_count from message
                            message_count = Integer.parseInt(received.substring(seperator+1, r

                            // replace array if hop is now smaller
                            if(message_count < array_count)
                            {
                                replacement = node_in_file.substring(0,sept+1);
                                replacement = replacement.concat(String.valueOf(message_count)
                                hop_peer[counter] = replacement;
                                System.out.println("UPDATED hop_peer: " + replacement);
                            }
                        } // end if

                        counter++;
                    }   // end while

                    if(!node_exist)
                    {
                        hop_peer[counter++] = received.substring(space+1, received.length());
                    }

                    // open up new hop_list file and write
                    BufferedWriter bufWriter = new BufferedWriter(new FileWriter("hop_peer.txt

                    int i=1;

                    while( i<counter )
                    {
                        bufWriter.write(hop_peer[i]);
                        bufWriter.newLine();
                        i++;
                    }

                    bufWriter.close();


    } // end try
            catch (Exception exception)
            {
                 exception.printStackTrace();
            }




} // end else if










//----------------------------------------------------------------------

        else
        {
            System.out.println("RECEIVED: " + received);
        }


    }// end function


// ------------------------------------------------------------------------------
// This functions checks if incoming content already exist in file
```

```

    public static boolean content_exist(String filename, String content) throws IOException
    {
        boolean exist = false;
        String current_line;

        BufferedReader bufReader = new BufferedReader(new FileReader(filename));

        while( (current_line = bufReader.readLine()) != null)
        {
            if(current_line.equalsIgnoreCase(content))
            {
                exist = true;
                bufReader.close();
                return exist;
            }
        }
        return exist;
    }   // end function content_exist()
// ----------------------------------------------------------------------------

} // end class

class PipeExample implements
                              Runnable,
                              OutputPipeListener,
                              RendezvousListener {

    static PeerGroup netPeerGroup = null;
    private final static String SenderMessage = "PipeListenerMsg";
    private PipeService pipe;
    private DiscoveryService discovery;
    private PipeAdvertisement pipeAdv;
    private RendezVousService rendezvous;
    String message = "";

    String dest_node;

    public void set_message(String msg)
    {
        message = msg;
    }

    public void send_name(String name)
    {
        dest_node = name;
    }

    public void peergroup(PeerGroup group)
    {
        netPeerGroup = group;
       // get the pipe service, and discovery
        pipe = netPeerGroup.getPipeService();
        discovery = netPeerGroup.getDiscoveryService();
    }

    /**
     *  the thread which creates (resolves) the output pipe
     *  and sends a message once it's resolved
     */

    public synchronized void run() {
        try {

            dest_node = dest_node.concat(".XML");
            System.out.println("Reading in " + dest_node);
            FileInputStream is = new FileInputStream(dest_node);
            pipeAdv = (PipeAdvertisement) AdvertisementFactory.newAdvertisement(MimeMediaType.
            is.close();

            // this step helps when running standalone (local sub-net without any redezvous se
```

```
            discovery.getRemoteAdvertisements(null, DiscoveryService.ADV, null, null, 1, null)
            // create output pipe with asynchronously
            // Send out the first pipe resolve call
            System.out.println("Attempting to create a OutputPipe");
            pipe.createOutputPipe(pipeAdv, this);
     /*       // send out a second pipe resolution after we connect
            // to a rendezvous
            if (!rendezvous.isConnectedToRendezVous()) {
                System.out.println("Waiting for Rendezvous Connection");
                try {
                    wait();
                    System.out.println("Connected to Rendezvous, attempting to create a Output
                    pipe.createOutputPipe(pipeAdv, this);
                } catch (InterruptedException e) {
                    // got our notification
                }
            }*/
        } catch (IOException e) {
            System.out.println("OutputPipe creation failure");
            e.printStackTrace();
            System.exit(-1);
        }
    }


    /**
     *  by implementing OutputPipeListener we must define this method which
     *  is called when the output pipe is created
     *
     *@param  event  event object from which to get output pipe object
     */

    public void outputPipeEvent(OutputPipeEvent event) {

        System.out.println(" Got an output pipe event");
        OutputPipe op = event.getOutputPipe();
        Message msg = null;

        try {
            System.out.println("Sending message");
            msg = new Message();
            Date date = new Date(System.currentTimeMillis());
            StringMessageElement sme = new StringMessageElement(SenderMessage, message , null)
            msg.addMessageElement(null, sme);
            op.send(msg);
        } catch (IOException e) {
            System.out.println("failed to send message");
            e.printStackTrace();
            System.exit(-1);
        }
        op.close();
        System.out.println("message sent");
    }

    /**
     *  rendezvousEvent the rendezvous event
     *
     *@param  event   rendezvousEvent
     */
    public synchronized void rendezvousEvent(RendezvousEvent event) {
        if (event.getType() == event.RDVCONNECT) {
            notify();
        }
    }
}


```