1-1-2007

Design and implementation of an unstructured overlay middleware to support MANET applications

Mashael Saad Al-Sabah
Ryerson University

DESIGN AND IMPLEMENTATION OF AN UNSTRUCTURED OVERLAY MIDDLEWARE TO SUPPORT MANET APPLICATIONS

By

Mashael Saad Al-Sabah

B.Sc. in Computer Science with High Honors
American University in Cairo, Egypt, 2004

A thesis
Presented to Ryerson University
In partial fulfillment of the
Requirements of the degree of
Master of Applied Science
In the program of
Computer Networks

Toronto, Ontario, Canada, 2007

Author's Declaration:

I hereby declare that I am the sole author of this thesis

I authorize Ryerson University to lend this thesis or dissertation to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis or dissertation by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Acknowledgments

I would like to express my deep gratitude to my supervisor, Dr. Muhammad Jaseemuddin for his great guidance and assistance throughout this research, never accepting less than my best efforts.

I would also like to thank Dr. Bobby Ma for all his support.

Finally, I would like to thank my parents for their continuous encouragement.

Abstract
Mashael Al-Sabah,
Design and Implementation of an Unstructured Overlay Middleware to Support MANET Applications
M.A.Sc, Computer Networks, Ryerson University, 2007

Recently, mobile ad hoc networks (MANET) have gained an enormous popularity, which has led to an increasing need for the development of MANET applications. Developing MANET application, however, is not an easy task. Developers have to deal with difficulties inherent from the wireless network environments.

In this thesis, the design of a middleware that abstracts the functionalities of an unstructured overlay network is proposed. The main non-functional requirements that our design tries to achieve is portability, efficiency, and usability. Portability is achieved by following the J2ME platform specifications. The middleware has been implemented and tested on a real time ad hoc network using different platform devices. The performance measurements of the middleware show that it achieves its efficiency requirements. Furthermore, the middleware's usability is validated by showing how different applications can be designed and deployed easily on top of it. It provides a simple MBR-based file-search and path establishment mechanisms which we have proved to be the basis for implementing a wide range of applications, such as file sharing, ALM and gaming applications.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ALM	Application Level Multicast
AODV	Ad hoc On demand Distance Vector
API	Application Programming Interface
ARP	Address Resolution Protocol
CAST	Scalable group Multicast /Anycast
CDC	Connected Device Configuration
CLDC	Connected Limited Device Configuration
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DHTs	Distributed Hash Tables
DOLR	Decentralized Object Location and Routing
DSR	Dynamic Source Routing
HP	Hewlett Packard
IBSS	Independent Basic Service Set
IDE	Integrated Development Kit
IGMP	Integrated Group Management protocol
IP	Internet Protocol
J2ME	Java 2 Micro Edition
JAR	Java Archive
JCL	Java Class Library
JVM	Java Virtual Machine
KBR	Key Based Routing
LAN	Local Area Network
LCS	Link Coloring Scheme
MANET	Mobile Ad hoc Network
MBR	Modified Biased Random Walk
MOM	Message oriented middleware
OLSR	Optimized Link State Routing
OOD	Object Oriented Design
OS	Operating System
OSPF	Open Shortest Path First
P2P	Peer to Peer

PDA	Personal Digital Assistant
RAM	Random Access Memory
RAON	Resource Aware Overlay Network
RREP	Route Reply
RREQ	Route Request
RTT	Round Trip Time
SIP	Session Initiation protocol
TCP	Transmission Control Protocol
TTL	Time To Live
UDP	User Datagram Protocol
UML	Unified Modeling Language
VoIP	Voice Over IP

1 Introduction

A Mobile Ad hoc network (MANET) is formed by mobile wireless nodes that can act as hosts and routers at the same time. In contrast with cellular structure where a single base station co-ordinates the link access and packet routing, no single node in MANET performs the co-ordination, rather link access and routing are distributed. IEEE 802.11 defines Independent Basic Service Set (IBSS), which is used to form MANET. The advantage of MANET over traditional infrastructure wireless LAN is that MANET can be formed in any place and at anytime without the need for installing and configuring an access point or a wireless router. However, the disadvantage of MANET is that control of the network has to be distributed among nodes, which makes the network management and administration tasks more complex; whereas in infrastructure wireless LANs, control is regulated by a special node usually refereed to as an access point. With the enormous availability and popularity of high end mobile devices, and with the widespread adoption of these devices, the popularity of MANET has increased and has led to a growing demand and interest in developing pervasive applications that can run on top of the ad hoc mobile nodes. Such applications require the cooperation of the nodes to perform distributed tasks. However, the characteristics of MANET impose many difficulties on application development over these environments.

Application development on MANET faces challenges due to network dynamics including ad hoc and unstable connectivity, abrupt disconnection; and due to resource constraints including limited power and computing resources. Overlay network on MANET is shown to be a promising approach for designing and deploying applications on MANET [5]. In this thesis we propose a middleware for overlay network that facilitates application design, development and deployment on MANET. The middleware builds the overlay topology and performs a query-query reply resource discovery mechanism on behalf of the user application.

In section 1.1, we introduce some of the challenges that application developers face when they design and implement applications that run on top of MANETs. These challenges

have earned a great deal of concern by MANET researchers over the past few years. Some key suggested solutions to these problems are presented in section 1.2. Then, we discuss our objectives and approach to middleware design for MANET application development.

1.1 Challenges of Application Development over MANETs

1.1.1 Network Dynamics

One difficulty that MANET application developers face is the dynamically changing network topology. Since nodes are allowed to move freely, the topology of the network changes accordingly and frequent disconnections may occur. Since management is distributed among all the nodes forming the ad hoc network, the logic of fault management must be examined and implemented by the developer, which would add further complexity on software development.

1.1.2 Diversity of Platforms

Another difficulty comes from the fact that the devices used to form the ad hoc network may be operated by different platforms, which introduces some portability issues. If an application developer writes an application for a specific platform using a specific programming language, the application may not work on a different platform. Example PDA operating systems include Palm OS, Windows mobile OS, Blackberry OS, Symbian OS, and Embedded Linux OS. Each operating system provides for developers its own set of APIs which are specific to their own platforms.

1.1.3 Resource-Constrained Devices

Mobile devices are very heterogeneous in terms of the resources they are equipped with like processing speed, power and memory capabilities and network interfaces. Mobile devices vary from one to another in terms of resource availability. Devices like laptops can offer fast CPUs and large amount of RAM and disk memory. For other mobile devices like pocket PCs and phones, power and memory are considered especially scarce resources because it is either expensive or impossible to augment these resources [20].

MANET application developers find this problem especially challenging because nodes have to perform routing operations in addition to their work.

1.2 Suggested Solutions to MANET Software Development Problem

1.2.1 Middleware

The term Middleware usually refers to a layer that acts an intermediate layer between different application components, providing a more functional set of Application Programming Interfaces (API) than the underlying platform. Mobile middleware allows the implementation of distributed applications connecting mobile and enterprise applications over wireless networks. Figure 1.1 depicts the middleware layer.

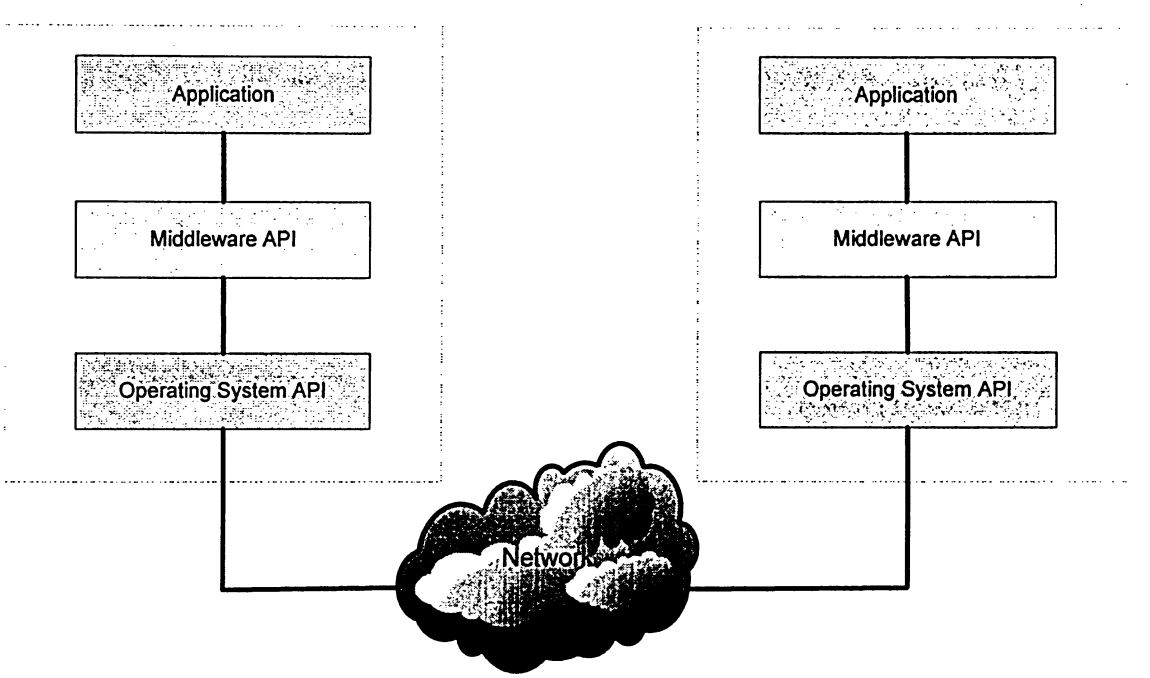

Figure 1.1 : Middleware Layer in the application-application communication structure

The advantages of using middleware are many, most important of which are the following:

- It allows two heterogeneous applications to work with each other because it provides a translation layer between the two applications.
- It facilitates software development. With the use of middleware, application developers will not need to go into the details of lower layer implementations, rather, they can focus on developing their applications.
- It provides platform independency and portability. A middleware is often designed so that it works as a cross-platform layer.
- It achieves location transparency. An application may communicate with other applications across networks, independent of network protocols.

The term middleware first appeared in the late 1980s to describe network connection management software [17]. Today, there are many categories of middleware, most common of which are:

- Message oriented middleware (MOM) supports the communication between distributed system components by facilitating message exchange [18]. It allows separate, uncoupled applications to communicate reliably through the sending and receiving of asynchronous messages. Messages are stored in a queue which can be accessed across the network. This messaging system is very flexible because it provides a high degree of anonymity between the message producer and the message consumer.

- Object middleware provides the abstraction of an object that is remote yet whose methods can be invoked just like those of an object in the same address space as the caller [17]. It makes object-oriented principles, such as object identification through references, inheritance, polymorphism, available for development of distributed systems [18]. The communication between objects (client object and server object) can be synchronous, deferred synchronous or asynchronous using thread policies. The Common Object Request Broker Architecture (CORBA) is a standard for distributed object computing.

Clearly, middleware is especially attractive for mobile and ad hoc networks applications. Such applications have to deal with recurring tasks like resource discovery, fault management, and communication handling. In [21], a survey of several research projects related to next generation middleware systems is presented.

An example for a middleware that was developed to support MANET applications is RUBI [4]. RUBI is a resource discovery framework for ubiquitous computing. It is based on the idea that there are significant similarities between resource discovery in a network and the operation of the routing algorithm since they both are concerned with the dissemination of information about the availability and efficiency of access to resources.

- In areas of low mobility, RUBI uses a discovery mechanism based on a proactive routing algorithm and resources are advertised throughout the network.
- In areas of high mobility, RUBI initiates a dynamic resource discovery using a reactive protocol and discovers resources as needed.

Furthermore, RUBI relies on the services provided by the link layer protocol such as 802.11 but does not require a routing protocol. RUBI is based on a routing protocol, so it can provide this functionality if needed.

Our middleware is different from RUBI in that it works independently of the routing protocol. Any MANET routing protocol can be used. Secondly, resource discovery in RUBI is based on announcing resources in areas of low mobility, but our middleware's is based on query and query reply resource discovery method similar to the ones used by overlay networks presented in the next section.

1.2.2 Overlay networks

An approach that has been suggested to facilitate application development over MANET is using overlay networks as viable application development architecture. An overlay network is basically a peer-to-peer virtual network that is logically built on top of the physical topology [1]. When a node attempts to join an overlay network, it has first to discover other overlay peers, and then make neighbor relations with some overlay peers.

The overlay neighbors might not necessarily be physical neighbors; they can be one hop away, or multiple hops away as shown in figure 1.2. Each overlay node can only communicate with the overlay network through its overlay neighbors.

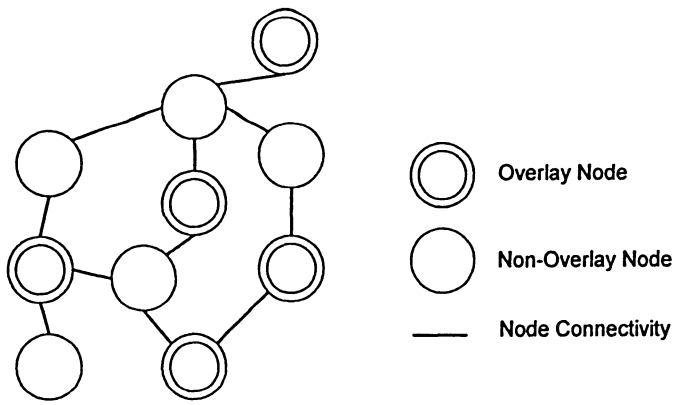

Figure 1.2: Overlay Network formed on top of the physical topology

Overlay network abstraction is especially attractive for MANET since it requires a subset of nodes to maintain application specific states while other nodes perform only underlying network functions such as routing. Thus, applications remain immune to the transience and instability of the nodes that are not engaged in its execution [5]. Overlay networks are categorized into two classes: the structured and unstructured networks.

1.2.2.1 Structured Overlay Networks

In a structured overlay network like CAN [10], Chord [11], Pastery [9] and Tapestry [12], nodes organize themselves in a structured graph and usually distributed Hash Tables DHTs are used as a substrate. Nodes are given NodeIDs which are populated from an ID space. Data is placed at random peers but at specific locations in order to make the search operation more efficient. Firstly, a hash function is used to generate keys from content (data). For instance, Chord uses SHA-1 hash function which takes a maximum of 2^{64} bits as input and generates a 160-bit output. The keys generated are a subset of the ID space from which NodeIDs are also distributed. The scheme of mapping keys to nodes depends on the design of the system. For example, in the Chord architecture, the node chosen to store the content of key K which belongs to the ID space is the one that

has a NodeID that is immediately greater than or equal to K (usually described as closest clockwise). Such node is called the successor of K in Chord's terminology. In Pastry, keys are mapped to the closest node (clockwise or counterclockwise). Tapestry maps a key to the node with the longest prefix match.

The graph of the overlay network is formed in the different structured overlay systems because each node needs to keep track of the node whose NodeID is immediately greater than itself, or smaller or even both. For example, in Chord, a node *n* keeps track of its successor (the node which has the smallest NodeID that is greater than *n*'s nodeID); therefore, nodes form a kind of a ring structure. This structure simplifies the overlay network management since the events of a new node joining the network or an old node leaving the network will have a complexity of $O(\log N)^2$ performance [8].

1.2.2.2 Unstructured Overlay Networks

In the unstructured overlay networks like Freenet [13], Gnutella [15], Gia [6] and Bittorrent [14], nodes are organized in random graphs. Searching for resources in this class of overlay networks can be centralized or distributed. Napster [16] and Bittorrent are examples for centralized-search distributed-download types of unstructured overlay networks. In bittorrent, a client looking for a file would send a query to a server, known as a tracker. The server would reply with the IP addresses and information needed of other clients that are currently downloading or uploading the file.

However, centralized search introduces scalability and single-point-of-failure problems. To solve these problems, distributed-search overlay architectures like Freenet and Gnutella and Gia were introduced. The earlier decentralized systems like Freenet and Gnutella used a TTL-controlled flooding paradigm to locate resources. Newer systems like Gia use flow-controlled random walks to route queries and locate resources.

1.2.2.2.1 Gia

Gia is a scalable Gnutella-like P2P unstructured overlay network that is designed for the internet. It introduces a lot of improvements over the design of Gnutella. Gia replaces Gnutella's flooding with random walks. A random walk is essentially a blind search in

that at each step a query is forwarded to a random node. Actually, the search protocol of Gia biases its walks towards the high-degree nodes because those nodes have pointers to a larger number of files and hence are more likely to have an answer that matches the query.

When a node first starts up in Gia, it uses a bootstrapping algorithm similar to the one used in Gnutella to locate other Gia nodes. Each Gia client maintains a host cache that consists of other Gia nodes (their IPs, port number and capacity). The capacity of each node is determined as a function of bandwidth, processing power, disk speed, etc.

To achieve topology adaptation, each node independently measures its level of satisfaction which is a quantity between 0 and 1 that represents how satisfied a node is with its current set of neighbors. A value of S=1 suggests that the node is fully satisfied. The node tries to gather more neighbors as long as its capacity allows a higher level of satisfaction.

To add a new neighbor, a node (say *X*) randomly selects a small number of candidate entries from those in its host cache that are not marked dead and are not already neighbors. From these randomly chosen entries, *X* selects the node with maximum capacity greater than its own capacity. If no such candidate entry exists, it selects one at random. Node *X* then initiates a three-way handshake to the selected neighbor, say *Y*.

During the handshake, each node makes a decision whether or not to accept the other node as a new neighbor based upon the capacities and degrees of its existing neighbors and the new node. In order to accept the new node, we may need to drop an existing neighbor.

In order to implement biased random walks, each node forwards a query to its high capacity neighbor as long as it has tokens from that neighbor. TTLs are used to control the duration of the random walks and book-keeping is used to avoid redundant paths. With book-keeping, each node generates a 16 bit random GUID and includes it to the

query msg. If a query with the same GUID arrives back at the node, it is forwarded to a different neighbor.

Finally, Gia improves the efficiency of the search protocol by using a one-hop replication scheme. Each Gia node actively maintains an index of the contents of each of its neighbors. The indices are exchanged between neighbors when they establish connections with each other and then periodically updated. When a node receives a query, it answers based on its own content or its neighbors' contents.

1.2.2.2.2 RAON

Resource Aware Overlay Network (RAON) [5] is a P2P system for MANET that performs query-forwarding decision taking into account link instability and power constraints. RAON is actually the adaptation of Gia network design for wireless ad hoc networks. Therefore, it incorporates all the design features of Gia, but modifies the biased random walk to incorporate dynamic factors in forwarding a query such as link instability and node power constraints.

RAON defines the capacity of a node as a static function or CPU speed and memory. Since bandwidth is irrelevant in wireless networks, it is removed from the definition of the capacity of a node. RAON introduces a ranking scheme which classifies a link to a node according to the power of the node and the link stability. Link stability can be obtained by measuring the RTT on the link. Further, a node also receives the residual power level of its neighbors. Based on energy and RTT, link coloring scheme function is performed as shown in table 1.1.

Table 1.1: Link Coloring Function [5]

Link Color	Indication	Condition
Green	Best	Power is high and RTT is low
Yellow	Average	Energy is medium and RTT is not high Or RTT is medium and Energy is not low
Red	Worst	Energy is low or RTT is high

As shown in table 1.1, a link is colored green when energy of the other end node is high and when the RTT is low. The other extreme color is red which is given to the link whose other end node has a low energy and high RTT. Therefore, the rank of green signifies a most efficient or best link, whereas a rank of red signifies a least efficient link. Because of the dynamics of the ad hoc network, these colors are evaluated periodically by sending probe messages to measure RTT and by exchanging power information among neighbors.

Routing in RAON uses a modified version of the biased random walk that was originally introduced in GIA. Modified biased random walk (MBR) algorithm, shown in figure 1.3, tries to avoid unstable links and low-energy nodes for a better performance. MBR also employs Gia's flow control mechanism to control the number of queries a node sends to a neighbor. It considers a neighbor node to be available if the node has token for the neighbor. It then selects an available number with the highest color and capacity. It gives color precedence over the capacity.

```
Algorithm MBR: Node x receives a query with qid and
determines the next hop for the query.

Let Col( i ) be the color of the link to neighbor i
Let Cap( i ) be the capacity of neighbor i

scolor = RED ;
scapacity = 0 ;
snode = x ;

For every neighbor i such that x has not forwarded query with qid to i before {
        If (Col( i ) > max_color) or ((Col( i ) == max_color) and (Cap( i ) >max_capacity))
        then {
                scolor = Col( i );
                 scapacity = Cap(i);
                 snode = i;
        }
        }
retum i ;
```

Figure 1.3 : Modified Biased Random Walk [5]

Finally, the flow control mechanism at a node can assign tokens to a neighbor proportional to the color of the link to the neighbor. For example, it can assign least

tokens to neighbors with RED link and more tokens to neighbors with green link. That is actually very similar to the flow control scheme adopted in Gia, in which nodes assign more tokens to neighbors with higher capacities.

1.3 Structured Vs. Unstructured Overlay Networks

It is important to first answer the question of which kind of overlay can perform better on the ad hoc environment. Structured overlays which are based on DHTs have proven their efficiency over the wired environments. Actually, there are increasing efforts to utilize structured overlays for wireless networks, such as [3].

However, building a DHT over MANET might not be the best option. The reason is that in a structured overlay, a node must hold indices for the files located at other nodes in the network regardless of its distance with them. If a node leaves the network or if a node is disconnected, which can be fairly assumed as a usual event in wireless environments, an expensive operation is invoked to relocate its file indices to another node. Frequent disruption of nodes of connectivity in MANET would require full redundancy or expensive relocation of file indices for a structured overlay network [5].

In unstructured overlay networks, the nodes do not form a logical graph and therefore, joining and leaving the network have a smaller computational complexity. Overlay routing can also be simplified in unstructured overlays so that it does not consume network's resources.

1.4 Objective

Gia's unstructured overlay network architecture was originally introduced to improve on the existing decentralized file sharing applications like Gnutella. RAON was introduced later on as an adaptation of Gia for the ad hoc networks, in which decentralization is required. A peer-to-peer overlay can provide a good substrate for creating large scale data- sharing, streaming and application-level multicast, and other distributed

applications because it provides important services like topology formation and resources discovery.

In addition to file sharing applications, overlay network abstraction can be used as a substrate for several upper layer applications such as VoIP, multicast and multimedia streaming, etc. However, in order to implement those applications on top of an overlay network, each one of these applications has to deal with similar issues regarding implementing the functionalities of the overlay network, such as joining the network, searching for resources and files, sending and receiving data, and leaving the network, which are tedious and recurring tasks to do.

Moreover, the concept of middleware as an intermediate layer between the ad hoc layer and the distributed applications is also desirable and encouraged. It gives the application developers great flexibility because it hides the lower layer implementation details. The use of middleware is also encouraged as a base for mobile application development, because it promotes portability and efficiency.

The objective of this thesis is to propose the design of a middleware that abstracts the functionalities of the unstructured overlay network. With such middleware, we combine the benefits of unstructured overlay networks in facilitating algorithm and application design and middleware in facilitating application development and implementation. Software development will be facilitated because developers will only have to deal with their applications related issues, without having to worry about the lower layer details of overlay network architecture.

Since the middleware targets ad hoc networks applications, the design of the middleware will take into consideration the fact that the mobile devices that will be used to form an ad hoc network will most likely by resource-constrained.

The rest of this thesis is organized as follows. In chapter 2, we present some related concepts, work and technologies that will influence our design. In chapter 3, we present

and discuss the design goals and requirements of unstructured overlay network middleware. In chapter 3, the design and implementation of the middleware is illustrated with Unified Modeling Language (UML) diagrams, and finally, in chapter 5, we validate our middleware by providing some examples on how different applications can be implemented on top of our middleware. We also evaluate it by showing a demonstration of how it works and provide its CPU and memory performance analysis.

2 Overview of Related Work and Technologies

This chapter is divided into two sections. The first section introduces the Java 2 Micro Edition (J2ME) platform and some of its useful libraries which influence the design of our middleware. The second section introduces an API for a structured overlay networks. Although our middleware targets the implementation of an unstructured overlay networks API, studying the structured networks counterpart API has been useful for our design. In section 5.8, we elaborate more on which methods in our middleware implement similar functions to some of the functions that the structured overlay API suggests.

2.1 Java 2 Platform, Micro Edition - J2ME

J2ME is a collection of Java technologies and specifications that can run on a consumer wireless device platform (like a PDA or cell phone). Essentially, J2ME was introduced to provide programmers with a programming language with the benefits of Java, such as portability and familiarity, which would run on a large number of mobile devices. In order to achieve this, J2ME is not simply a cut down version of Java, but is designed to be modular, so as to best fit onto the full spectrum of resource availability [22]. Figure 2.1 depicts the J2ME platform components and its relationship with other java platforms.

The Java ME platform is composed of three elements:

- A configuration provides the most basic set of libraries and virtual machine capabilities for a broad range of devices. Currently, J2ME provides two configurations, which are namely, Connected Limited Device Profile (CLDC), and Connected Device Profile (CDC). CLDC targets smaller devices like 16/32 bit processor mobile phones. It only requires from 128K to 512KB for J2ME environment and applications. On the other hand, CDC targets more capable devices like high-end PDAs or smart phones. Typically, these devices include a 32-bit microprocessor/controller and require about 2 MB of RAM and 2.5 MB of ROM for the Java application environment.

- A profile is a set of APIs that support a narrower range of devices. These APIs include a broad range of built-in network protocols, and extensive support for networked and offline applications that can be downloaded dynamically. For example, CLDC can be complemented by Mobile Information Device Profile (MIDP), where as CDC can be complemented by the Foundation Profile (FP), or Personal Profile (PP).
- An optional package is a set of technology-specific APIs. Examples for those APIs include Wireless Messaging API, Mobile Media API, Bluetooth API, etc.

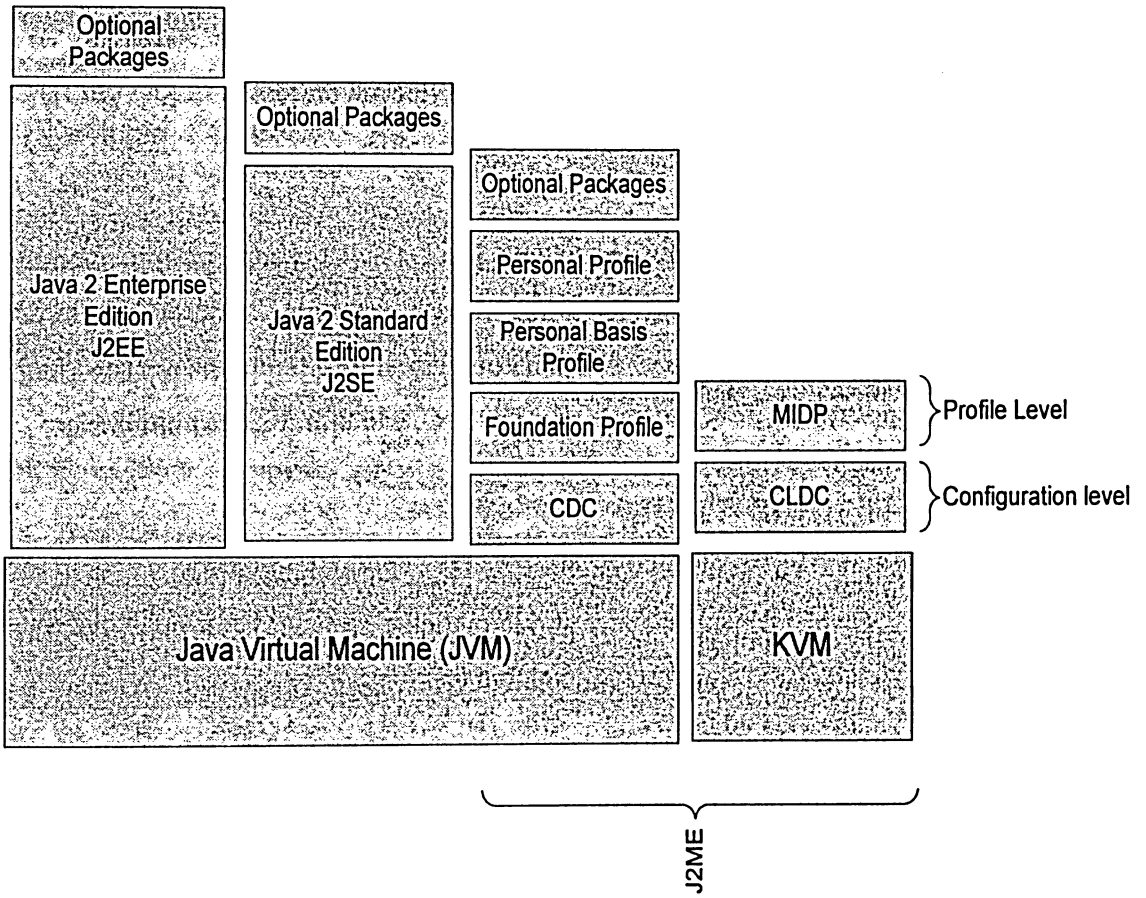

Figure 2.1 : J2ME Platform

The reason why Java code written according to J2ME platform can run on many devices is because of the Java Virtual Machine (JVM) concept. A JVM is actually a software that emulates the Central Processing Unit (CPU) of the physical computer which is responsible for running Java programs. Java programs are built to run on a virtual

machine, allowing them to run on any real machine that has a JVM. Most mobile device vendors support their products with their own implementations of JVM.

It is important to point out that Java code that complies with J2ME platform specification will only run on a device if and only if a vendor-supplied JVM is available on the device. Furthermore, CDC programs run on a different virtual machine than the one that CLDC programs run on. Sun Microsystems has developed two virtual machines for the J2ME platform. KVM is a virtual machine for CLDC, whereas CVM is a virtual machine for CDC. Therefore, CDC and CLDC programs do not interoperate. A java code written on top of CDC configuration will not run on a device if it only supports CLDC virtual machine. The opposite is true; a CLDC program will not run on top of a CDC virtual machine.

Because CDC configuration provides a richer set of APIs that can support implementing more advanced applications like distributed applications, the design of our unstructured overlay middleware will be based on CDC and Foundation Profile. In the next section, we present some concepts and APIs supported by CDC, which influenced our design.

2.1.1 Java Networking for Small Devices

Java provides a rich and comprehensive networking interface which supports many network protocols. Table 2.1 lists the most important networking APIs provided by the java.net package, which CDC supports for small devices.

Table 2.1 : Useful Classes from java.net package

Class	Description
java.net.InetAddress	A class that abstracts an Internet Protocol (IP) address
java.net.Socket	This class abstracts a client endpoint of a TCP communication channel
java.net.ServerSocket	This class abstracts a server endpoint of a TCP communication channel
java.net.MulticastSocket	The multicast datagram socket class is useful for sending and receiving IP multicast packets.

2.1.1.1 Peer-to-Peer Networking

The classes shown in table 2.1 provide methods and functions that are useful for distributed networking applications. For example, Socket and ServerSocket classes allow the implementation of client/server TCP application, or even peer-to-peer applications. The sequence diagram in figure 2.2 shows how a connection can be established between a client and a server. The server first binds a listening socket to a transport address (IP address and port number). Then the client connects to that transport address. After the server accepts the connection, the client and server can exchange data until one of them closes the connection.

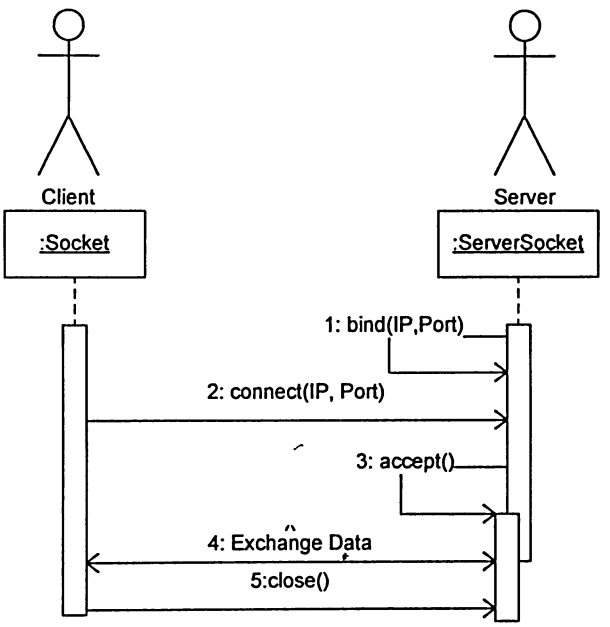

Figure 2.2 : Client-Server Communication Sequence Diagram

2.1.1.2 IP Multicast Networking

IP multicast is a network layer service, which is usually used with UDP as a transport layer protocol. The java.net library supports multicast transmission through instances of the java.net.MulticastSocket class, which extends the UDP java.net.DatagramSocket class with multicast-specific operations, such as joining and leaving a multicast group.

The MulticastSocket class provides an interface to the underlying Internet Group Management Protocol (IGMP) API which allows a host to join or leave specific multicast groups. The MulticastSocket.joinGroup(InetAddress grp) method is used to request subscription for the multicast group address specified as a parameter, grp. Once the joinGroup method has been invoked, the MulticastSocket periodically transmits IGMP membership reports, and responds to IGMP membership queries as specified by the IGMP protocol. Finally, applications can request to leave a group by invoking the MulticastSocket.leaveGroup(InetAddress grp) method.

2.1.2 Java Multitasking

Most modern operating systems support multitasking; they can run several processes or programs at the same time. In reality, if the operating system utilizes only one CPU, only one process can be executed at a time. Operating systems, therefore, solve this problem with scheduling. In scheduling, only one process is executed at any given time, while other processes wait for their turns. Similarly, The JVM allows an application to have multiple threads of execution running concurrently. Threads exist within a process and are sometimes called lightweight processes [23].

There are three main situations in which a software developer needs concurrency [24]:

- Non-blocking I/O: concurrency is important when the program performs a blocking call. For example, if a program sends and receives data over the network, it will need to perform a read operation from the network socket. A read operation blocks the program until there is data available. The data could be delayed in transit over the network. If the program is blocked, then it can not do anything else. It would be more efficient for the program to spawn a separate thread to perform the read operation from the network socket.
- Alarms and Timers: The program might need to set a timer so that it executes a specific task when the timer expires. In that case, concurrency solves that problem by allowing the programmer to simulate a timer in a separate thread.

- Independent Tasks: if the programmer wishes to execute two or more independent tasks simultaneously to improve the performance of the program, then he/she will have to resort to concurrency.

There are two ways to create a new thread of execution in java, either by extending (or inheriting from) the java.lang.Thread, and override its run() method, or by implementing the Runnable interface. In [23], a comprehensive tutorial on how to program with threads is provided.

2.2 An API for Structured Peer-to-Peer Overlays

In [7], the fundamental abstractions provided by structured overlays are highlighted. Based on these abstractions, a common API for structured P2P overlays is defined. As the first step, a *key-based routing API (KBR)* is defined. It represents basic (tier 0) capabilities that are common to all structured overlays. Actually, it is shown that KBR can be easily implemented by existing overlay protocols and that it allows the efficient implementation of higher level services and a wide range of applications. Thus, the KBR forms the common denominator of services provided by existing structured overlays.

Table 2.2 : Tier 1 Interfaces [7]

DHT	DOLR	CAST
put(key,data)	publish(objectId)	join(groupId)
remove(key)	unpublish(objectId)	leave(groupId)
value = get(key)	sendToObj(msg,objected,[n])	multicast(msg,groupId) anycast(msg,groupId)

A number of higher level (tier 1) abstractions that can be built upon the basic KBR have been identified, as shown in table 2.2. These abstractions include distributed hash tables (DHT), group anycast and multicast (CAST), and decentralized object location and routing (DOLR). Efforts to define common APIs for these services are currently underway.

The DHT abstraction provides the same functionality as a traditional hash table, by storing the mapping between a key and a value. This interface implements a simple store and retrieve functionality, where the value is always stored at the live overlay node(s) to which the key is mapped by the KBR layer. Values can be objects of any type.

The DOLR abstraction provides a decentralized directory service. Each object replica (or endpoint) has an *objectID* and may be placed anywhere within the system. Applications announce the presence of endpoints by *publishing* their locations. A client message addressed with a particular *objectID* will be delivered to a *nearby* endpoint with this name. Note that the underlying distributed directory can be implemented by annotating trees associated with each *objectID*; other implementations are possible. One might ask why DOLR is not implemented on top of a DHT, with data pointers stored as values; this is not possible because a DOLR routes messages to the nearest available endpoint—providing a locality property not supported by DHTs. An integral part of this process is the maintenance of the distributed directory during changes to the underlying nodes or links. Figure 2.3 depicts the basic abstractions and the KBR API.

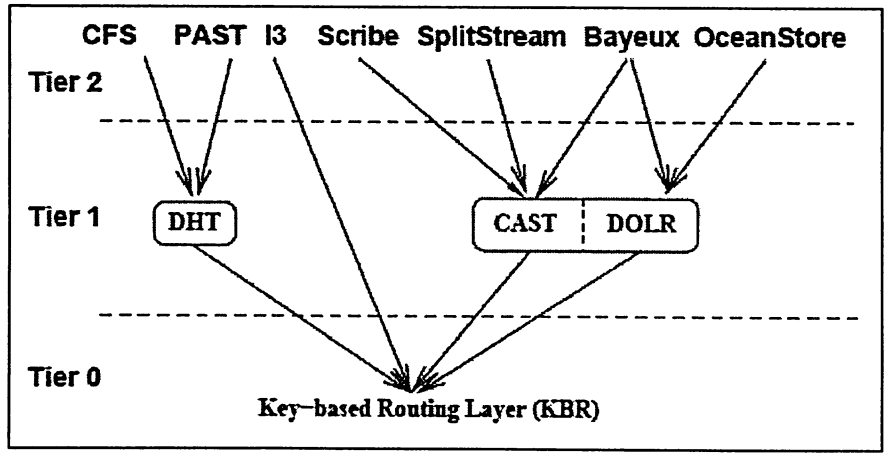

Figure 2.3 : Basic abstractions and APIs, including Tier 1 interfaces: distributed hash tables (DHT), decentralized object location and routing (DOLR), and group anycast and multicast (CAST) [7]

The CAST abstraction provides scalable group communication and coordination. Overlay nodes may join and leave a group, multicast messages to the group, or anycast a message to a member of the group. Because the group is represented as a tree,

membership management is decentralized. Thus, CAST can support large and highly dynamic groups. Moreover, if the overlay that provides the KBR service is proximity-aware, then multicast is efficient and anycast messages are delivered to a group member near the anycast originator.

Most applications and higher-level (tier 2) services use one or more of these abstractions. Some tier 2 systems, like i3, use the KBR directly [7]. Scribe [26] is an application level multicast application that can use CAST implementation at tier 1.

2.2.1 Key-based Routing API

The KBR API is composed of two types of functions, which are, the routing messages functions and the routing state access functions. In the next sections, we summarize the APIs data types and the API functions.

2.2.1.1 Data Types

A *key* is a 160-bit string. A *nodehandle* encapsulates the transport address and nodeID of a node. The transport address is the IP address and the port number. A *msg* type contains application data of arbitrary length.

2.2.1.2 Routing Messages

A parameter p will be denoted as $\rightarrow p$ if it is a read-only parameter and $\leftrightarrow p$ if it is read-write parameter. An ordered set p of objects of type *T* is denoted by *T[] p*. Table 2.3 lists the functions that are used to route messages.

Table 2.3 : Routing Messages Functions in KBR

Function	Return Type	Parameters	Description
route	void	key→ K, msg→ M, nodehandle→ hint	This operation forwards a message, *M*, towards the root of key *K*. The optional *hint* argument specifies a node that should be used as a first hop in routing the message. A good hint, e.g. one that refers to the key's current root, can

			result in the message being delivered in one hop; a bad hint adds at most one extra hop to the route. Either *K* or *hint* may be NULL, but not both. The operation provides a best-effort service: the message may be lost, duplicated, corrupted, or delayed indefinitely. The **route** operation delivers a message to the key's root. Applications process messages by executing code in upcalls which are invoked by the KBR routing system at nodes along a message's path and at its root. To permit event-driven implementations, upcall handlers must not block and should not perform long-running computations.
forward	void	key ↔ K msg ↔ M, nodehandle↔nextHopNode	This upcall is invoked at each node that forwards message *M*, including the source node, and the key's root node (before deliver is invoked). The upcall informs the application that message *M* with key *K* is about to be forwarded to *nextHopNode*. The application may modify the *M*, *K*, or *nextHopNode* parameters or terminate the message by setting *nextHopNode* to NULL. By modifying the *nextHopNode* argument the application can effectively override the default routing behavior.
deliver	void	key → K msg → M	This function is invoked on the node that is the root for key *K* upon the arrival of message *M*. The deliver upcall is provided as a convenience for applications.

2.2.1.3 Routing State Access

The API allows applications to access a node's routing state via the following calls. All of these operations are strictly local and involve no communication with other nodes. Applications may query the routing state to, for instance, obtain nodes that may be used by the forward upcall above as a next hop destination.

Some of the operations return information about a key's *r*-root. The *r*-root is a generalization of a key's root. A node is an *r*-root for a key if that node becomes the root for the key if all of the *i*-roots fail for $i<r$. The node may be the r-root for keys in one or more contiguous regions of the ID space. Table 2.4 lists the functions that help applications access the routing state information.

Table 2.4 : Routing State Access Functions

Function	Return Type	Parameters	Description
Locallookup	nodehandle[]	key → K, int → num boolean→safe	This call produces a list of nodes that can be used as next hops on a route towards key *K*, such that the resulting route satisfies the overlay protocol's bounds on the number of hops taken. If *safe* is true, the expected fraction of faulty nodes in the list is guaranteed to be no higher than the fraction of faulty nodes in the overlay; if false, the set may be chosen to optimize performance at the expense of a potentially higher fraction of faulty nodes.
neighborSet	nodehandle[]	int → num	This operation produces an unordered list of nodehandles that are neighbors of the local node in the ID space. Up to *num* node handles are returned.
replicaSet	nodehandle[]	key → k int → maxrank	This operation returns an ordered set of nodehandles on which replicas of the object with key *k* can be stored. The call returns nodes with a rank up to and including *max rank*. If *max rank* exceeds the implementation's maximum replica set size, then its maximum replica set is returned.
Update		nodehandle →n boolean→joined	This upcall is invoked to inform the application that node _ has either joined or left the neighbor set of the local node as that set would be returned by the neighborSet call.
range	boolean	Nodehandle→N rank → r key ↔ lkey key ← rkey	This operation provides information about ranges of keys for which the node N is currently an r-root. The operations returns *false* if the range could not be determined, *true* otherwise. It is an error to query the range of a node not present in the neighbor set as returned by the update upcall or the neighborSet call.

2.2.2 Examples of How Applications Can Use KBR

Table 2.2 shows application level (tier 1) abstractions. In the following section, we cite some examples from [7] that elaborate on how the KBR API can be used to implement tier 1 abstractions.

2.2.2.1 DHT Example

route(K,[PUT,V,S], NULL)
This call routes the message [PUT, value, S] to the root of the key K (i.e the node that stores the value that corresponds to the key K). The PUT function asks the receiving node to store the value V as a corresponding value for the key K. S is a nodehandler for the sender node.

route(key, [GET,S], NULL)
This call routes the message [GET, S] to the root of the key K. The GET function asks the receiving node to return to the node that has the nodehandler S, the value corresponding to the key K.

2.2.2.2 CAST Example

To implement the multicast/anycast functionality, a key is associated with each group and a key's root becomes the root of the group's multicast tree.

A node joins the group by routing a SUBSCRIBE message containing its nodehandler At the intermediate nodes which work as relays, when forward function is invoked, either the node is a member, so it terminates the SUBSCRIBE or it inserts its nodehandler and forwards the message towards the group's key root if it is interested in joining the group. Finally, a node may multicast or anycast a message to the group using group's key. For Multicast, the group key's root, upon receiving the message, forwards it to its children, but for anycast, the first node on the path of the group key's root forwards the message to its children.

2.2.2.3 DOLR Example

route (objectID,[PUBLISH, objectID, S], NULL)

This call routes the message [PUBLISH, objectID, S]. The purpose of the publish function is to announce the availability of an object. At each hop, an application stores the mapping between the object ID and the Nodehandler S of the sender.

The *unpublish* operation shown in table 2.2 walks through the same path and re moves mappings. The *sendToObj* operation shown in table 2.2 delivers a message to n nearby replicas of a named object. It begins by routing the message towards the object root using **route(objectId, [n, msg], NULL).** At each hop, the upcall handler searches for local object references matching objectId and sends a copy of the message directly to the closest n locations. If fewer than n pointers are found, the handler decrements n by the number of pointers found and forwards the original message towards objectID by again calling **route(objectId, [n, msg], NULL).**

3 Requirements and Design Issues

In this chapter we will discuss non-functional and functional requirements of a middleware for mobile applications. Following that we will discuss issues related to the middleware design.

3.1 Non-functional Requirements for Mobile Application Middleware

3.1.1 Portability

With the huge variety and diversity of available high end mobile devices and PDAs today in the market, it is fair to assume that different nodes in MANET would come with different software specifications and platforms. Therefore, a key non-functional requirement to achieve is portability.

After careful investigation, J2ME platform is found to be the best choice for software development that will help us meet our requirements. It provides a robust, flexible environment for applications running on a broad range of other embedded devices, such as mobile phones and PDAs. Applications based on Java ME specifications are portable because they are written once for a wide range of devices. The creators of J2ME platform, Sun Microsystems, claim that the Java ME platform is deployed on millions of devices and supported by leading tool vendors. For more information about J2ME platform, please refer to 2.1.

3.1.2 Efficiency

Since the middleware will most likely be used by high-end mobile devices, PDAs and laptops, and since energy and memory are considered limited resources for such devices, both time and space efficiency must be taken care of. The choice of implementing the middleware in the Java programming language also helps improving the efficiency requirements for the many reasons.

Firstly, Java takes care of memory because it provides garbage collection. In addition to executing code, the JVM is responsible for managing memory, like allocating memory from the operating system, and remove garbage objects. An object is considered garbage if there are no more references to the object after it has been created in the heap. Deallocating unused objects is very important especially for small devices so that other applications can utilize the deallocated memory.

Multithreading enables concurrent execution of several threads within the same program. Thus, it is a convenient way to decompose large programs into relatively independent smaller tasks and increase the overall efficiency [18]. It is an important requirement for efficient distributed peer-to-peer applications. Java has support for multithreading.

3.2 Functional Requirements for mobile Application Middleware

After studying the existing unstructured overlay networks such as Gia and RAON, the common functional requirements that the unstructured overlay middleware must provide, can be highlighted as follows:

1. An application that runs on top of the overlay middleware will be represented as a single overlay node. One or more nodes may be hosted by a single physical host. With the aid of the middleware, the application shall be able to learn its assigned overlay address from the middleware. The overlay address is generated by the middleware. It is composed of two parts, a transport address (IP and port) and a node ID. The overlay address must be unique across the overlay network.
2. The middleware shall perform a bootstrapping functionality. Before the node can join the network, it must first learn which nodes in the MANET are actually overlay peers. It must also announce to other nodes about its willingness to join the overlay.
3. The application shall be able to ask the middleware to join the overlay network. After the middleware learns which nodes in the MANET are overlay peers, it

starts communicating with them to establish a neighborhood relationships. It must also be able to monitor and manage its links to its neighbors.

4. The application should be able to specify to the middleware, the maximum number of overlay neighbors it can have at a time. The overlay node is considered to be satisfied if its number of neighbors is equal to the maximum number of neighbors it can have.
5. The middleware should provide the application with a means to route an application-layer message to an overlay peer, whether or not the address of the destination overlay peer is given.
6. The middleware should allow the application to search for resources such as files or people in the overlay network.
7. The middleware should allow the application to leave the network without disrupting the overlay network.

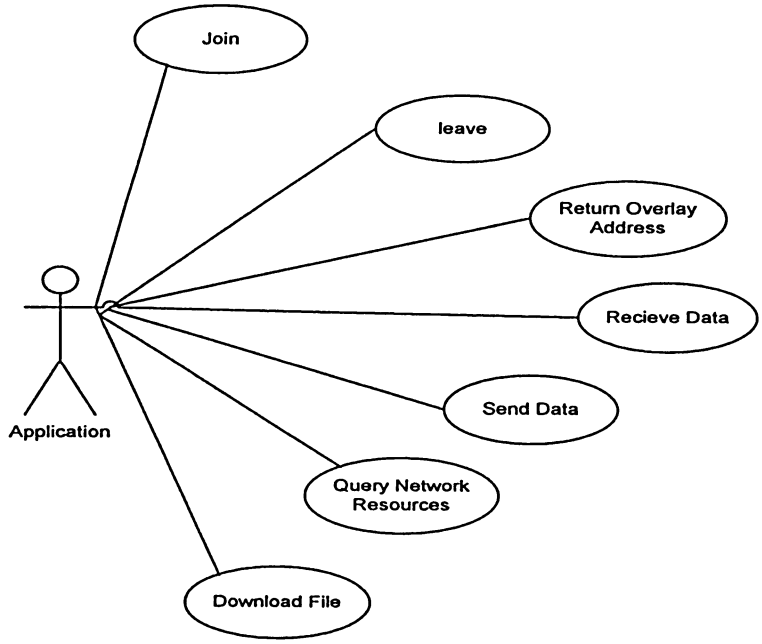

Figure 3.1 : System Requirements Use Case Diagram

Figure 3.1 shows the system requirements use case diagram. In the coming sections, each of these functional requirements is examined thoroughly and then the suggested implementation of each of these functional requirements is presented.

3.2.1 Assigning an Overlay Address to the Node

An overlay network is constructed by end hosts, each of which runs an overlay protocol to communicate with other overlay nodes. An overlay node represents an application that runs on top of the overlay middleware. Therefore, one or more overlay nodes can be hosted by a single physical host.

The first service that the middleware is expected to provide its user applications with is assigning them with overlay addresses. An overlay address could be a combination of the transport address and a node ID. The node ID can be assigned to the application randomly. The transport address can be the IP address of the physical host and a port address that an application chooses to set.

3.2.2 Bootstrapping Functionality

Bootstrapping is a key service that the application expects from our middleware. In order for the node to join the network and build the overlay topology, it must first learn about the existing overlay peers. The node must also announce its availability for other nodes in case it is willing to join the network. There are several ways by which bootstrapping can be implemented; all of these ways can be categorized as either centralized or distributed.

Centralized bootstrapping can be implemented with the aid of a server node that is known to all other nodes. For example, the server node can announce itself to other overlay peers by broadcasting a message to the overlay network. When a node wants to join the overlay network, it will first contact the server node. The server node will store the overlay address of the newly joining node and will send it the list of existing overlay nodes that are interested in making neighbors. The node can then select from the list the peers it wants to make neighbor relationships with.

Figure 3.2 depicts a centralized bootstrapping mechanism. When node E attempts to join the overlay network, nodes A, C and D are already overlay neighbors. Nodes A and C

are connected to each other through the ad hoc layer (Node B). Node D announces itself to the ad hoc network as a bootstrapping server periodically. When node E learns about D, it sends a request to join the overlay network to D. Then, D replies with the list of unsatisfied overlay peers, for example, A and D. Node C does not appear in the list because it is satisfied with the number of neighbors it has.

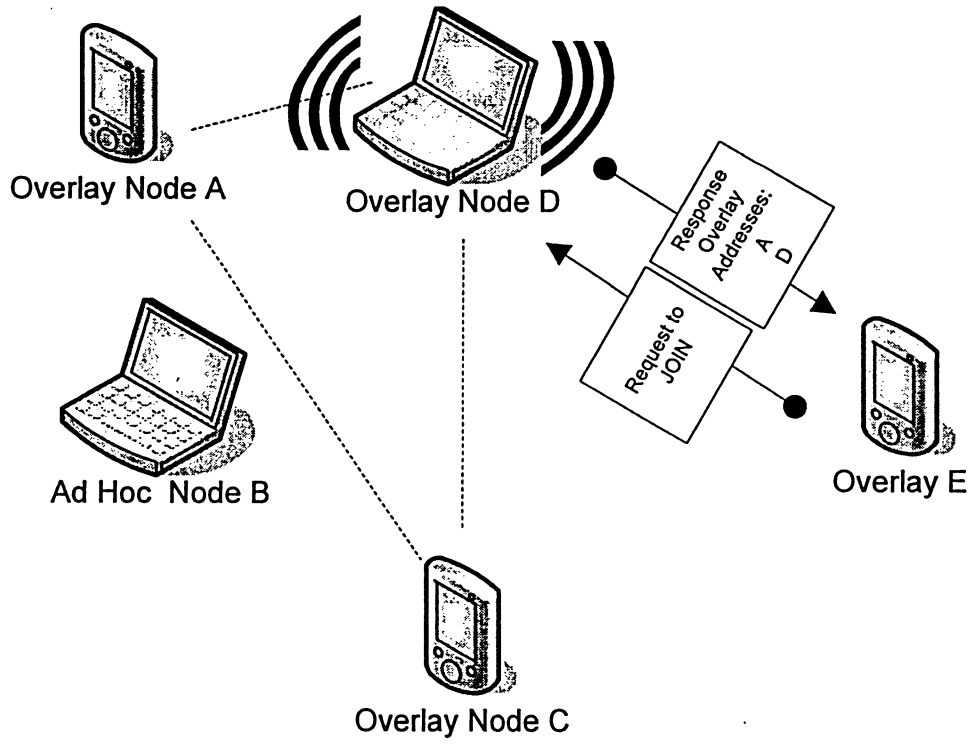

Figure 3.2 : Centralized Bootstrapping

A problem that rises with the centralized bootstrapping approach is how to select the server. If the server was chosen to be an overlay peer, then this will give birth to other problems. Firstly, how is the server node elected? It is true that algorithms similar to the ones used by OSPF [42] routers to elect a designated router can be used, but such algorithms will introduce further complexity which will exhaust small device resources like memory and processing power. Furthermore, such algorithms will require the nodes to exchange excessive traffic for the election process, which can consume the network bandwidth.

Another problem that is introduced with the centralized approach is the single-point-of-failure problem. If the server node fails, then newly arriving nodes can no longer join the network. Even if a backup server node exists in the network, there are no guarantees in ad hoc networks that a node will not disconnect at any time. Alternatively, the server node can be a fixed server on the internet, but with this approach, the advantage of the ad hoc network is lost because forming the overlay network would be conditioned by having internet accessibility for the ad hoc nodes.

A better approach is to implement a distributed bootstrapping. When a node wants to become a part of the overlay network, it can multicast to other overlay peers a joining message, that contains its overlay address, to let other nodes know about its willingness to make neighbors. If there are other unsatisfied overlay nodes in the network, then they can contact the newly joining node to make neighbor relations with it. When the newly joining node becomes satisfied, it will stop announcing itself to its overlay peers.

Figure 3.3 depicts how distributed bootstrapping takes place. Node E, simply announces itself to the overlay peers. In the meantime, unsatisfied overlay peers like Node A also continue to announce themselves periodically. After the unsatisfied nodes learn about each others addresses, they can communicate through ad hoc in order to establish neighbor relationships with each other in a pure distributed manner.

The advantage of this approach is twofold. Firstly, it is totally distributed, so nodes do not relay on a server to get them connected to the overlay. This in turn solves the single-point-of-failure problem that is inherent from relaying on servers.

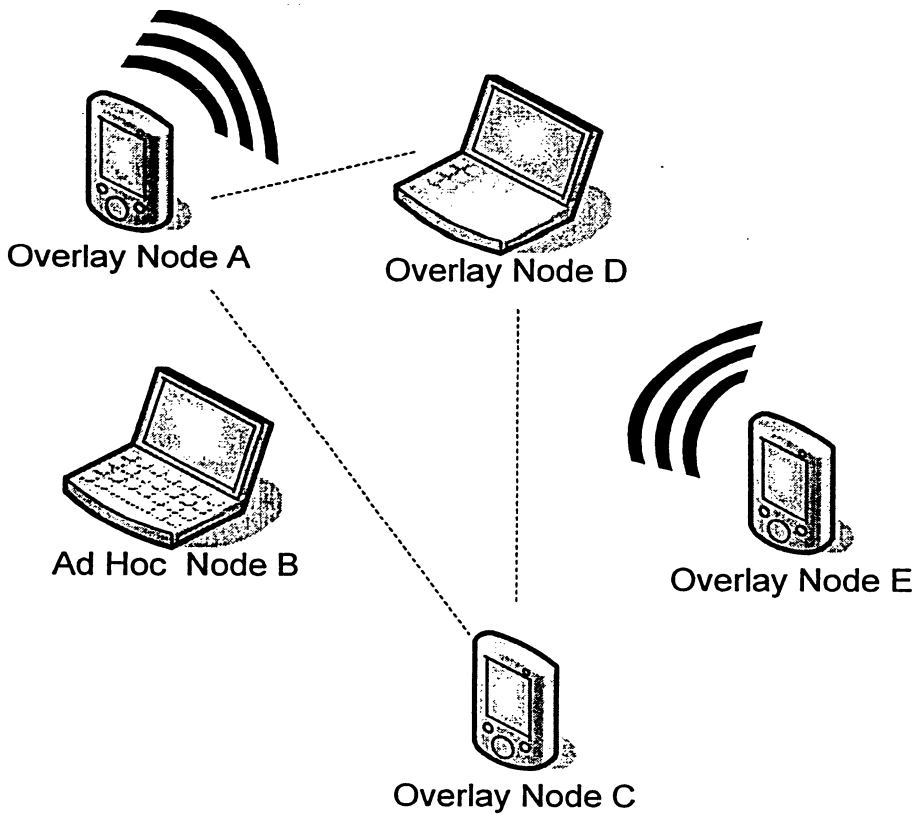

Figure 3.3 : Distributed Bootstrapping

Another advantage of this approach is that it achieves some degree of locality-awareness. That is, a node will most probably establish neighbor relations with its physically closest overlay peers first. The reason is that the physically closer overlay peers are most likely the first to hear the newly joining nodes' announcements. It is desirable for overlay neighbors to be also physical neighbors because the communication cost between overlay neighbors will be significantly decreased.

3.2.3 Establishing Neighbor Relations and Managing Connections

An overlay network, by definition, involves peer-to-peer communication. Peer-to-peer means that one peer acts as a client and contacts the second peer, which acts as a server that is waiting for incoming connections. When a node performs the distributed bootstrapping discussed in the previous section, the node will discover overlay peers and will make itself discovered by them. Thus, at any instance of time after bootstrapping,

the node can connect to overlay peers, or expect peers to connect to it simultaneously, for the purpose of neighbor relations establishment. The maximum number of neighbors an overlay node can have should be specified by the user application.

Therefore, to perform the neighbor establishment and connection management functionality, the node must have two components-- a client and a server. The client gets the addresses of the overlay peers that are discovered by the bootstrapping functionality. It can then connect to them using TCP connections. On the other hand, the server listens on a port for incoming connections from other peers.

The sequence diagram in figure 3.4 illustrates the connection establishment mechanism. Node B's client first connects to node A's server and requests neighbor establishment. Node A checks if it is unsatisfied and if node B is not a neighbor. Since both conditions are true, node A's server returns an Accept message to node B's server. Similarly, node B's client sends a request to node C's server. Before C's server sends a reply, node C's client sends a request message to B's server. Since node B is not node C's neighbor yet, node C's server sends an accept message to node B's client. Finally, node B's server sends a deny message to the pending request that was sent in order for node C's client to close the connection.

After the neighbors relations are established. It is important to have the client and server components monitor and manage the connections. Actually, both components have to perform similar tasks such as forwarding queries, or replying to overlay control messages. Therefore, we can group the client and server components to be under a connection manager subsystem.

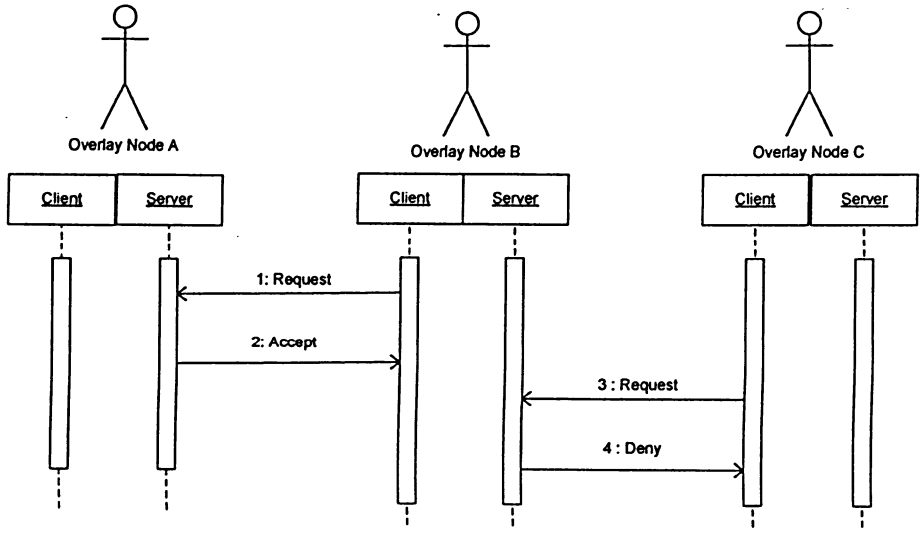

Figure 3.4 : Neighbor Establishment between Overlay Nodes

3.2.4 Resource Discovery

Most of the applications on overlay networks need to search for resources. In structured overlay networks, the nodes form an overlay logical graph which gives them a level of topology-awareness. Moreover, each node has an overlay routing table. This gives them a degree of knowledge of where a resource may be located in the overlay.

On the other hand, nodes in unstructured overlay networks do not have that knowledge. Many unstructured overlay system designs, such as Napster and Gnutella, suggested the use of central servers to search for queries. In addition to the scalability and the single-point-of-failure problems introduced with the use of central servers, the idea of a central server that has to be always available for an ad hoc network is pointless since ad hoc nodes are expected to disconnect at any time.

Therefore, searching for resources has to be done blindly. Flooding the network with search queries is an example of blind search, where the query is propagated to all neighbors within a certain radius [30]. However, flooding behavior is especially hostile to wireless ad hoc environments since it consumes a lot of bandwidth. Flooding can also

raise some scalability issues as the network grows. Finally, flooding-based techniques are effective for locating highly replicated items, but they are poorly suited for locating rare items [2].

Random walk is another well-known technique, which forwards a query message to a randomly chosen neighbor at each step until the item in question is found [30]. RAON suggests a blind search mechanism based on Modified biased random walks (MBR). Simulation results presented in [5] show that query success rate, when the number of peers is around 30, is around 90%. The success rate degrades to 80% when the number of peers increases to 50. In the worst case scenario, like when the network is highly dynamic, success rate varies from 70% to 60%. Hence, MBR algorithm can be implemented in the middleware to meet the routing and resource discovery requirements.

In order for the unstructured middleware to implement an MBR algorithm similar to RAON's, a Link Coloring Scheme (LCS) needs to be implemented as well. LCS is used to categorize the links to neighbors according to the power of the neighbor and the link stability. Table 1.1 shows how the link coloring function works. MBR actually bases its decision of routing on the colors that LCS assigns to the links to overlay neighbors.

From the discussion above, our unstructured middleware has to provide an MBR functionality that is also based on LCS functionality. Such functionalities will not interact directly with the user program, but they will be used in order to discover resources or route data for the user application.

3.2.4.1 Link Coloring Scheme (LCS)

Since LCS is a function of round-trip-time (RTT) of a link to a neighbor and the power level of that neighbor, our middleware must provide an LCS functionality to communicate with the neighbor in order to measure RTT and power. Since the overlay middleware establishes TCP connections with neighbors, the middleware can simply use the same connection to measure RTT and query the neighbor for its power level.

LCS can be implemented by two ways, which are namely, reactive or proactive. In the reactive approach, LCS colors the links to neighbors when the middleware needs to make an MBR decision. That is, it communicates with all of its neighbors to measure RTT and power only when it has data to route. On the other hand, in the proactive approach, the LCS component periodically measure RTT and power from the neighbor.

The advantage of the reactive approach is that it saves more bandwidth when compared to the proactive approach in which communication with the neighbors is required on a periodic basis. However, we could argue that a proactive approach is more efficient because it provides a faster response. MBR will not have to wait for the LCS component to communicate with all of its neighbors and wait for their responses. Rather, links will be already colored whenever the middleware needs to execute MBR.

Secondly, the middleware can be designed so that the application developer can set how often LCS is executed depending on the requirements of the user application. For example, if the application only routes data occasionally, the LCS proactive execution periods can be made larger than if the application needed to route data more often.

3.2.4.2 Overlay Routing

Routing is invoked in one of two states. The first state is the one in which the middleware needs to route data for the user application. The second state is when the middleware receives from a neighbor an overlay query or data that is destined for another overlay peer. In both cases, the middleware will route the overlay traffic to the next hop which is its best neighbor.

One issue we have to consider when we talk about routing is whether our middleware should have an overlay routing table or not. Obviously, when a node first joins the network, it has no clue where the resources it is looking for are located, but as it starts to search for them, it discovers the paths to some of the network resources.

For example, say that the application needs to send data periodically to a person in the network. Every time the application asks the middleware to send data to a specific person, MBR forms a random path depending on the colors of the links at each hop. Because of the randomness, there is a probability that data does not arrive to the destination every time.

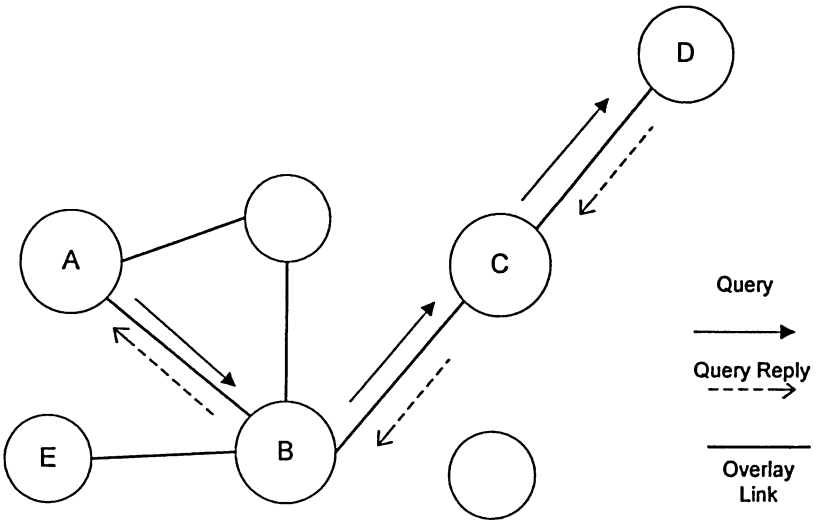

Figure 3.5 : Query reply over Overlay Network using the Reverse Path

Alternatively, the middleware can save in a cache the information of the next hop, the previous hop, and the status of the path. Saving the previous hop can help the destination send data to the source through the reverse path. When the source receives the response from the destination, it can set the status of the path to ready, meaning that the middleware can continue to use the same path to the destination. The advantage of this approach is that it increases the middleware's reliability and responsiveness.

Actually, saving the information can improve the responsiveness and reliability of the whole overlay network. In figure 3.5, node A is looking for a resource *X* in the overlay, so it sends out a query for the resource *X*. The query travels from A to B to C and finally to D where *X* is located. At each hop in the path, each node saves in a cache, the source address of the query, the previous hop, the queried resources *X*, and the next hop it used.

When the query arrives at D, it sends out a query reply through the reverse path to the source of the query, A. Each hop, upon receiving the query reply, sets the status of the path to true so that it can use it for the future for other *X* queries. Now say that node E is looking for *X*, if its query hits any node that participated in the path between A and D, E's query could be routed through the correct path.

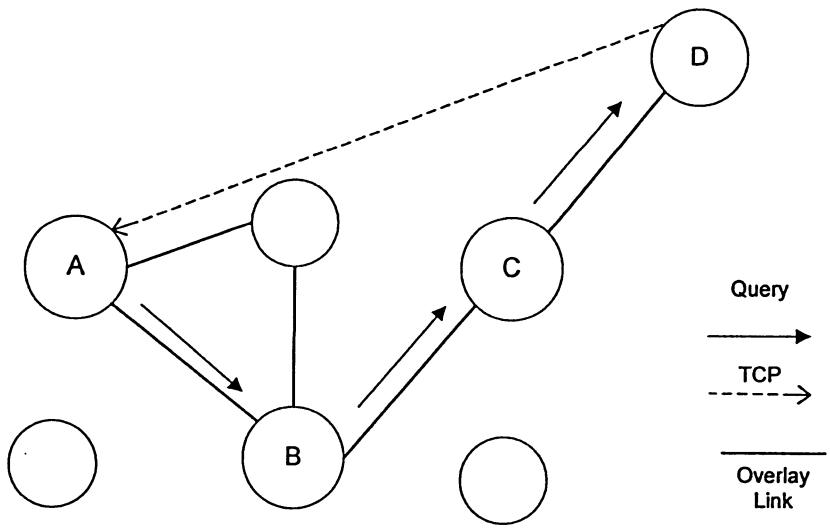

Figure 3.6 : Query Reply over TCP

The advantage of establishing the overlay path is that its formation is based on the best links at each hop in terms of power level of the next hop and its RTT. Secondly, some applications require the establishment of overlay paths. For example, an application level multicast application needs to establish overlay paths in order to build a multicast tree. Furthermore, an overlay reverse path is shown to perform better for some applications like file download in networks like CAON [44] because the reverse overlay path is less congested than the shortest TCP path between two overlay nodes.

The disadvantage of using it is that in ad hoc, intermediate nodes in the path are not guaranteed to be available all the time. For example, in figure 3.5, after B forwards the query to C, it gets disconnected. A problem occurs when the query reply arrives at C from D. C will have to route the query reply randomly towards the source, and the response might not reach the source.

For this reason, communication over overlay might be undesirable for the application after the destination is discovered. Another approach is shown in figure 3.6 , after the query arrives at the destination D using MBR, D can then rely on the TCP layer to connect to the source of the query, A to send it a query reply. To accommodate for different requirements that different applications need, our middleware should give the application the flexibility of choosing whether to use the reverse-overlay or TCP path after the resource has been discovered.

4 Design of an Unstructured Overlay Middleware for MANETs

In this chapter, we discuss the design of our unstructured overlay middleware. The purpose of our proposed design is to meet the functional and non-functional requirements that were discussed in the previous chapter. Since we will be developing in java, our design steps will be influenced by the Object-Oriented Design (OOD) concepts. In OOD, objects are abstractions of real-world or system entities. They allow us to express system functionality in terms of object services.

In order to carry out the system design process, we proceed with the following steps, as suggested in [25]:

- Define the context of our system.
- Design the system architecture.
- Identify the principal objects in the system and group them into packages.
- Develop design models and show scenarios of how objects interact with each other.

4.1 Middleware Context

The aim of our middleware is to implement the overlay layer, and provide an interface to a wide range of ad hoc network applications. Figure 4.1 depicts the system layer model. Our middleware resides between the application and the network layers. It implements the application-support layer and the overlay layer and it may depend on the ad hoc layer. For example, the ad hoc layer could provide an interface to retrieve some information needed for the operation of the overlay layer, such as RTT of a link and power level of an ad hoc peer, from the underlying network layer. The network layer can be any ad hoc routing protocol which supports multicast. A user application will be added on top of the application-support and will only interact with its simple interface.

The overlay layer carries the core of the middleware and implements the logic of the unstructured overlay network and its services, such as bootstrapping, establishing neighbors, routing, and overlay network resource discovery. In the next section, we identify the different subsystems in our middleware based on these services.

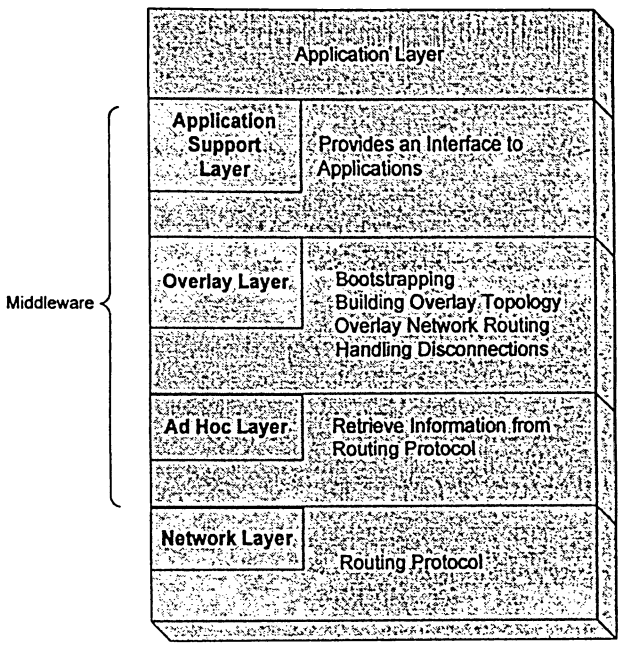

Figure 4.1 : System Layers Diagram

4.2 Unstructured Overlay Middleware Subsystems

At this step, we can map each service our middleware must provide to a subsystem. Figure 4.2 shows the possible subsystems the middleware can have. Application-support subsystem can be thought of as an implementation of the application support layer. Likewise, neighbor-establishment and bootstrapping subsystems correspond to the overlay layer. Services subsystem can be considered an implementation of a cross-layer that provides services to the other two layers.

In the next few sections, we further elaborate on the components of each subsystem. The components could be services that a subsystem provides, such as LCS, or they could be

abstractions of entities that exist in the overlay network, such as Node, Neighbor and Peer.

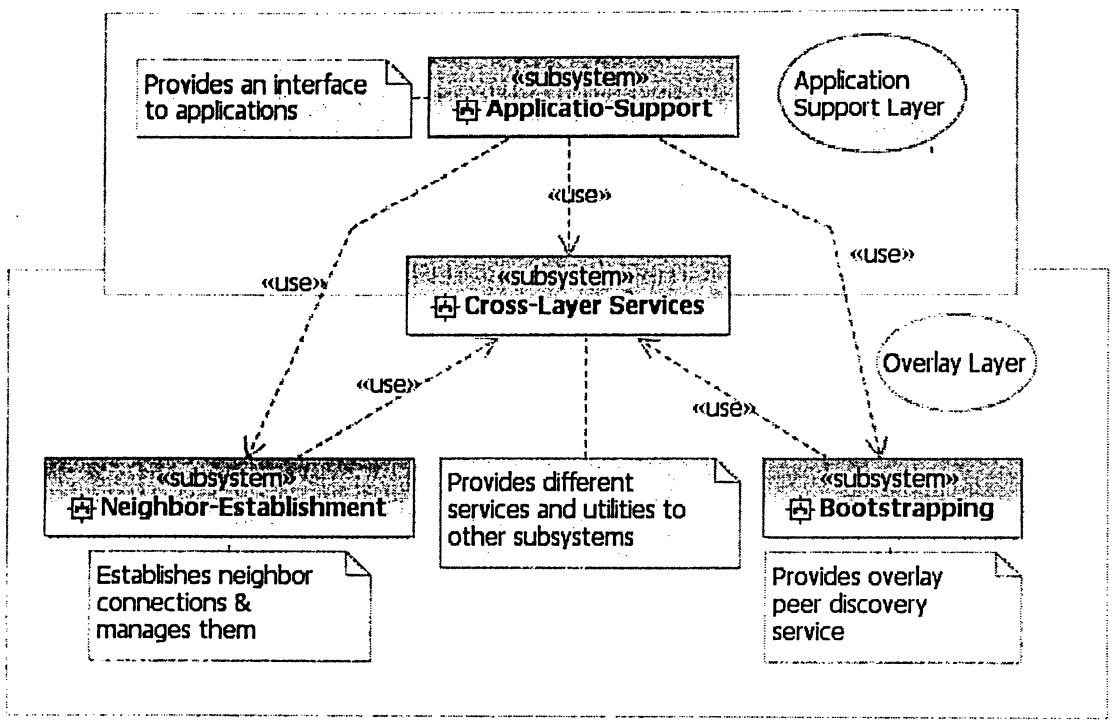

Figure 4.2 : Subsystems-Layer Diagram

4.2.1 Application-Support Subsystem

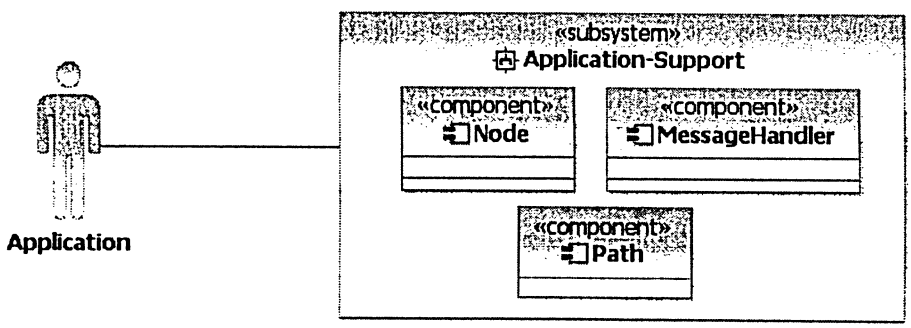

Figure 4.3 : Application-Support Subsystem

The application-support subsystem shown in figure 4.3 is the only subsystem in our middleware which interacts directly with the application. It provides an interface to the

applications for the underlying unstructured overlay layer. It is mainly composed of three components, which are namely, Node, MessageHandler and Path.

The Node component abstracts a node in an unstructured overlay network. It provides the functionalities which an overlay node is capable of, such as getting an overlay address, joining the network, searching and locating resources, such as files, returning a path to a resource on the network, and leaving the network. Typically, each application is represented by a single node. The Node component generates a random ID for the application. Therefore, any application in the overlay can be identified by its NodeID, and host IP address.

The MessageHandler component is provided by the user application and is registered with the Node component. It is invoked by the overlay layer when an application-level message is received. It actually gives the application a means of communicating with other overlay nodes at the application-level. MessageHandler is also triggered when the overlay layer needs to report errors and exceptions to the application.

The Path component abstracts a path to a resource on the overlay network. After the application issues a query to a file for example, the overlay layer proceeds with the resource discovery functionality. An overlay query packet is forwarded, and if the resource is found, a query reply would arrive. The application might want to use the path traversed by the query and query reply to send and receive data to and from the destination of the file using the same path. Furthermore, the application might wish to perform some functions on the path, such as refresh the path or measure its RTT. The Path component is returned to the application through the Node component.

4.2.2 Cross-Layer Services Subsystem

As its name implies, this subsystem, depicted in figure 4.4, provides different services to the other two layers. The services include providing the other layers with storage databases. Other useful utilities that the system needs is generating overlay packets, caching query routing information and allowing global preference configuration for the

whole system. In the next three sections, we further discuss the design of three important utilities that this subsystem must provide, which are namely, packet, databases, and LCS components.

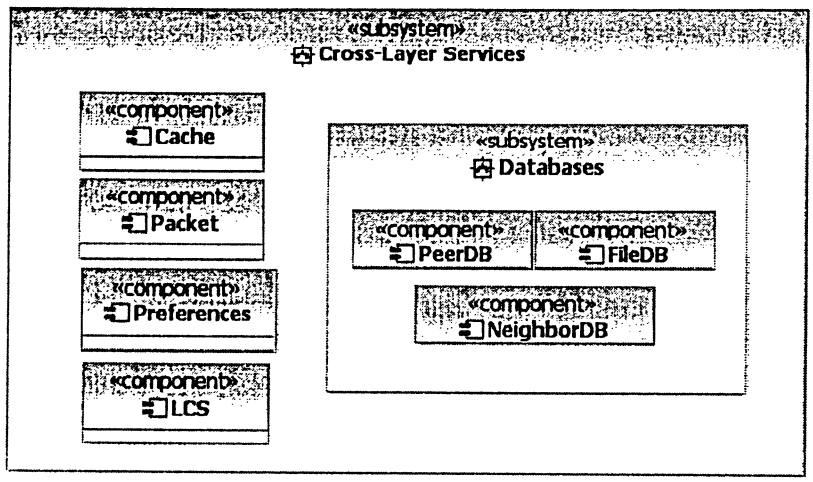

Figure 4.4 : Cross-Layer Services Subsystem

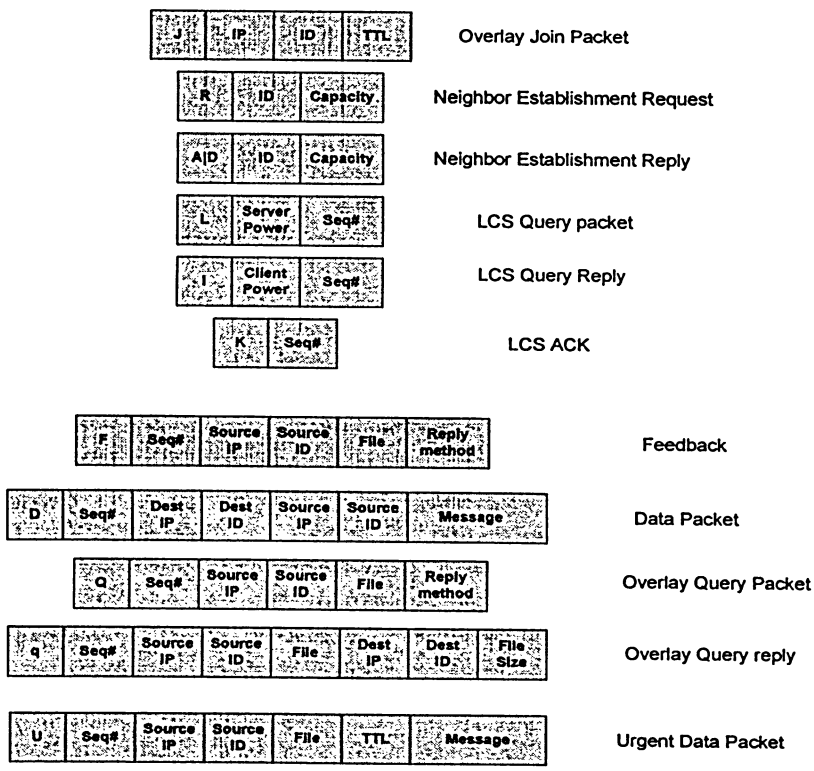

Figure 4.5 : Overlay Packets

4.2.2.1 Overlay-level Packets

In order for overlay nodes to communicate and understand each other, it is clear that a common language must be defined between them. The packets component is essential for creating overlay-level packets (shown in figure 4.5), which are:

- Overlay join packet: when an overlay node receives this multicast packet, it understands that the sender is looking for neighbors. Therefore, this packet only carries its sender's overlay address.
- Neighbor establishment request packet: when an overlay node receives this packet, it understands that the sender is willing to establish a neighbor relation. This packet carries the capacity of the physical host of the sender overlay node.
- Neighbor establishment reply packet: sent when a node receives a neighbor establishment request packet. The reply could either be "Accept" or "Deny". If the reply is accept, it will carry the capacity of the physical host of the sender.
- LCS query packet: Sent to a neighbor in order to measure the RTT of the link to that neighbor, and also to request the neighbor's power level. This packet contains the sender's host power level.
- LCS reply packet: sent to a neighbor as a reply to the LCS query packet. It carries the sender's host power level.
- LCS ACK packet: sent to a neighbor to acknowledge the receipt of an LCS reply.
- Resource query packet: sent when a node is looking for a resource in the overlay network. The query packet sets up a path between the source and destination.
- Resource reply packet: if the resource query packet arrives at a node that has the resource, the middleware generates a resource reply packet towards the source of the resource query packet. This packet sets up the reverse path from destination to source.
- Data packet: issued from source to destination after a path has been set up by query and query reply packets.
- Error feedback packet: issued by the overlay layer for path maintenance and to report errors and problems to the MessageHandler component.

- Urgent data: delivers a message to a node which has a specific file. Therefore, it does not require sending a prior query packet for path setup. It is also TTL controlled and not dropped when a loop occurs.

4.2.2.2 Databases

In addition to Node, there are other key entities in the overlay network, like Neighbor, Peer and File (or Resource). Peer abstracts an overlay network peer. An overlay peer is an ad hoc node that runs an instance of our middleware program and is not necessarily an overlay neighbor. Neighbor entity abstracts an overlay network neighbor. Finally, a File component abstracts a file or a resource in the overlay network. A Node can have zero or more File, Peer or Neighbor entities. Those entities are shown in figure 4.6.

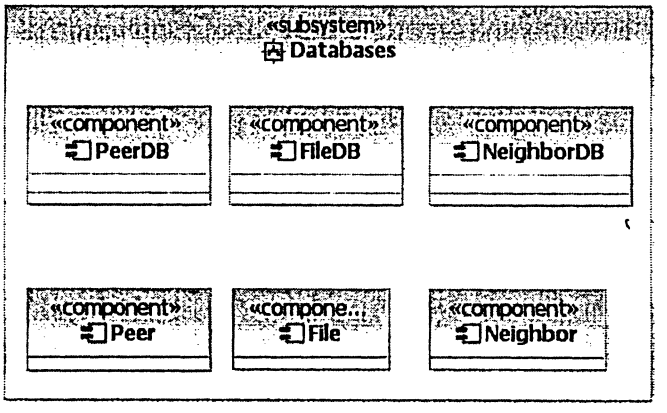

Figure 4.6 : Databases Subsystem

The databases subsystem is important to store, delete or search for peer, neighbor or file information. The databases subsystem also provides a means by which the application-support, the bootstrapping and the neighbor-establishment subsystems can communicate and cooperate with each other. For instance, after the bootstrapping subsystem discovers an overlay peer, it adds that peer information to the PeerDB component. Once the neighbor-establishment subsystem learns about the newly added peer, it retrieves its information from PeerDB, connects to it and sends it a neighbor establishment request packet. If the peer accepts, then its information will be added to the NeighborDB.

4.2.2.3 Link Coloring Scheme (LCS)

For every pair which has established neighbor relationship, one of them is considered to be the client side of the connection, while the other is considered the server side. Instead of having an LCS component associated with the two sides of the connection, a case in which the two parties will send LCS query packets and replies, our system is better off when only one of the two sends the LCS queries. In our system, triggering LCS is only associated with the server side of the connection.

Since the LCS component implements a functionality that is performed periodically, it will be associated with a timer, which when it expires, LCS component performs its task, and goes back to wait for the timer's expiry. In java, periodic operations can be implemented by extending (or inheriting from) the java.lang.TimerTask class, and implementing its run() method. The operation of LCS proceeds as follows:

- The LCS component at the server side saves the current time stamp, and then sends an LCS query packet, which contains the power level of its physical host.
- The client retrieves the power level from the LCS query and saves it. It also takes the current time stamp and sends out an LCS reply packet, which contains its own power level.
- When the server receives the LCS reply, it retrieves the power level of the client's host. It then calculates the RTT by subtracting the current time stamp from the previous one. Finally it sends an LCS ACK to the client.
- The client calculates the RTT of the link to the server as well by subtracting the time stamps.
- Both client and server save a number of consecutive RTTs so that they can measure the average RTT value, on which they base, along with the power, the color of the link.

4.2.3 Bootstrapping Subsystem

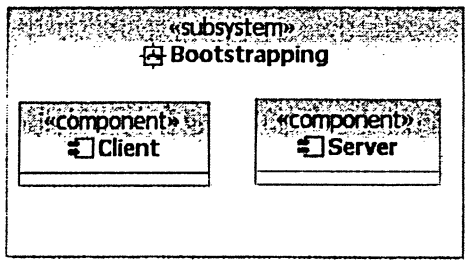

Figure 4.7 : Bootstrapping Subsystem

One of the functional requirements that our system has to support is a distributed bootstrapping service. Each node announces itself to its overlay peers, and it also receives their announcements. Therefore, the bootstrapping subsystem will have two components as shown in figure 4.7, a client and a server. The client is responsible for multicasting overlay join requests to its peers. The Server's duty is to listen for incoming overlay join requests sent by other overlay peers. Multicasting is supported in java through the java.net.MulticastSocket API, which we have discussed in section 2.1.1.

In order for the bootstrapping subsystem to operate properly, it needs support from the underlying ad hoc layer. The ad hoc layer should have a routing protocol with a multicast support. Alternatively, a small program can be added to non-overlay ad hoc nodes to propagate the multicast packets across the network.

4.2.4 Neighbor Establishment Subsystem

Because neighbor establishment is also distributed, the corresponding subsystem is composed of two components, a client and a server (as shown in figure 4.8). The client component is responsible for establishing the TCP connections, while the Server component is responsible for listening for incoming TCP connections. The Connection Manager component is in charge of monitoring the connections established by client or server and managing them.

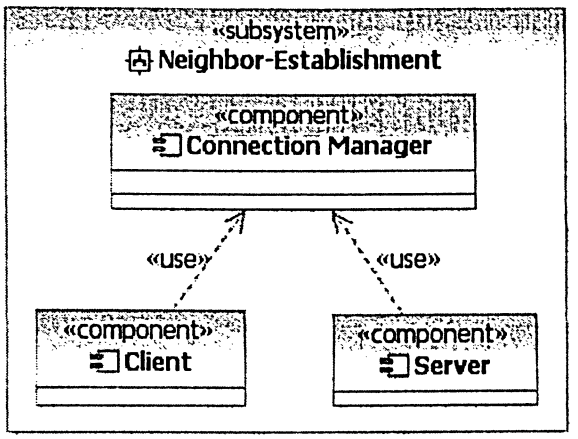

Figure 4.8 : Neighbor Establishment Subsystem

The Connection Manager component also performs overlay routing and handling of overlay packets. For example, if a resource query packet is received by the client or server, the connection manager either generates a resource query reply or it forwards the resource query to the next hop. This means that client and server components heavily rely on the connections manager, as it performs tasks for both of them.

Another important task that the connection manager is in charge of is setting up an overlay path between source and destination and maintaining the path. In the next section, we discuss the mechanism by which paths are established and maintained by the system.

4.2.4.1 Establishing and Maintaining an Overlay Path

Because the destination is unknown to the source when a query packet is sent, locating the destination starts with a blind search, which has to be done on top of overlay. On the other hand, when the query packet arrives at its destination, the source of the query will be known and the query reply can be sent over TCP or UDP. However, sending a reply over overlay has its advantages as well, so an application might need to keep all communication at the overlay level.

To achieve this requirement, our middleware should allow the source of the query or data packet to specify its preferable layer of communication. In our design, this information is indicated to the destination in the last field of the query packet (as shown in figure 4.5).

When the application needs to send data to a specific node in the overlay, it first has to establish an overlay path to that node. The application may not know which overlay node exactly it wants to communicate with, but it knows initially that its target node has a specific file, *X*. The source node then sends a query packet similar to the one shown in figure 4.5 with a sequence number. The packet then is handed to the overlay layer, which performs MBR to find out which neighbor is the next hop for the packet. Before sending out the packet, it saves its information in the cache, such as, the source IP and ID, the next hop calculated, the sequence number, and the current time stamp. Every hop, upon receiving the query, caches its information before sending it to the next hop.

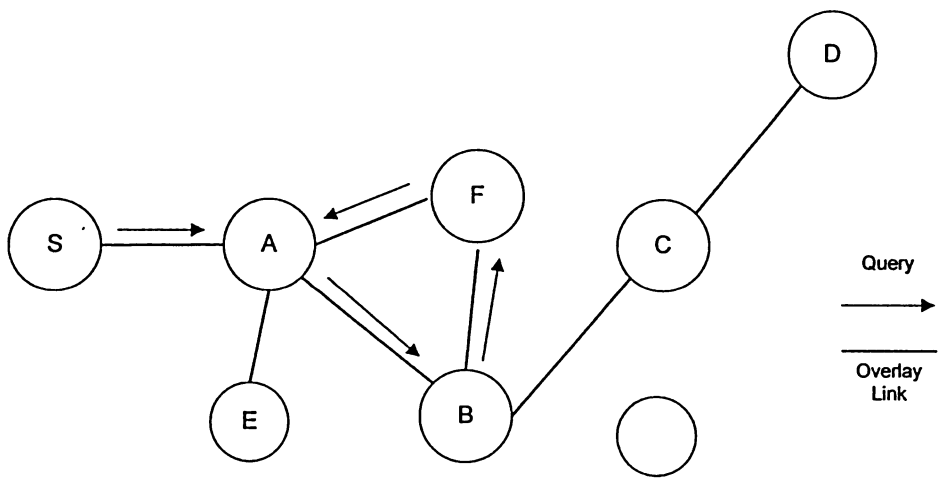

Figure 4.9 : Overlay Routing Loop

Because the next hop calculation is based on MBR, a loop such as the one shown in figure 4.9 could be created. To solve this problem, if a query packet hits a node which it has previously visited, the node simply forwards the query packet to a different neighbor. The node keeps track of all the neighbors to which it forwards the query. If the node has already sent the query to all of its neighbors, it drops the query. For example, in figure 4.9, A forwards the packet to E.

There are other scenarios in which the overlay path establishment could fail. A node could receive a query, but disconnect before it forwards it to the next hop. Another case is one in which a node fails in the path before a query reply arrives at the source. We solve these problems by allowing the application to indicate the number of transmissions it requires the middleware to perform for a specific query. The middleware issues a query packet and waits for a time duration, which depends on the size of the overlay network, before it sends the subsequent query packets. The time duration should typically be equal to the Round-Trip Time (RTT) of the longest loop-free path from the source to any other node. When a new query packet is generated, it will have a different sequence number.

When the query packet arrives at a node that has file *X*, the overlay layer generates a query reply packet and sends it to the source using the reply method desired by the source. If the reply method specified is TCP, the destination establishes a TCP connection to the source and sends it the query reply. If, however, the method required was the overlay reverse path, every node along the path uses its cache information to forward the query reply towards the source. They also record the destination information.

Only when the query reply packet arrives can the source exchange data with the destination. Any end can issue a data packet to the other end of the path. Data packets refresh the path, as each node in the path takes a new time stamp whenever the data packet traverses it. When the data packet arrives at any end, its message part is passed to the MessageHandler component, which is implemented by the application and registered with the middleware. If data packets exchange stops, the cache entry of the path expires and is no longer useable.

A problem occurs if a node in the intermediate path disconnects. For example, in figure 4.10, node S sent a data packet to D. Before D's reply reaches S via the reverse path, node B disconnects. The problem is that not only the path to S is lost, but also other nodes in the path, such as A, still have stale information of the broken path until the information expires. This means that if S keeps sending data packets, it will refresh the invalid information stored at A's cache even though they will not be delivered correctly

to D. We solve these problems by sending an error feedback packet from the node prior to the disconnected node towards the sender of the data packet.

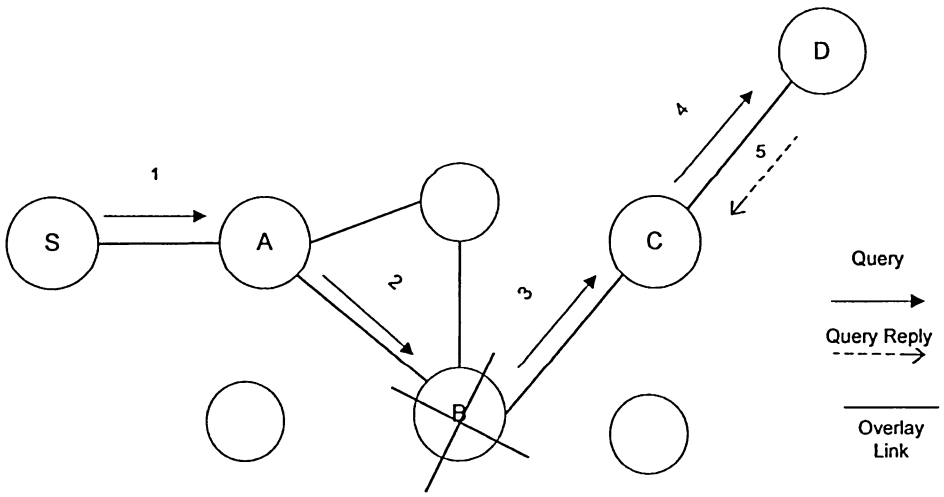

Figure 4.10 : Intermediate Node Disconnects

For example, when C discovers the failure of B, it issues an error feedback packet towards D that removes the stale cached routes from the nodes traversed. When D receives the feedback packet, it understands that the initial established path no longer works, a situation in which S has to send a new query packet to establish a new overlay path.

Feedbacks are important because they allow the system to recover in case paths fail. They are also important because they involve the application in solving the problem. A feedback always causes the MessageHandler component to be invoked.

4.3 System Architecture and Subsystems Interaction

The diagram in figure 4.11 depicts the overall system architecture which is composed of all the subsystems we discussed in the previous section. The diagram shows how the components of different subsystems interact with each other. The application only interacts with Node, MessageHandler and Path components. Node acts as an intermediary

between the application and the underlying subsystems. When the application asks the Node to join, the Node allocates memory for the databases and then it starts the bootstrapping and the Neighbor establishment subsystems.

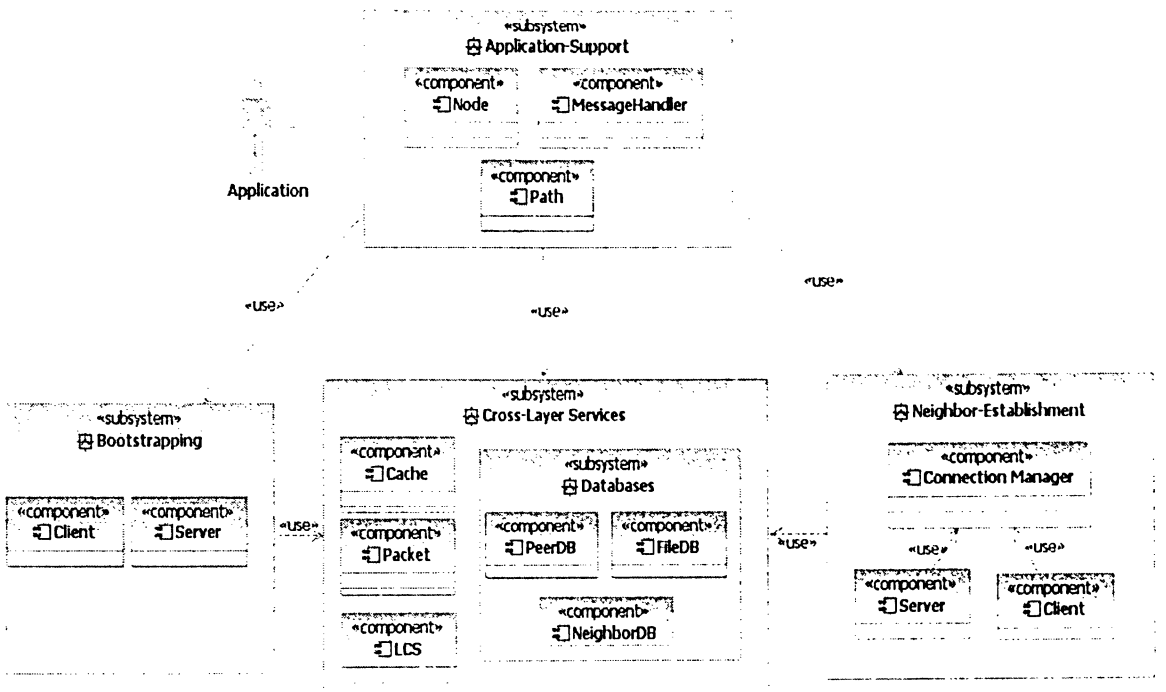

Figure 4.11 : System Architecture

When the application needs to search for a file in the overlay network, the Node component generates a resource query packet and passes it to the Connection Manager component to forward it. The Connection Manager also performs routing when Client and Server receive resource queries from their overlay connections. Finally, both bootstrapping and neighbor-establishment subsystems will use the databases subsystem to store the overlay peer and neighbor information. Node also interacts with FileDB to retrieve information about the queried files and resources. When a resource is found, its destination is updated in FileDB. Then, the application can retrieve the discovered Path from the Node component.

4.4 Object Identification

Having the overall system architecture diagram deduced in the previous section, object identification process can be carried out easily. We clearly need to associate an object with each component or entity, which provides a service, in our system. Firstly, the middleware objects are divided into three packages as shown in figure 4.12.

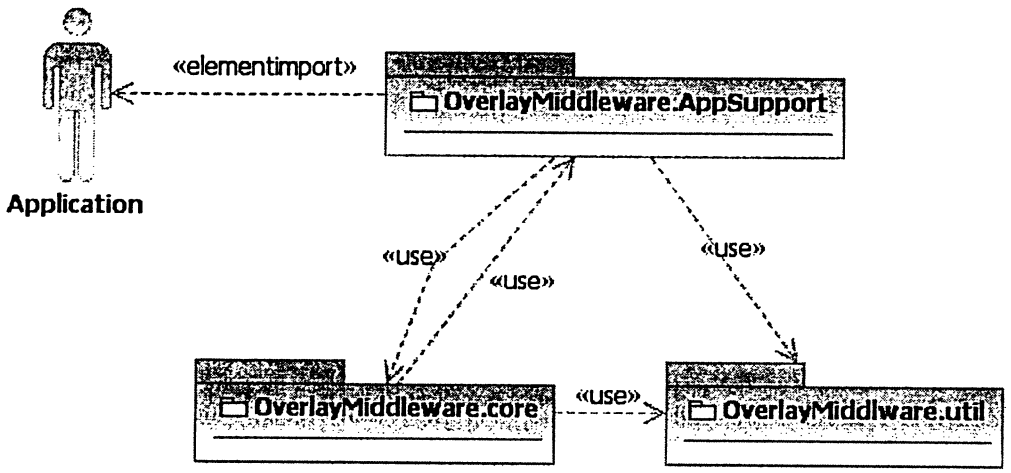

Figure 4.12 : System Package-Use Case Diagram

OverlayMiddleware.AppSupport implements the application-support layer of our middleware and contains object classes that provide an interface to the user application. OverlayMiddlewar.core implements the overlay layer and contains object classes that implement the bulk of the overlay layer. It combines the functionalities of the bootstrapping and the neighbor-establishment subsystems. Finally, OverlayMiddleware.util contains object classes that provide utilities to the other two packages.

There are three classes in the OverlayMiddleware.AppSupport, shown in figure 4.13, which are namely, Node, MessageHandler and Path. A Node object abstracts a node in an overlay network. It provides applications with a list of methods such as Join, Leave, Query, Download, ReturnPath and SendUrgentData. When an application instantiates a new Node object, it must specify the maximum number of neighbors it can have at a time. This value is passed by Node to the overlay layer. A Node object has a unique

NodeID which is composed of the hosts IP address and a higher level overlay ID. The node object also keeps track of the available overlay peers/ neighbors which are discovered by the overlay layer.

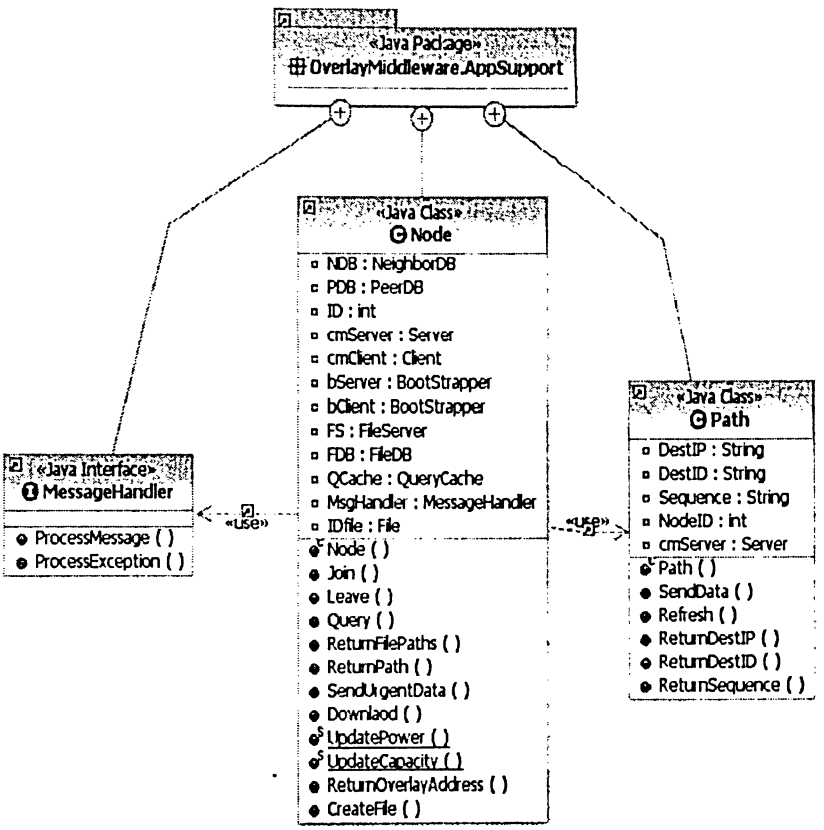

Figure 4.13 : OverlayMiddleware.AppSupport Package

Applications launch search for a resource through the Query method of the Node object. ReturnPath() returns a Path object to the application. A Path object is an abstraction of an overlay path from the node where the query is launched to the node where the sought after resource is discovered. It provides a set of functions that can be performed on the overlay path such as SendData(), Refresh(), and other methods to retrieve the sequence of the path and retrieve destination overlay address. An application can use the Path object to utilize the overlay path for application-specific purposes. For example, a VOIP application can use the overlay path for SIP signaling, or Application-Level Multicast (ALM) application can use the overlay path for multicast delivery etc.

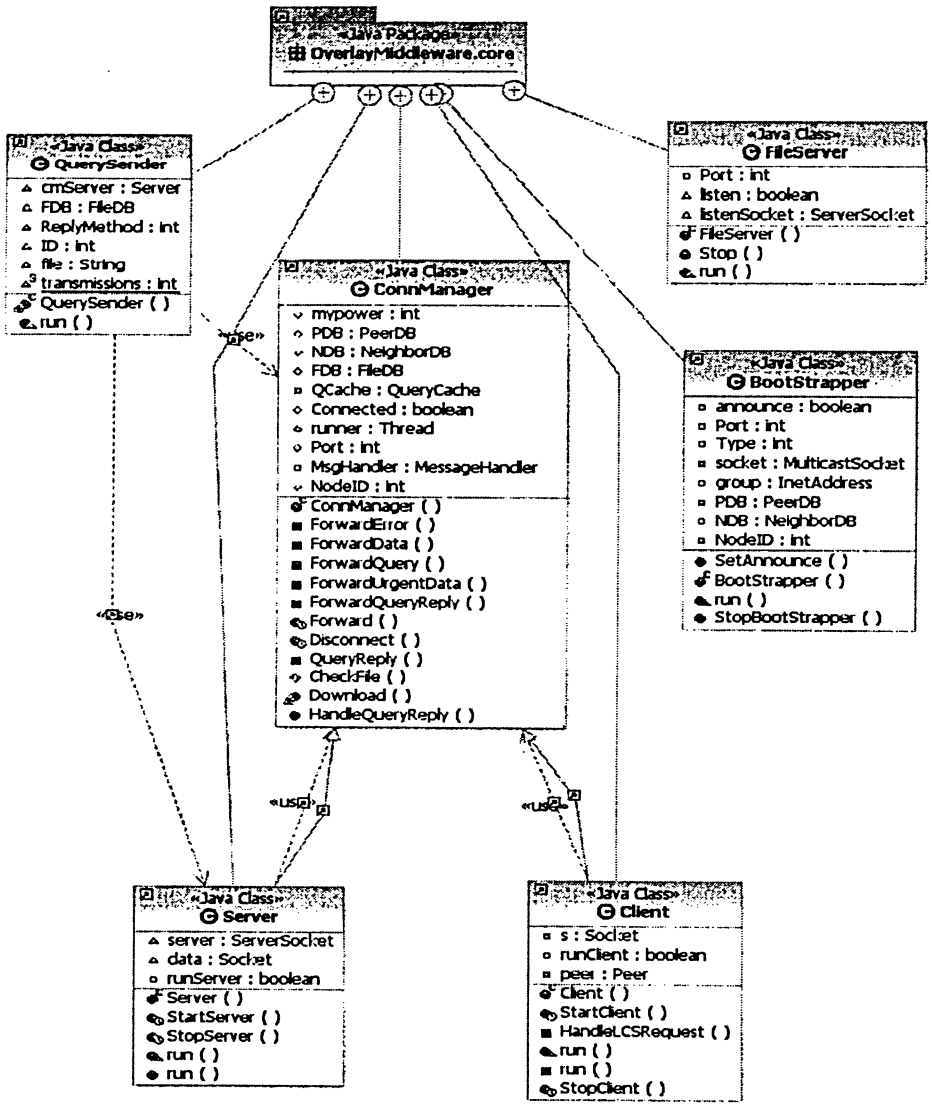

Figure 4.14 : OverlayMiddleware.core Package

The MessageHandler interface has two methods and they are to be implemented by the user and registered with Node. One method is ProcessMessage(), which is invoked when application-level data is to be delivered to the application. The second method is ProcessException(), which is called when the overlay layer reports an error to the application, such as reporting about a lost path when an error feedback packet is received. The main significance of the MessageHandler object is to enable application-level communication.

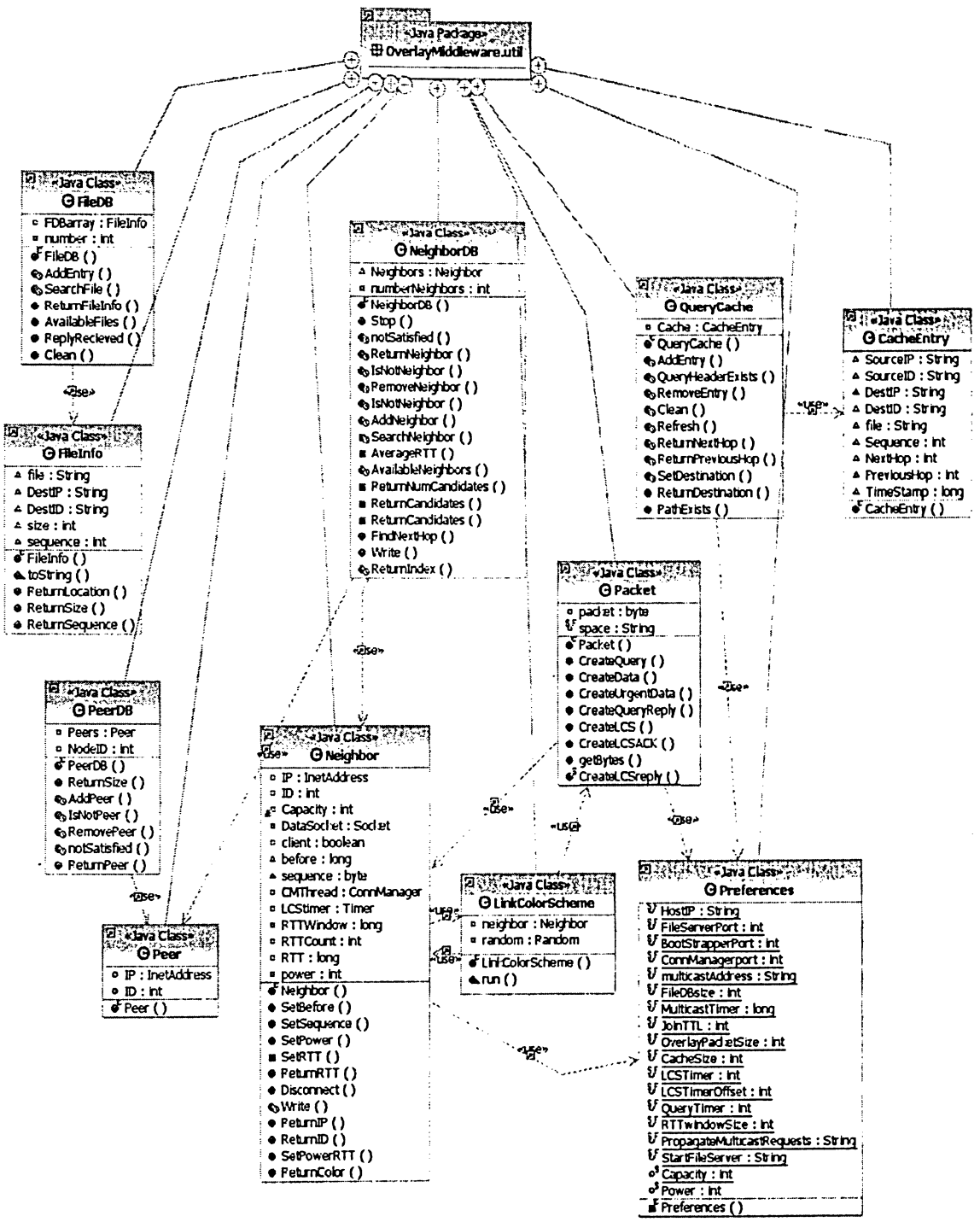

Figure 4.15 : OverlayMiddleware.util Package

OverlayMiddleware.core package, shown in figure 4.14, is composed of six object classes, Client, Server, ConnManager, FileServer, Bootstrapper and QuerySender. The first two classes use the java.net.Socket and java.net.Serversocket API to establish TCP connections. ConnManager provides connection management methods and functions such as routing and overlay packet handling. Since both Client and Server need these functionalities for their connections, they both extend (or inherit from) ConnManager.

In order for the Bootstrapper to work properly, Node class instantiates two Bootstrapper objects and sets the Bootstrapper.Type value so that one of them acts as a client and the other acts as a server. The server Bootstrapper object uses java.net.MulticastSocket API to join a multicast group and capture Overlay join packets sent by the client Bootstrapper. The client Bootstrapper keeps multicasting Overlay joins until the number of neighbor connections the node has is equal to the Node.Max_Neighbors value, specified by the application to the instance of Node it uses.

All of the five classes in OverlayMiddleware.core package extend java.lang.Thread class and override its run() method. This allows them to run in parallel each on a separate thread. QuerySender extends TimerTask class because it performs a periodic task of sending a query packet a number of transmissions if a query reply is not heard.

OverlayMiddleware.util whose class diagram is shown in figure 4.15, provides utilities that the other packages need to function properly. For example, it provides four types of data structures NeighborDB, FileDB, PeerDB and QueryCache. Each one of these data structures is used to add, delete, retrieve and search for Neighbor, FileInfo, Peer and CacheEntry objects, respectively. It also provides LCS and packet generation services.

A Neighbor object abstracts an overlay neighbor; it holds the state of an overlay neighbor such as, its overlay address (NodeID and IP), power level, RTT, and color of its link. A Neighbor object is also associated with a LinkColorScheme object, which inherits from java.lang.TimerTask and implements its run() method to send LCS Request packets

periodically. Neighbor object also provides the Write() method which allows other threads to send packets to a neighbor.

As its name implies, the Preferences class is used to set the application preferences. For example, the developer can set the ports that the middleware operates on. The developer can also set the query cache size, the different timers that the middleware uses, such as the MulticastTimer attribute, which defines in milliseconds how often the client Bootstrapper object sends out overlay join packets. Another timer that can be set in Preferences class is the LCSTimer attribute, which defines in milliseconds how often LCS query packets are sent.

4.5 Design Models and Scenarios

In this section, we discuss in details how the system objects interact with each other and how nodes interact with other nodes over the overlay network. We use one of the most expressive dynamic models that documents, for each mode of interaction, the sequence of object interactions that take place in a sequence model. We will also adopt a specific notation to refer to instances of objects. For example, *:ClassName* is an object that is an instance of the class ClassName.

In order for an application to utilize the services provided by our middleware, it will only deal with the simple API provided by the OverlayMiddleware.AppSupport package. Figure 4.16 illustrates the interaction between the application main object and our middleware's Node and Path objects.

When the application instantiates a Node object, *:Node*, it passes to it a Max_Neighbors parameter, which is an integer value that represents the maximum number of neighbors a node can have at a time. *:Node* passes this value to the overlay layer to restrict the number of neighbors it can have. The application can then ask *:Node* to join the network, return its assigned overlay address and query a resource (messages 2-5). The application can also ask the Node object to return the discovered path (message 6-7). The application can use the Path object to send data or download the file (message 8-9). Finally, when the

application has no more interest in being connected to the overlay network, it can ask Node to leave the network (message 10)

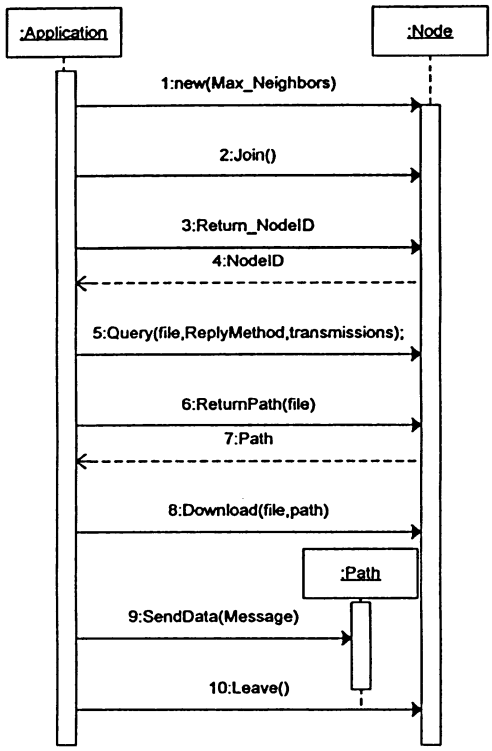

Figure 4.16 : Application-Node Sequence Diagram

The sequence diagram in figure 4.17 shows the interaction between *:Node*, *:Bootstrapper*, *:Server*, and *:Client*. *:Node* instantiates each one of these objects and starts them running, each on a separate thread, so that they can run in parallel. When *:Bootstrapper* starts working, it discovers the overlay peer B, and it adds it to *:PeerDB* (message 5). *:Client* requests the peer from *:PeerDB* with the ReturnPeer() method (message 6). *:PeerDB* then returns a Peer object to *:Client*. A Peer object holds the overlay address of the peer B.

Finally, *:Client* of node A establishes a TCP connection to Node B on a port where there is a *:Server* waiting for the incoming connection. Once the TCP connection is made, the *:Client* sends a neighbor establishment request to *:Server* at node B. If node B is willing to have more neighbors, it sends a neighbor establishment reply that indicates its

acceptance. Then both *:Server* and *:Client* objects at both ends create a new Neighbor object and add it *:NeighborDB*. The peer entry for node B in *:PeerDB* is then deleted (message 12).

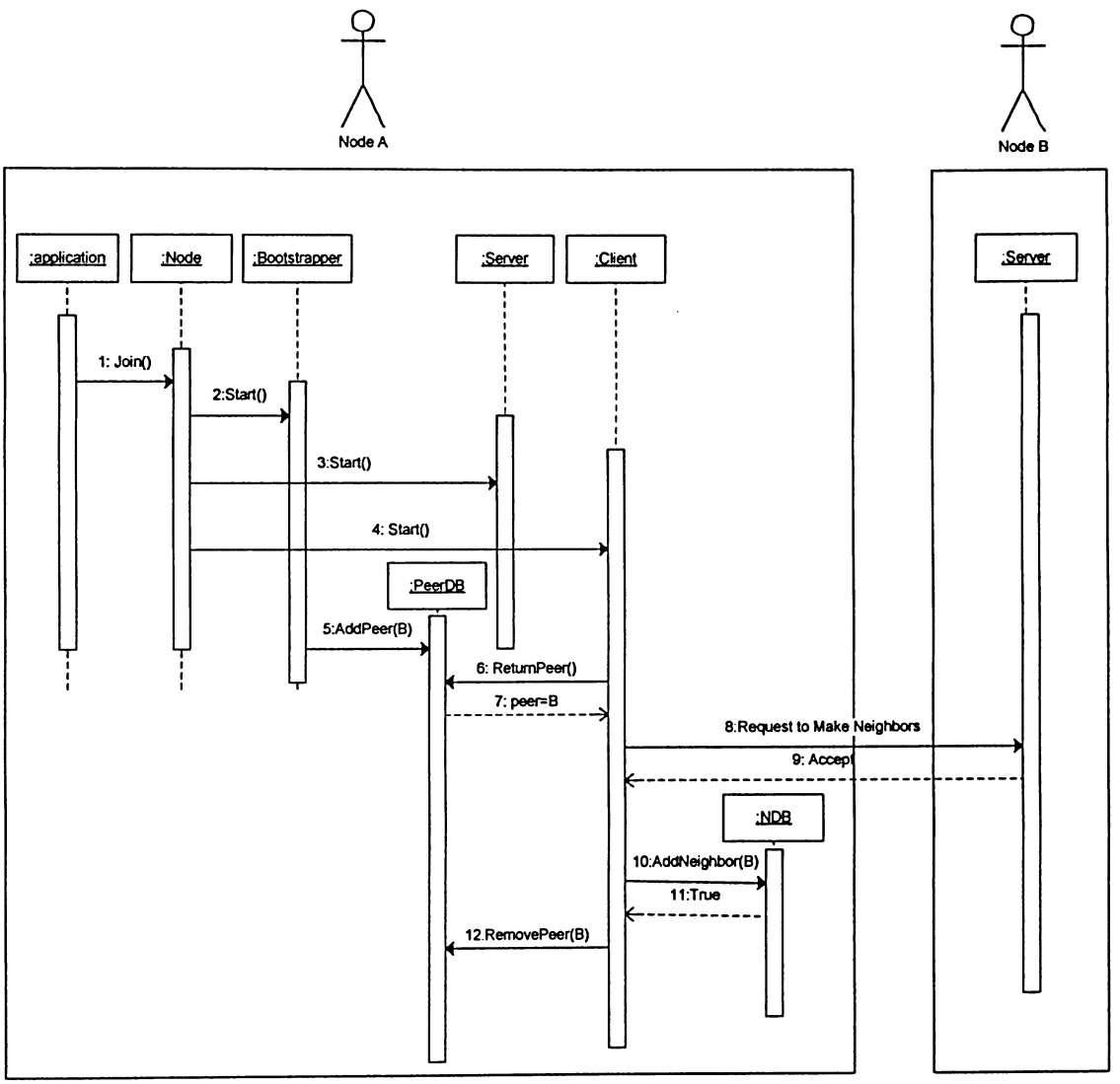

Figure 4.17 : Join and Neighbor Establishment

For each created neighbor TCP connection, a separate thread is dispatched. That is why both Client and Server objects inherit from java.lang.Thread. Figure 4.18 shows a snapshot of the Client.run() method. The variable runClient is a Boolean variable that is initialized to true. As long as the node is not satisfied, the Client object retrieves a peer object from PDB, the PeerDB object. It then clones itself, and calls Client.run(peer) to

establish the connection and manage it on a separate thread. Figure 4.19, shows the Server.run() method. Again, runServer is a Boolean variable that is set to true to turn on the server, which listens for incoming TCP connections. When a connection arrives, the object clones itself in order to dedicate the cloned object on separate thread to take care of the incoming connection, and it goes back to listening for more incoming TCP connections.

```
public void run(){
    int PDBsize = PDB.ReturnSize();
    if (peer == null){
     while (runClient){
       if (NDB.notSatisfied()) {
                for(int i=0; i<PDBsize ;i++){
                    Peer peer = PDB.ReturnPeer(i);
         //if the returned peer is not null and if it is not an already existing neighbor
                    if (peer != null && NDB.IsNotNeighbor(peer) ){
                        try{
                            //clone the current Client object
                            Client newClient = (Client) clone();
                            //call run(peer) of the newly created Client object
                            newClient.runner = new Thread(newClient);
                            newClient.peer = peer;
                            newClient.runner.start();
                            try{ Thread.sleep(5000);}catch(InterruptedException e){}
                        }catch(Exception e){}
                    }//if
                }//for
       }//if
    }//while
   }else {run(peer);}
 }//run
```

Figure 4.18 : Client.run() dispatches a separate thread for each peer

```
public void run(){
    if (server != null){ //only the first runner server will execute this part
        while (runServer){
            try {

                Socket datasocket = server.accept();
                Server newSocket = (Server) clone();
                newSocket.server = null;
                newSocket.data = datasocket;
                newSocket.runner = new Thread(newSocket);
                newSocket.runner.start();

            } catch (Exception e){}
        }
    } else {  // the other threads will come here.
        run(data);
    }
}
```

Figure 4.19 : Server.run() dispatches a separate thread for each incoming connection

When a Neighbor object is created, it instantiates a LinkColorScheme object, which inherits from java.lang.TimerTask and overrides its run() method. The Neighbor object starts a timer which when it expires, LinkColorScheme.run() method executes. The duration of the timer can be set in Preferences.LCSTimer. The LinkColorScheme periodic execution is triggered only from the server side of the connection.

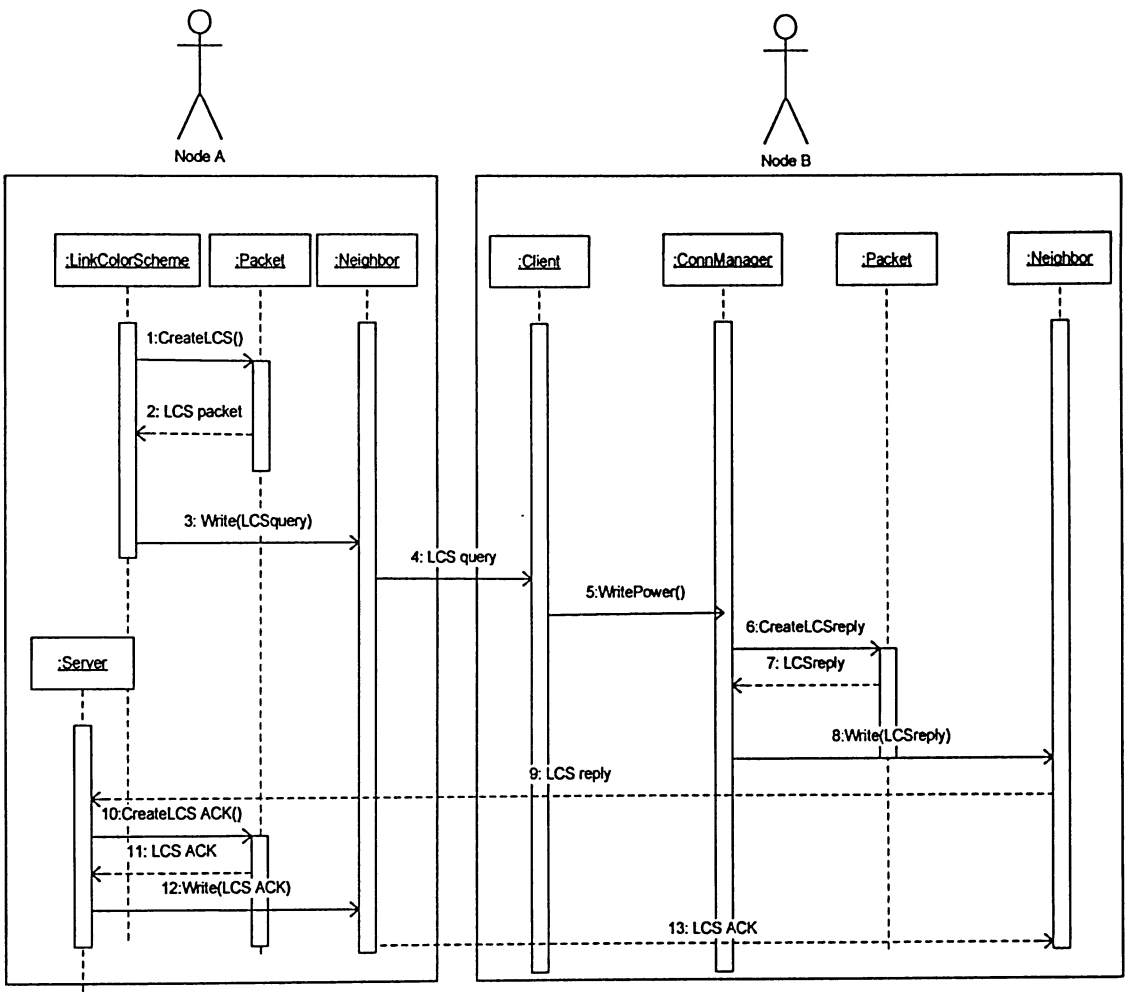

Figure 4.20 : LCS Sequence Diagram. Node A is the Server side of the connection, whereas Node B is the Client side.

For example, in figure 4.20, node A acts as the server side of the connection, while node B acts as the client side. Node A has a *:Neighbor* which abstracts its neighbor, node B. *:Neighbor* is associated with the *:LinkColorScheme* as shown the figure. When the timer

expires, LinkColorScheme.run() method executes on a separate thread. It first saves the current time stamp. It then contacts the Packet object to create an LCS query packet (message 1). When the LCS query packet is created, LinkColorScheme object calls Neighbor.Write() method in order to send the packet to Node B (message 3).

When the LCS query packet arrives at the Client object of node B, the Client object asks its parent, *:ConnManager* to handle the packet (message 5). *:ConnManager* retrieves the power value of node A written in the packet. It also takes the current time stamp of the system. Then, it contacts *:Packet* to create an LCS query reply that includes node B's power. In message 8 in figure 4.20, *:ConnManager* contacts the *:Neighbor* which represents node A in order to send the LCS query reply to node A. *:Neighbor* then sends the packet to *:Server* of node A. From the LCS query reply, node A retrieves the power value of node B, and calculates the RTT value by subtracting the current time stamp from the time stamp it took before sending the LCS query packet. The final step for node A is to send an LCS ACK packet, so that node B can calculate the RTT value as well.

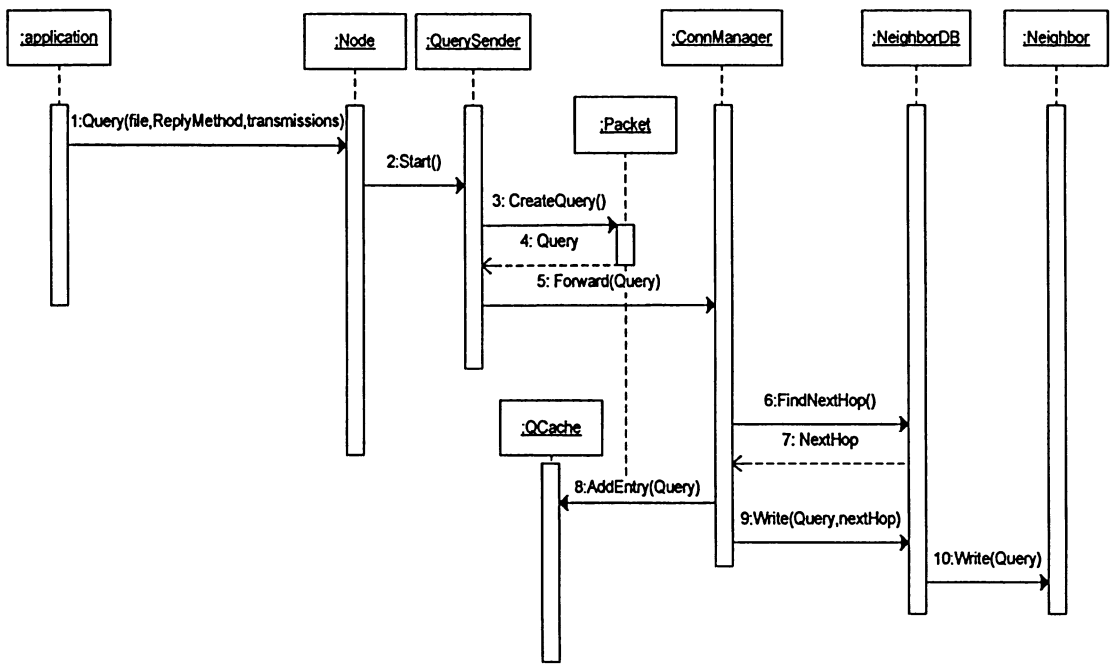

Figure 4.21 : Node.Query() called by the application Sequence Diagram

For each overlay node, the RTT, power level and capacity values of each overlay neighbor is stored in the corresponding Neighbor object. To have a better and more accurate measurement of RTT, each Neighbor object keeps track of a number of consecutive RTT measurements. Then the average RTT value is calculated and used along with the power value of the neighbor in order to classify the link to the neighbor, using the color function illustrated by table 1.1.

After Neighbor objects are created, and LCS is performed, the system is ready to perform routing or forwarding. There are two types of overlay packets that can invoke routing, resource query and data packets. Those packets are either created in the same system because the user application needs to route them, or received through the overlay network and need to be handled.

Figure 4.21 shows the sequence of events when an application calls the Query() method. The query method takes three parameters, the file name, the method of reply preferred, and the number of query packet retransmissions, in case a reply is not received within a certain time limit. The method of reply can be TCP or resource query reverse path from the destination. In message 3, *:Node* asks the *:Packet* to create a resource query packet.

When the newly created packet is returned (message 4), *:Node* asks *:ConnManager* to forward the packet (message 5). *:ConnManager* asks the *:NeighborDB* to find the next hop (message 6). The next hop returned is an integer value that represents the index of the *:Neighbor* to which the resource query packet is forwarded. When the next hop is returned (message 7), *:ConnManager* adds an entry to the query cache (message 8), and asks the *:NeighborDB* to send the query to the next hop (message 9). Finally, *:NeighborDB* sends the packet to the neighbor whose index is equal to next hop (message 10).

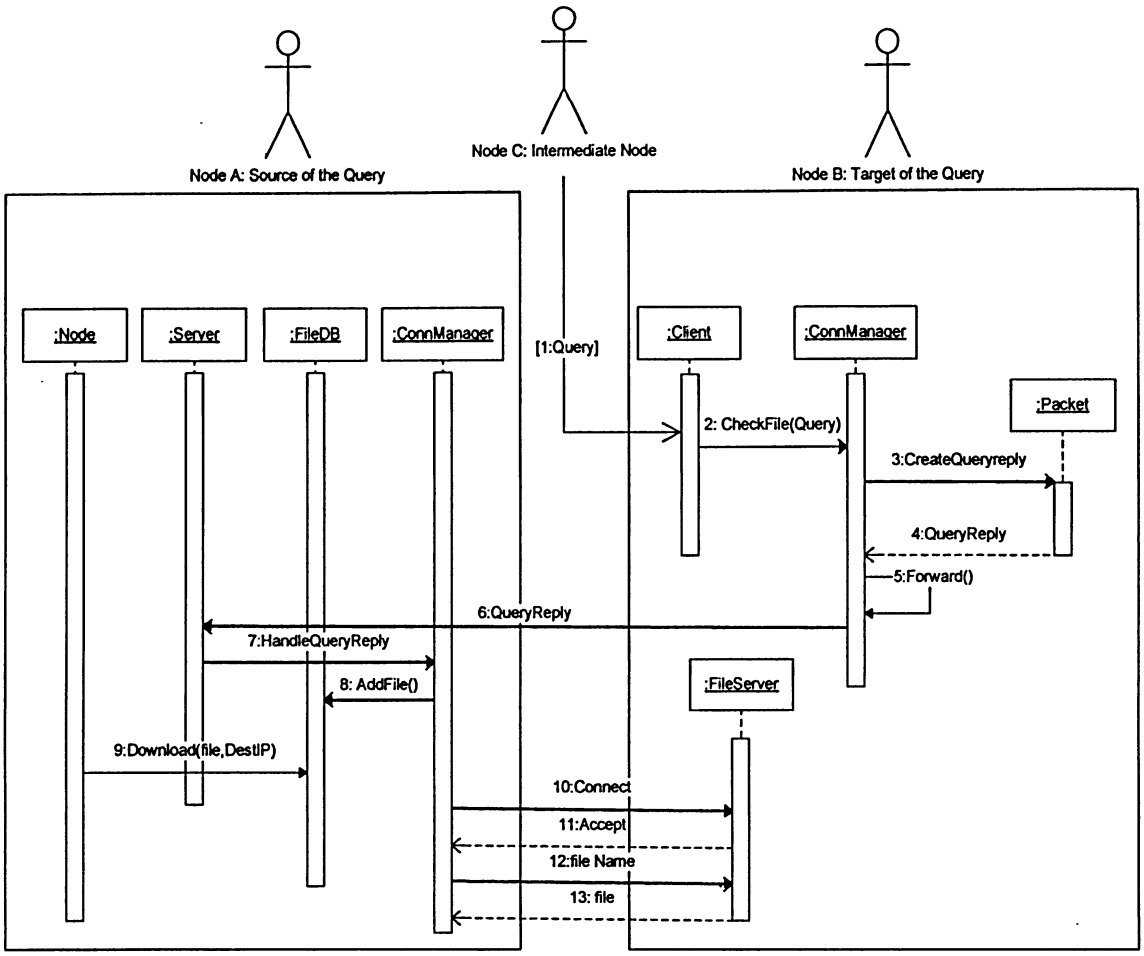

Figure 4.22 : File Transfer Sequence Diagram

Figure 4.22 shows the sequence of events that take place between objects when a resource query packet arrives at an overlay node that has the resource queried. In the diagram, node A is the source, node C is an intermediate node and node B is the destination. Node B, when it receives the resource query, it checks if it has the resource (message 2). *:ConnManager* of node B finds out that the resource exists in its host system, in a predefined directory, so it asks *:Packet* to create a query reply packet (message 3). The query reply packet is returned to *:ConnManager* (message 4).

When a resource query hits a node in which the resource in question exists, there are two ways to send back the query reply, either through the reverse path to the source, or by contacting the source directly through TCP. In figure 4.22, the second approach is adopted. Node B sends the query reply directly to the source through TCP. The Server

object of Node A receives the reply (message 5), and then asks *:ConnManager* to handle the query reply (message 6).

:ConnManager creates a new file database entry and adds it to *:FileDB* (message 8). When the application calls Node.Download(file) function, *:ConnManager* proceeds with downloading the file (message 9). It connects to the *:FileServer* of node B using the address specified in the query reply packet over TCP (message 10), and it requests the file by its name (messages 11 and 12). Finally, the *:FileServer* of node B sends the file to *:ConnManager* of node A (message 13).

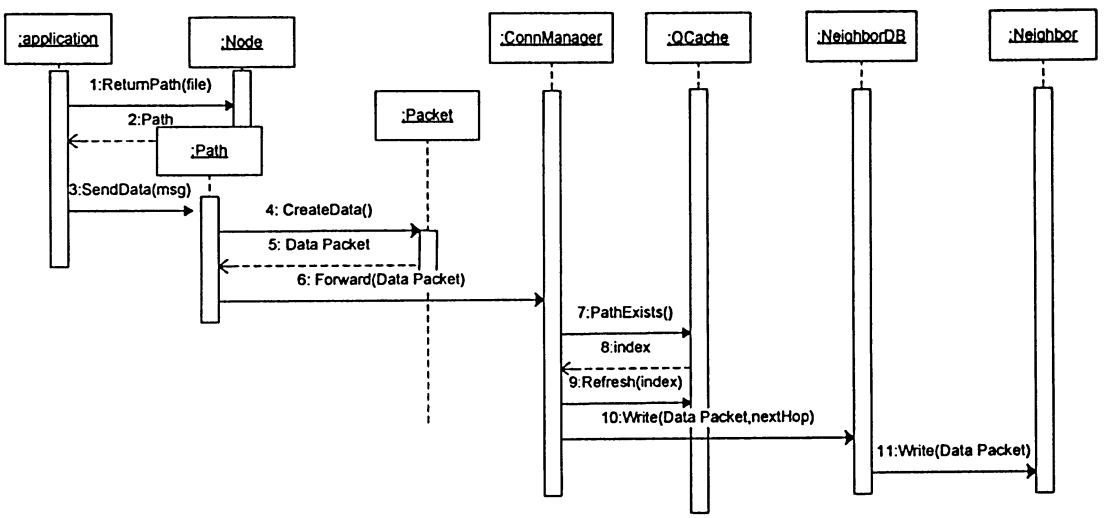

Figure 4.23 : Path.SendData() issues a Data Packet Sequence Diagram

Node interface also allows the application to send data to node after the overlay path between source and destination has been established. Figure 4.23 shows the sequence of events that occur when the application calls SendData() method. *:Path* uses *:Packet* to create a data packet (messages 4 and 5). *:Path* passes the packet to *:ConnManager* to forward the packet (message 6). If a route exists in *:QCache*, it is refreshed and the next hop is retrieved (messages 7-9). Finally, *:ConnManger* contacts *:Neighbor* to send the packet (message 10-11).

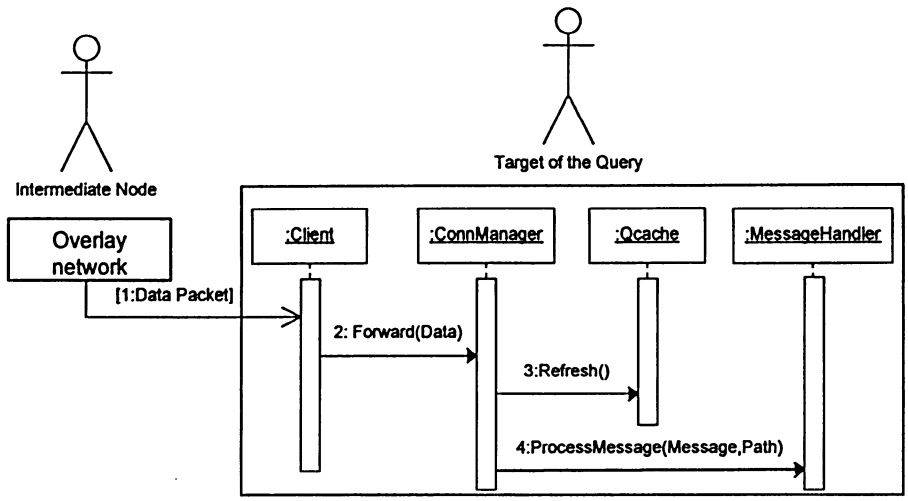

Figure 4.24 : Data Packet received at the Destination Sequence Diagram

Figure 4.24 shows what happens to the target system when the data packet is received. The Client object of the target receives the data packet (message 1). *:Client* passes the data packet to *:ConnManager* to forward it. *:ConnManager* finds out that this node is the destination node and refreshes the corresponding path entry in *:Qcache* (message 3). Finally, *:ConnManager* passes the message port of the data packet to the application support layer by calling ProcessMessage() method of MessageHandler, an interface that is implemented by the user.

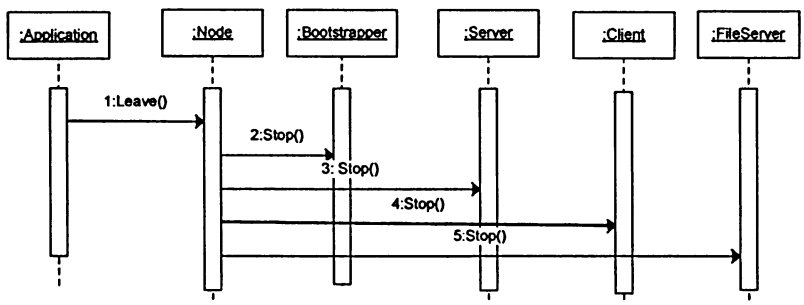

Figure 4.25 : Node.Leave() Sequence Diagram

The overlay leave operation is depicted by the sequence diagram in figure 4.25. When the application calls the Leave() method of Node object. The Node object stops all of the running threads including *:Bootstrapper*, *:Server*, *:Client* and *:FileServer* threads.

5 Evaluation and Validation of Design

In the previous chapter, we presented the design of our unstructured overlay middleware. We proceed in this chapter as follows; we evaluate the design by showing how our middleware meets its non-functional requirements. Middleware validation is illustrated by showing how developing different applications can be significantly simplified with the use of our API. We illustrate how our middleware works by presenting ethereal simulations and execution traces. Finally, we present our middleware's CPU and memory performance measurements.

5.1 Non-Functional Requirements Achieved

5.1.1 Ease of Use

Our middleware implements an easy-to-use application-support layer API, which interfaces the underlying complex overlay layer. Actually, developers can gain the benefits of the overlay without really having to understand how it works in details.

5.1.2 Portability

Portability is a key non-functional requirement, which our middleware achieves since it conforms to the specification of the J2ME platform. Please refer to section 3.1.1 for an explanation how J2ME helps us achieve portability.

5.1.3 Independence of underlying Ad Hoc Layer

Our middleware can run on top of any MANET routing protocol such as OLSR [43], AODV or DSR [43]. The only condition required is that the routing protocol must support multicast. However, it is important to point out that our system can perform best on top of AODV or on top of a routing protocol that can provide the overlay layer with link RTT and power values of neighbors. In section 5.7, we elaborate on how to optimize performance by utilizing the underlying routing protocols.

5.1.4 Efficiency

Our middleware achieves two efficiency parameters, which are namely, time and space. As for time efficiency, since our middleware is multithreaded; each main component runs on a separate thread concurrently with other components. No one component delays the other. In addition, a simple routing algorithm is adopted at the overlay layer. Space efficiency is also improved because overlay routing does not use a routing table. Rather, caching is used. Secondly, JVM's garbage collection makes sure unneeded memory is deallocated.

5.1.5 Network Friendly

Our resource discovery mechanism relies on a single query packet, unlike some other ad hoc middlewares, such as [29], in which resource discovery is based on broadcasting resources. Announcing resources and services periodically is hostile to the wireless network, since it consumes bandwidth. Also, maintaining the overlay path does not require the transmission of keepalive packets or heartbeats. Rather, a path remains in caches as long as it is being used by the application, because data packets refresh the path.

5.1.6 Reliability

Reliability is achieved in two ways. Firstly, our routing algorithm is power-aware. This is because routing is based on link colors. A link color is based on RTT of a link to a neighbor, and the neighbor's current power level. This can guard against loosing packets as higher power nodes are less likely to disconnect.

Secondly, routing is improved with the use of error feedback packets that are sent from the first node that discovers a broken link towards the sender of the data packet. The error feedback packet not only informs the application of lost data packets, but also removes the stale routes from caches along the way to the source, resulting in an overall more reliable and robust system.

5.1.7 Usability

Our middleware can support the development of a wide range of MANET applications. We shall show how different applications can use our API in the next section.

5.2 Middleware Deployment

From an application's perspective, our middleware provides a mechanism for locating files in the overlay network through queries and query replies. Also, the file locating process involves a path establishment between source and destination of queries. How can this simple mechanism provide a base for developing different applications? We answer this question in the following few sections by providing example applications that can be built on top of our middleware.

5.2.1 Application-Level Multicast

Application-level multicasting (ALM) is one popular application that has been suggested in many systems, such as [26], [27], [28], and [7], to scale well on top of overlay networks. In this section, we explain how we can build an ALM application using our system.

Basically, a multicast group is created when a node, referred to as the root, creates a file with the name that is equal to the group ID. Creating a group ID file can be done with a call to Node.CreateFile() method. If any node is interested in joining the group, it first issues an overlay query packet, by calling Node.Query() method, to locate the group ID file (e.g. Query(grpID, ReplyMethod , Transmissions) where ReplyMethod is reverse-overlay path). If the query packet arrives at the root, its overlay layer generates a query reply that is sent towards the source through the overlay reverse path.

When the querying node receives the query reply packet sent from the root, it sends to the root a join message (e.g Path.SendData(JOIN,grpID)). When the root (or any other member node) receives the join message, it adds the overlay address of the newly joining node to its list of multicast children. The newly joining node saves the overlay address of

the root as its multicast parent. It then creates a group ID file. Thus, the file will be replicated in the network as more nodes join the multicast group. Whenever a member node receives an application level join message, it adds the sender to its list of children and then sends it a query reply.

To multicast data to the group members, each node simply sends an overlay data packet which carries ALM message to its multicast parent towards the root (e.g. SendData(MULTICAST,grpID,msg)). When the root receives the multicast message, it sends out the overlay data packet to all of its children. Similarly, a node can anycast by sending an overlay data packet carrying anycast message to its parent, which in turn propagates overlay data packets to all of its children. A non member node that wishes to multicast data to a specific group can send an overlay urgent data packet whose destination is a node that has the group ID file. The first group member node that receives the packet will propagate the multicast message to its parents towards the root.

Finally, a node can leave a multicast group by sending an overlay data packet carrying an application-level leave message to its multicast parent and children (e.g. SendData(LEAVE,grpID,msg)). Its parent simply removes the leaving node from the list of its children. The leaving nodes children search for a new parent. Furthermore, if a node disconnects without sending a leave message, overlay error feedback packets report to the application about broken paths whenever a child sends a data packet over the path. When the application receives an error feedback, it can issue a new query packet to find a new path to a new multicast parent.

It is important to point out that developing some applications paves the way for other applications. For example, ALM can be used as a base application that allows the implementation of streaming applications like video and voice conferencing. Figure 5.1 shows how ALM algorithm can be implemented using our API.

```
Class MessageHandlerImp implements MessageHandler{

DB = [groupID  parent  children[ ] ]
DB[] GroupsDB;
Node node;
//invoked by the overlay layer when an an application-level message is received.
void ProcessMessage(byte[] message, Path path){
        Path parent = GroupsDB.GetParent(message.GrpID);
        switch(message.type){
                MULTICAST {
                        if (path == parent || parent == null) MulticastDataToChildren(message);
                        else parent.SendData(message);
                }
                JOIN{
                        GroupsDB.AddChild(message.GrpID,path);
                }
                LEAVE{
                        if (path != parent) GroupsDB.RemoveChild(message.GrpID,path);
                        else{
                                parent = null;
                                JoinGroup(message.GrpID);
                        }
                }
                ANYCAST{
                        MulticastDataToChildren(message);
                }
        }

}
void MulticastDataToChildren(GrpID, message){
        while(more_children_available()){
                Path path = DB.GetNextChild(message.GrpID);
                path.SendData(message);
        }
}
void JoinGroup(GrpID){
        ReplyMethod = 2 :Reverse Overlay Path
        Transmissions = 1;
        node.Query(GrpID, ReplyMethod , Transmissions );
        Wait(Preferences.QueryTimer);
        Path path = node.ReturnPath(GrpID);
        If (path != null){
                Path parent = path;
                GroupsDB.add (GrpID and parent);
                Node.CreateFile(GrpID);
                byte[] message = [JOIN  GRPID ];
                path.SendData(message);
        }

}
void MulticastData(GrpID){
        Path parent = GroupsDB.GetParent(GrpID);
        while (more_data_available() && parent != null){
                byte[] DATA = getData();
                byte[] message = [MULTICAST  GRPID  DATA];
                parent.SendData(message);
        }

}
void LeaveGroup(GrpID){
        byte[] message = [LEAVE GRPID];
        Path parent = GroupsDB.GetParent(GrpID);
        parent.SendData(message);
        MulticastDataToChildren(message);
}
Void CreateGroup(GrpID){
        node.CreateFile(GrpID);
        Path parent = null;
        GroupsDB.add(GrpID,Parent);
}
//Start Here
void ALM( DB_size, imax_neighbors, GrpID){
        GroupsDB = new database(DB_size);
        node = new Node(max_neighbors);
        Node.Join();
        JoinGroup(GrpID);
        MulticastData();
        LeaveGroup(GrpID);
        Node.Leave();
}
```

Figure 5.1 : ALM Algorithm

5.2.2 File Sharing

One straightforward application that can be easily built on top of our middleware is a file sharing application. The user of the application can store the files to be shared in a specific directory. First, the application can search for the location of the file on the overlay network by making a call to Node.Query() method which issues an overlay query packet. Because file download does not require keeping a path between the source and destination of the query, the source can set the reply method of the query packet to TCP.

When the query arrives at a node that has the file, it sends a query reply to the source over TCP. The query reply packet contains the overlay address of the destination. It also contains the file information. If the source is willing to download the file, it can make a call to Node.Download() method to download the file. Figure 5.2 shows an algorithm of file sharing application that uses our API.

```
Void StartFileSharing(max_neighbors){
        node = new Node(max_neighbors);
        node.join();
        int ReplyMethod = 2;
        int Transmissions = 1;
        node.Query(flower.JPG,ReplyMethod,Transmsiions);
        wait(Preferences.QueryTimer);
        Path path = node.ReturnPath(flower.JPG);
        if (path != null){
                node.Download(flower.JPG,path);
        }
}
```

Figure 5.2 : File Sharing Algorithm

5.2.3 Gaming or Chatting Applications

Gaming and chatting applications implementation will also be straightforward if our API is used. Basically, each game has a certain game ID and a corresponding ID file. When the end user wants to play a certain game with other players on the overlay network, it can inform its application. The application then uses Node.CreateFile() method to create a file whose name is equal to the game ID.

To find other players on the network, the application can use Node.Query() method to search for the game ID file on the network (e.g. Query(GameID, ReplyMethod, Transmissions)). The path establishment between two players will be complete when the source receives a query reply packet. Next, the application can use Path.SendData() method in order to send data between the two players, like invitation and permission to play and game data as well.

```
Class MessageHandlerImp implements MessageHandler{
Node node;
//Invoked by the overlay layer when an application-level message arrives
Void ProcessMessage(byet[] message, Path path){
        switch(message.type){
                INVITE{
                        boolean reply = Prompt_Player();
                        byte[] DATA;
                        if (reply) {
                                DATA = [ACCEPT,GameID];
                                Launch_Game();
                        }
                        else DATA = [DENY GameID]];
                        path.SendData(DATA);
                }
                ACCEPT{
                        Launch_Game(path);
                }
                DENY{
                        SearchPlayer();
                }
                GAME{
                        //Game Data could be the next movement or result
                        Process(GAME_DATA);
                }
}
Void LaunchGame(path){
        byte[] GAME_DATA = GetNext();
        byte[] DATA = [GAME GameID  GAME_DATA];
        path.SendData(DATA);
}
Void SearchPlayer(GameID){
        Transmissions = 1;
        ReplyMethod = 2;
        node.Query(GameID, ReplyMethod,Transmissions);
        wait(Preferences.QueryTimer);
        Path path = node.ReturnPath(GameID);
        if (path != null){
                byte[] DATA = [INVITE GAMEID];
                path.SendData(DATA);
        }
}
//Start Here
Void StartGaming(max_neighbors,GameID){
        node = new Node(max_neighbors);
        node.join();
        node.CreateFile(GameID);
        SearchPlayer(GameID);
        }
}
```

Figure 5.3 : Gaming Application Algorithm

For example, if a player is willing to play Tic-Tac-Toe with another player. It makes a request to its application to start a new game. The application will have a table that associates each game with an ID. The application will create a file whose name is equal to the ID of Tic-Tac-Toe and places it in a specific directory. Then, it will start looking for other players by sending a query packet that will be looking for the same file in the network. Figure 5.3 shows the structure of a gaming application that is based on our API.

If another player on the network is willing to play the same game, then the same file will also exist in its program directory and a query reply will be sent towards the first player. Once the path is established, the first player can send data packet carrying invitation message to the second player, if the second player sends acceptance message in the data packet, Tic-Tac-Toe interface will be launched at both sides. Every time a player makes a move, its move information is sent as a message in the data packet to the other player. Winning, losing or draw information is also sent to both players at the end of the game.

5.3 Demonstration Using a Simple Data Transfer Application

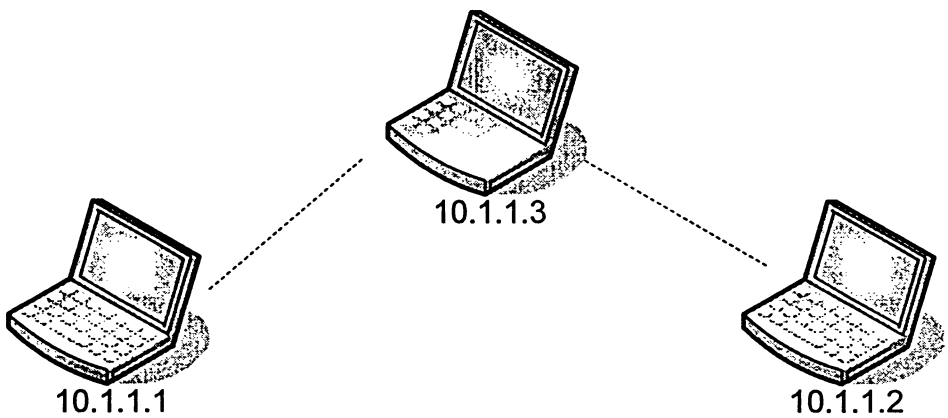

Figure 5.4 : Simulation Topology

This section provides a demonstration of how our middleware works on the real network. Due to the nature of ad hoc network, simulating on the real world environment with a large number of nodes is very difficult. Therefore, we will use a simple topology consisting of three laptops with their wireless cards configured to run in ad hoc mode.

```
package OverlayMiddleware.util;

/**
*Configurations and preferences.
*/
public class Preferences {
   private Preferences(){}
   //HostIP can either be an IP in String format, or "null", if it is null, the bootstrapper will choose an interface to announce
   public static final String  HostIP= "null";
   //The port on which the File Server operates
   public static final int FileServerPort = 5555;
   // The port on which the bootstrapper operates
   public static final int BootStrapperPort = 1111;
   // The port on which the Client and server operate
   public static final int ConnManagerport = 4444;
   //The multicast group used for bootstrapping
   public static final String multicastAddress = "228.5.6.7";
   //The file database size
   public static final int FileDBsize = 100;
   //The duration of time between sending a Join multicast requests, specified in milliseconds
   public static final long MulticastTimer = 10000;
   //Join packet TTL, used if propagating multicast packets is required.
   public static final int JoinTTL = 1;
   //The overlay packet size in bytes, 40 bytes must be allowed for header
   public static final int OverlayPacketSize = 100;
   //The size of the Query Cache
   public static final int CacheSize = 10;
   //The duration of time between sending an LCS request and the next, specified in milliseconds
   public static final int LCSTimer = 30000;
   //The duration of time before the first execution of the LCSTimer object when a new neighbor si created
   public static final int LCSTimerOffset = 10000;
   //The duration of time between the same query retransmissions if a reply is not recieved,specified in milliseconds
   public static final int QueryTimer = 70000;
   //The RTT window size over which the average RTT is measured. 0 is a bad value
   public static final int RTTwindowSize = 5;
   //Configure whether the multicast requests are to propagate or not
   public static final String PropagateMulticastRequests = "NO"; //nodes are set not to propagate the multicast requests
   //File Server is set to work when the Node object joins.
   public static final String StartFileServer = "YES";
   //The capacity setting used in MBR, can be changed through Node.UpdateCapacity();
   public static int Capacity = 1;
   //The power setting used in link coloring scheme, and can be changed through Node.UpdatePower();
   public static int Power = 1;
}
```

Figure 5.5 : Preferences Class

All of three stations run our middleware, and a test application. Our middleware was originally developed using NetBeans Integrated Development Environment (IDE) [36] in a project named, OverlayMiddleware. The test application specifies that nodes 10.1.1.1 and 10.1.1.2 can have only one neighbor, while 10.1.1.3 can have two neighbors. Applications running on 10.1.1.3 and 10.1.1.2 are started first so that they make neighbors with each other first before starting 10.1.1.1. When 10.1.1.1 runs the test application, it makes neighbors with 10.1.1.3 because it is the only node that can make 2 neighbor connections. Our purpose of this test is to form the overlay topology shown in figure 5.4. Also, the application in 10.1.1.1 will query a file, me.txt, which only exists in 10.1.1.2. All of the timers and preferences setting of our middleware are configured as shown in OverlayMiddleware.util.Preferences class depicted in figure 5.5.

Figure 5.6 shows the source code of the application running in 10.1.1.1. All the application has to do is to import OverlayMiddleware.AppSupport package. As we have mentioned already in the previous chapter, any application using our middleware must first implement the MessageHandler interface. The MessageHandlerImp class is an implementation of MessageHandler, and it also adds the method Start(). This method is called by the main function to start the program.

```
package application;
import OverlayMiddleware.AppSupport.*;
import OverlayMiddleware.util.Log;

class MessageHandlerImp implements MessageHandler{
   Node node;
   public void ProcessMessage(byte[] Message, Path path){
      Log.log(new String(Message).trim()+" Receieved from :");
      Log.log(path.ReturnDestIP+" "+path.ReturnDestID+" "+path.ReturnSequence);
   }
    public void ProcessException(Exception e, byte[] message, Path path){
      Log.log("EXCEPTION"+ e);
   }
    public void Start(){
      int max_neighbors=1;
      Node node = new Node(max_neighbors, this);
      node.Join();
      try{Thread.sleep(40000);}catch(InterruptedException e){}
      int transmissions = 2;
      int ReplyMethod = 2; // The reply method prefered is overlay reverse path, if 1 then TCP
      node.Query("me.txt",ReplyMethod,transmissions);
      try{Thread.sleep(40000);}catch(InterruptedException e){}
      Path path = node.ReturnPath("me.txt");
      byte[] msg = "PING".getBytes();
      try{
        path.SendData(msg);
        path.Refresh();
       }catch(Exception e){ System.out.print("exception "+e);}
        try{Thread.sleep(40000);}catch(InterruptedException e){}
      Log.log("leaving now");
      node.Leave();
   }
}
public class Main {
   public static void main(String[] args) {
      MessageHandlerImp MsgHandler = new MessageHandlerImp();
      MsgHandler.Start();
   }
}
```

Figure 5.6 : Demo Application running on 10.1.1.1

When Start() is invoked, it first instantiates a new Node object. It passes the maximum number of neighbors which is equal to 1, and it also passes a reference to itself to the newly created Node object. It then calls Node.Join() to join the overlay network. After a wait of 40 seconds, Node.Query() is called to locate the file "me.txt" which only exists in 10.1.1.2. After another wait of 40 seconds, the application calls Node.ReturnPath(). This

function returns a path object, which the application uses to send the message, "PING", to the destination via the Path.SendData() method. Finally, the application refreshes the path to 10.1.1.2 using Path.Refresh() and then leaves using Node.Leave().

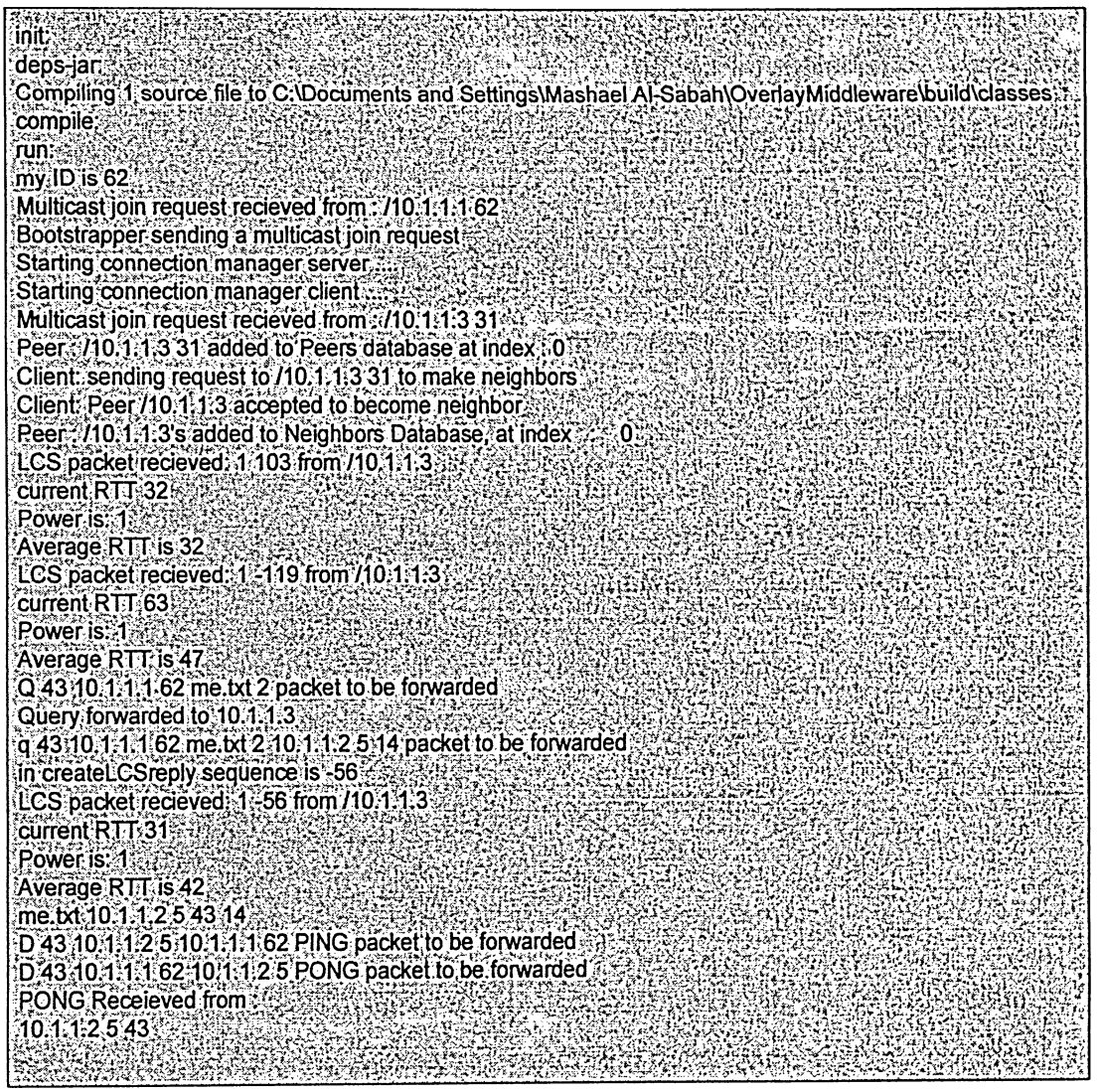

Figure 5.7 : 10.1.1.1 Output

The applications running in 10.1.1.2 and 10.1.1.3 are simpler. They both instantiate a new Node object and call Node.Join() to join the overlay network. The only difference between them is that when 10.1.1.2 receives a message, it sends the reply message "PONG". Figures 5.7, 5.8 and 5.9 show an execution trace of each node. We have used

print statements throughout the middleware program in order to be able to trace its execution.

```
init:
deps-jar:
Compiling 1 source file to C:\Documents and Settings\admin\OverlayMiddleware\
OverlayMiddleware\build\classes
compile:
run:
my ID is 5
Starting connection manager server ...
Multicast join request recieved from : /10.1.1.2 5
Bootstrapper sending a multicast join request
Starting connection manager client ...
Server: request recieved from : /10.1.1.3
Server: accept sent
Peer : /10.1.1.3's added to Neighbors Database, at index : 0
Multicast join request recieved from : /10.1.1.3 31
sending LCS with sequence -77
LCS packet recieved: 1 -77 from /10.1.1.3
current RTT 16
Power is: 1
Average RTT is 16
Multicast join request recieved from : /10.1.1.3 31
Multicast join request recieved from : /10.1.1.1 62
Peer : /10.1.1.1 62 added to Peers database at index : 0
Multicast join request recieved from : /10.1.1.3 31
sending LCS with sequence -12
LCS packet recieved: 1 -12 from /10.1.1.3
current RTT 31
Power is: 1
Average RTT is 23
File requested is: me.txt
file exists
Packet : q 43 10.1.1.1 62 me.txt 2 10.1.1.2 5 14 : is to be forwarded
previous hop is /10.1.1.3
sending LCS with sequence 102
LCS packet recieved: 1 102 from /10.1.1.3
current RTT 31
Power is: 1
Average RTT is 26
Packet : D 43 10.1.1.2 5 10.1.1.1 62 PING : is to be forwarded
PING received from
10.1.1.1 62 43 :
Packet : D 43 10.1.1.1 62 10.1.1.2 5 PONG : is to be forwarded
sending LCS with sequence 8
LCS packet recieved: 1 8 from /10.1.1.3
current RTT 32
Power is: 1
Average RTT is 27
```

Figure 5.8 : 10.1.1.2 Output

```
run
my ID is 31
Bootstrapper sending a multicast join request
Starting connection manager server
Multicast join request recieved from : /10.1.1.3 31
Starting connection manager client
Multicast join request recieved from : /10.1.1.2 5
Peer : /10.1.1.2 5 added to Peers database at index : 0
client: sending request to /10.1.1.2 5 to make neighbors
client: Peer /10.1.1.2 accepted to become neighbor
Peer: /10.1.1.2's added to Neighbors Database, at index : 0
Multicast join request recieved from : /10.1.1.3 31
Bootstrapper sending a multicast join request
LCS packet recieved: 1 -77 from /10.1.1.2
current RTT 31
Power is: 1
Average RTT is 31
Multicast join request recieved from : /10.1.1.3 31
Bootstrapper sending a multicast join request
Multicast join request recieved from : /10.1.1.1 62
Peer : /10.1.1.1 62 added to Peers database at index : 0
Multicast join request recieved from : /10.1.1.3 31
Bootstrapper sending a multicast join request
Server: request recieved from : /10.1.1.1
Server: accept sent
Peer : /10.1.1.1's added to Neighbors Database, at index : 1
sending LCS with sequence 103
LCS packet recieved: 1 103 from /10.1.1.1
current RTT 31
Power is: 1
Average RTT is 31
LCS packet recieved: 1 -12 from /10.1.1.2
current RTT 31
Power is: 1
Average RTT is 31
sending LCS with sequence -119
LCS packet recieved: 1 -119 from /10.1.1.1
current RTT 31
Power is: 1
Average RTT is 31
File requested is: me.txt
Requested file does not exist
Q 43 10.1.1.1 62 me.txt 2 packet to be forwarded
Query forwarded to 10.1.1.2
q 43 10.1.1.1 62 me.txt 2 10.1.1.2 5 14 packet to be forwarded
LCS packet recieved: 1 102 from /10.1.1.2
current RTT 31
Power is: 1
Average RTT is 31
sending LCS with sequence -56
LCS packet recieved: 1 -56 from /10.1.1.1
current RTT 31
Power is: 1
Average RTT is 31
D 43 10.1.1.2 5 10.1.1.1 62 PING packet to be forwarded
D 43 10.1.1.1 62 10.1.1.2 5 PONG packet to be forwarded
```

Figure 5.9 : 10.1.1.3 Output

Figure 5.10 shows a snapshot of packets captured by Ethereal [38] that we ran in 10.1.1.1. The first packet that 10.1.1.1 sent was an IGMP membership report to destination 224.0.0.22 as shown in packet number 5 captured in the figure. Packets 6 and 7 are overlay bootstrapping join packets that were being multicast in the network. After 10.1.1.1 learns about 10.1.1.3 using ARP as shown in packets 8 and 9, 10.1.1.1 establishes a TCP connection to 10.1.1.3, as shown in the 3-way handshake in packets 10, 11 and 12.

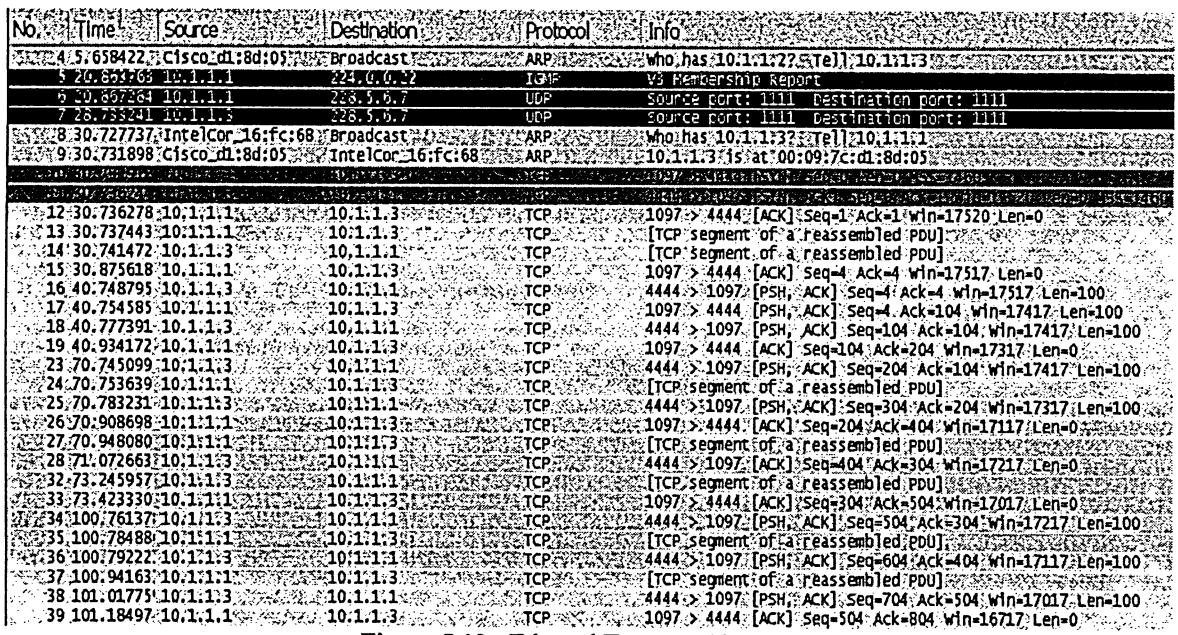

Figure 5.10 : Ethereal Traces at 10.1.1.1

In packet 13, 10.1.1.1 sends an overlay neighbor establishment request packet to 10.1.1.3, which sends an overlay neighbor establishment reply as shown in packet 14. This makes 10.1.1.1 the client side of the connection. Because LCS starts executing 10 seconds after a new neighbor is created, as configured in Preferences class, shown in figure 5.5, and because 10.1.1.3 is the server side of the connection, it sends an LCS query packet as shown in packet 16. The following two packets, 17 and 18 are also LCS reply and LCS acknowledgement packets exchanged.

Other important packets captured include packet number 27, which is the query packet sent from 10.1.1.1. Packet 32 is the query reply packet that comes from 10.1.1.3. Finally,

packets 37 and 38 are the data packets that were sent to and from 10.1.1.1 to 10.1.1.2 through 10.1.1.3.

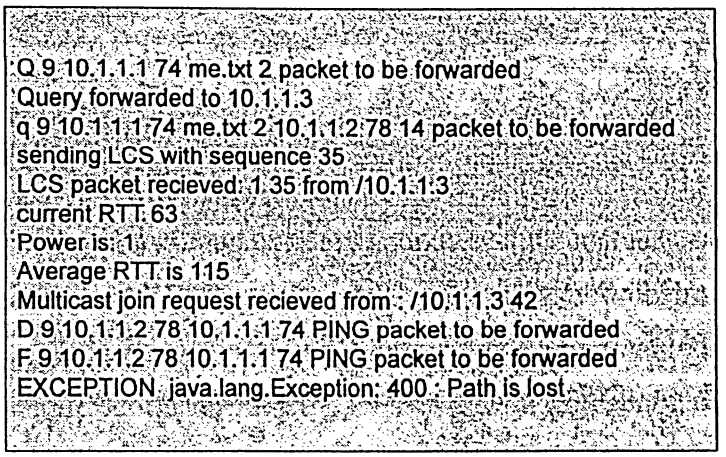

Figure 5.11 : Output of 10.1.1.1 when 10.1.1.2 disconnects

Figure 5.11 shows the output of 10.1.1.1s application when 10.1.1.2 disconnects from the network after the overlay path has been established and before the overlay data packet is sent from 10.1.1.1. Since 10.1.1.3 is the intermediate node between 10.1.1.1 and 10.1.1.2, it receives the data packet sent from 10.1.1.1 and tries to send it to 10.1.1.2. Because 10.1.1.2 is disconnected, 10.1.1.3 sends an error feedback packet towards 10.1.1.1. The effect of the error feedback is that it removes the stale cache information and invokes MessagHandler.ProcessException().

5.4 Performance Measurements

In this section, we present the statistics collected by NetBeans IDE profiler [35] that was used, in station 10.1.1.1, to analyze the performance of our system when it was running in a set up similar to the one in section 5.3. The same application shown in figure 5.6 will be used to perform performance measurements of our system. The only difference is that in this section we added a call to Node.Download("me.txt) to 10.1.1.1 application in order to actually download the file after a query reply is received from 10.1.1.2.

The NetBeans Profiler is a powerful tool that provides important information about the runtime behavior of an application. Imposing relatively little overhead, the NetBeans Profiler tracks thread state, CPU performance, and memory usage [34]. The performance is mainly concerned with how our system utilizes resources such as CPU time and memory.

5.4.1 CPU

Figure 5.13 shows the total CPU execution time and number of invocations for some methods of our middleware. The figure does not show CPU time for all methods in the system. The rest of our middleware's methods were left out because they use 0% CPU time. The first four methods, FileServer.run(), Server.run(),Bootstrapper.run() and Client.run() take up most of the CPU time because they start concurrently and each one of them has an infinite loop which only stops when Node.Leave() is called. Although NeighborDB.notSatisfied() and PeerDB.ReturnPeer() have the highest numbers of invocations (24275710 and 1024661, respectively), they do not take up any CPU time because they are of complexity O(1).

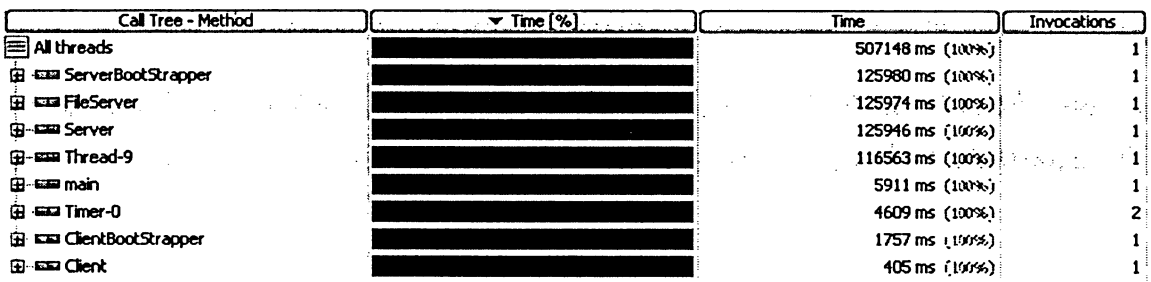

Figure 5.12 : CPU Time per Thread

It is worth pointing out that CPU performance can be further improved in applications in which the FileServer object is not needed. The FileServer is optional and can be turned off at OverlayMiddleware.util.Preferences class. Furthermore, Log.log is optional and is used to log the print statements into a file, which is an I/O operation that takes approximately 16.3 milliseconds of CPU times for 43 invocations. Figure 5.12 shows the total execution time per thread.

Hot Spots - Method	Self time ...	▼ Self time	Invocations
OverlayMiddleware.core.FileServer.**run** ()		125974 ms (24.8%)	1
OverlayMiddleware.core.Server.**run** ()		125946 ms (24.8%)	1
OverlayMiddleware.core.BootStrapper.**run** ()		122503 ms (24.2%)	2
OverlayMiddleware.core.Client.**run** (OverlayMiddleware.util.Peer)		113513 ms (22.4%)	1
OverlayMiddleware.AppSupport.Path.**SendData** (byte[])		5372 ms (1.1%)	2
OverlayMiddleware.util.PeerDB.**IsNotPeer** (OverlayMiddleware.util.Peer)		5240 ms (1%)	3
OverlayMiddleware.core.Client.**run** ()		3064 ms (0.6%)	2
OverlayMiddleware.core.ConnManager.**ForwardQueryReply** (byte[], OverlayMiddlewa...		3031 ms (0.6%)	1
OverlayMiddleware.core.ConnManager.**ForwardQuery** (byte[], OverlayMiddleware.util....		2515 ms (0.5%)	1
OverlayMiddleware.core.QuerySender.**run** ()		2093 ms (0.4%)	2
OverlayMiddleware.core.BootStrapper.**StopBootStrapper** ()		232 ms (0%)	2
OverlayMiddleware.core.ConnManager.**Download** (String, String, int)		133 ms (0%)	1
OverlayMiddleware.AppSupport.Node.**Query** (String, int, int)		65.8 ms (0%)	1
OverlayMiddleware.AppSupport.Node.**<init>** (int, OverlayMiddleware.AppSupport.Mess...		41.0 ms (0%)	1
OverlayMiddleware.core.BootStrapper.**<init>** (int, int, OverlayMiddleware.util.PeerDB, ...		24.9 ms (0%)	2
OverlayMiddleware.util.Log.**log** (String)		16.2 ms (0%)	43
OverlayMiddleware.AppSupport.Node.**Join** ()		9.7 ms (0%)	1
OverlayMiddleware.core.Server.**StartServer** (int, int)		2.61 ms (0%)	1
OverlayMiddleware.core.Client.**HandleLCSRequest** (byte[], OverlayMiddleware.util.Nei...		2.6 ms (0%)	4
OverlayMiddleware.util.FileDB.**<init>** (int)		1.67 ms (0%)	1
OverlayMiddleware.AppSupport.Node.**ReturnPath** (String)		1.45 ms (0%)	1
OverlayMiddleware.util.Neighbor.**Write** (byte[], String)		1.28 ms (0%)	7
OverlayMiddleware.core.FileServer.**Stop** ()		1.11 ms (0%)	1
OverlayMiddleware.core.Server.**StopServer** ()		1.4 ms (0%)	1
OverlayMiddleware.util.NeighborDB.**<init>** (int)		1.1 ms (0%)	1
OverlayMiddleware.AppSupport.Node.**Leave** ()		0.952 ms (0%)	1
OverlayMiddleware.core.ConnManager.**<init>** (OverlayMiddleware.util.NeighborDB, Ove...		0.935 ms (0%)	2
OverlayMiddleware.core.ConnManager.**ForwardData** (byte[], OverlayMiddleware.util.N...		0.822 ms (0%)	3
OverlayMiddleware.util.QueryCache.**<init>** ()		0.808 ms (0%)	1
OverlayMiddleware.util.PeerDB.**<init>** (int, int)		0.727 ms (0%)	1
OverlayMiddleware.util.Log.**create** ()		0.514 ms (0%)	1
OverlayMiddleware.core.ConnManager.**Forward** (byte[], OverlayMiddleware.util.Neighbor)		0.492 ms (0%)	5
OverlayMiddleware.util.Neighbor.**Disconnect** ()		0.467 ms (0%)	1
OverlayMiddleware.core.Client.**StartClient** (int, int)		0.433 ms (0%)	1
OverlayMiddleware.core.ConnManager.**HandleQueryReply** (byte[])		0.410 ms (0%)	1
OverlayMiddleware.util.Neighbor.**SetPowerRTT** (int, byte)		0.311 ms (0%)	4
OverlayMiddleware.util.Log.**close** ()		0.165 ms (0%)	1
OverlayMiddleware.util.Packet.**CreateLCSreply** (byte[], int, OverlayMiddleware.util.Nei...		0.161 ms (0%)	4
OverlayMiddleware.util.Packet.**CreateQuery** (String, int, String, int)		0.102 ms (0%)	1
OverlayMiddleware.util.NeighborDB.**IsNotNeighbor** (OverlayMiddleware.util.Peer)		0.001 ms (0%)	3
OverlayMiddleware.util.NeighborDB.**notSatisfied** ()		0.001 ms (0%)	24275710
OverlayMiddleware.util.PeerDB.**ReturnPeer** (int)		0.000 ms (0%)	1024661
OverlayMiddleware.util.QueryCache.**Refresh** (int)		0.000 ms (0%)	4

Figure 5.13 : CPU Time per Method

The total threads executed in the system are shown in figure 5.14. The first one belongs to the JVM and it starts the main thread which belongs to the user application of our middleware. The main thread calls Node.Join(), which then spawns the five threads:

ServerBootstrapper, ClientBootstrapper, Server, Client and FileServer. The Client thread spawns Thread-9 to monitor the connection with its neighbor. After a wait of 40 seconds (please refer to 10.1.1.1 application, figure 5.6), main thread calls Node.Query() which uses a Timer thread, Timer-0, to send the query packet.

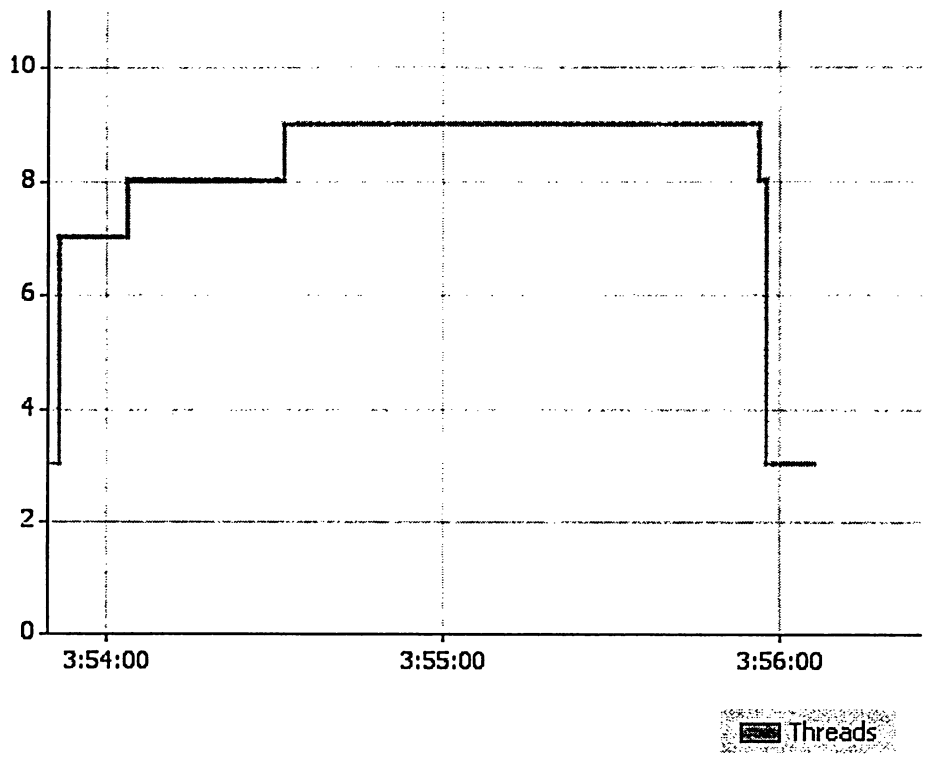

Figure 5.14 : Threads in the system during the 2 minutes execution of 10.1.1.1

Figure 5.15 shows the detailed thread states during the application execution. FileServer, Thread-9, Server and SeverBootstrapper graphs show that they were running 100% of the time during the application lifetime. The ClientBootstrapper thread state graph shows that it was sleeping 98.9% of its life time. The reason is that the way the ClientBootstrapper works is that it sends out an overlay join packet, and then sleeps for a duration of time that is set in Preferences.MulticastTimer (set to 10 seconds as shown in figure 5.5).

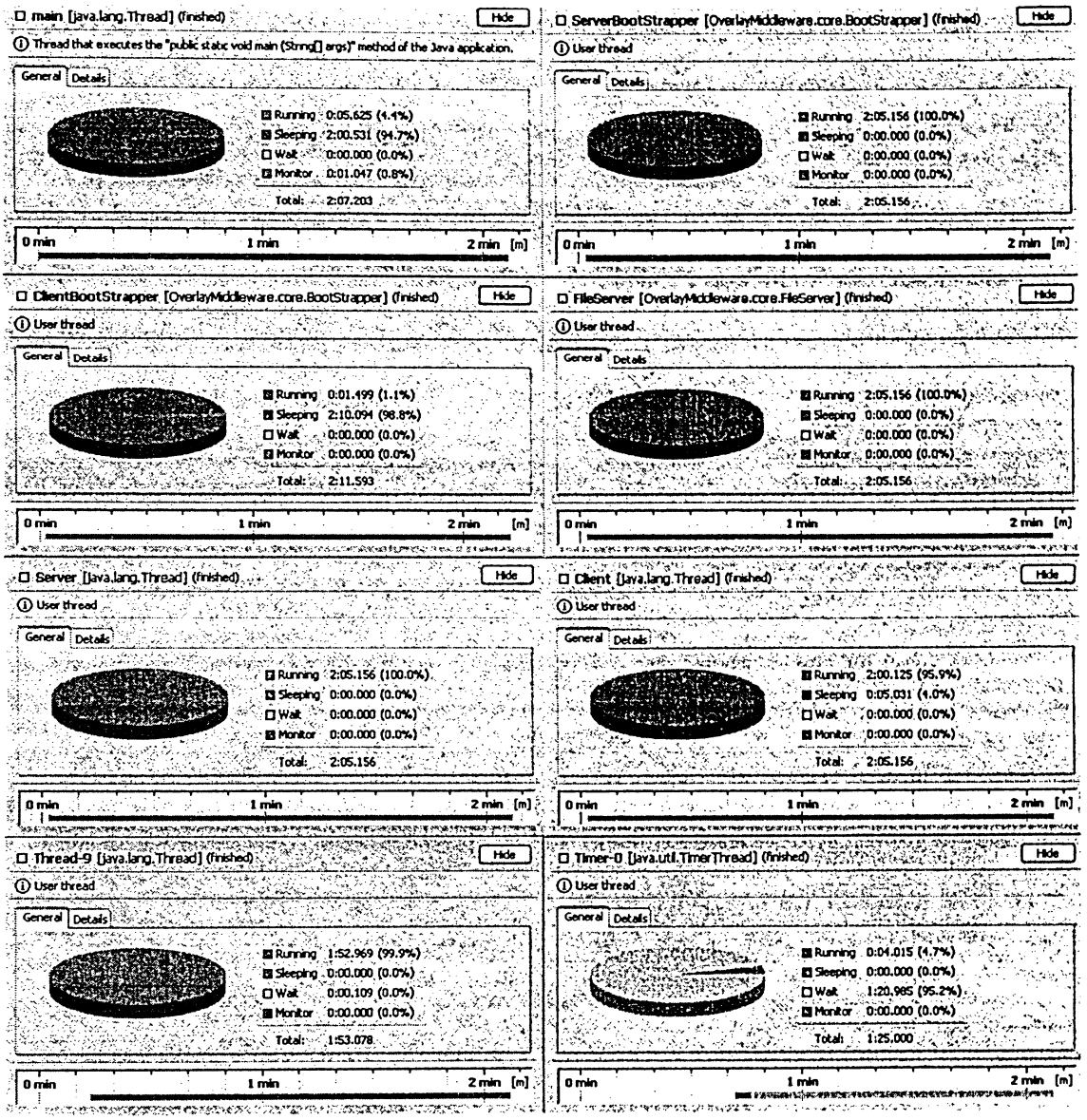

Figure 5.15 Threads Execution Details

The thread state graph of Timer-0 thread shows that it is waiting 95.2% of the time, and running for only 4.7%. The reason why a timer thread is used to send a query packet is to resend the packet if a reply is not received after a certain query timer expires. The query timer is configurable in Preferences.QueryTime member, which is set to 70 seconds as shown in figure 5.5. Since the call to Node.Query is issued 40 seconds after the application starts, and since the application only runs for approximately 120 seconds, this means that QuerySender.run() is executed twice, which in turn means that Timer-0 thread

state is changed to running state twice. The two executions of QuerySender.run() method only take 2093 milliseconds (approximately 0.4% of CPU time as shown in figure 5.14).

5.4.2 Memory

Class Name - Allocated Objects	Bytes Allocated	Bytes Allocated	Objects Allocated
OverlayMiddleware.util.FileInfo[]		416 B (1.9%)	1 (0%)
OverlayMiddleware.core.BootStrapper		128 B (0.6%)	2 (0.1%)
OverlayMiddleware.core.FileServer		104 B (0.5%)	1 (0%)
OverlayMiddleware.util.Neighbor		72 B (0.3%)	1 (0%)
OverlayMiddleware.AppSupport.Node		64 B (0.3%)	1 (0%)
OverlayMiddleware.core.Server		64 B (0.3%)	1 (0%)
OverlayMiddleware.core.Client		56 B (0.3%)	1 (0%)
OverlayMiddleware.util.CacheEntry[]		56 B (0.3%)	1 (0%)
OverlayMiddleware.util.CacheEntry		56 B (0.3%)	1 (0%)
OverlayMiddleware.core.QuerySender		56 B (0.3%)	1 (0%)
OverlayMiddleware.util.FileInfo		32 B (0.1%)	1 (0%)
OverlayMiddleware.AppSupport.Path		32 B (0.1%)	1 (0%)
OverlayMiddleware.util.NeighborDB		16 B (0.1%)	1 (0%)
OverlayMiddleware.util.FileDB		16 B (0.1%)	1 (0%)
OverlayMiddleware.util.QueryCache		16 B (0.1%)	1 (0%)
OverlayMiddleware.util.Peer[]		16 B (0.1%)	1 (0%)
OverlayMiddleware.util.PeerDB		16 B (0.1%)	1 (0%)
OverlayMiddleware.util.Peer		16 B (0.1%)	3 (0.1%)
OverlayMiddleware.util.Packet		16 B (0.1%)	3 (0.1%)
OverlayMiddleware.util.Neighbor[]		16 B (0.1%)	1 (0%)

Figure 5.16 : Memory allocated for OverlayMiddleware Objects

Table 5.1 : Memory allocated for Objects used by OverlayMiddleware

Class Name	Bytes Allocated	Objects Allocated
char[]	11,384 B (48.3%)	860(31.7%)
byte[]	2,808B (11.9%)	132(4.9%)
String	1,560B (6.6%)	591(21.8%)
TimerTask	528 B (2.2%)	1 (0%)
StringTokenizer	40B (0.2%)	9(0.3%)
DatagramPacket	32B (0.1%)	5(0.2%)
MulticastSocket	48B (0.2%)	2 (0.1%)
Socket	24B (0.1%)	4 (0.1%)
ServerSocket	24B (0.1%)	2(0.1%)
Timer	24B (0.1%)	1(0%)

Figure 5.16 displays a list of all OverlayMiddleware classes, with the total size and number of instances allocated. It can be seen that OverlayMiddleware.util.FileInfo[] is the biggest memory consumer among the other classes. FileInfo[] is an array of FileInfo objects that is allocated for the FileDB object. The size of the FileInfo array can be

increased or decreased by changing Preferences.FileDBsize as desired by the developer. The current setting is 100 elements as shown in figure 5.5.

Table 5.1 shows the bytes and objects allocated to be used by our middleware. Actually, char[] class, for which the JVM allocated around 11,300 bytes, is not used directly by our middleware, but it was used by J2ME API which our system used. The classes byte[], and String are two important types that our system used heavily as indicated by bytes and objects allocated in table 5.1. They were mainly used to create overlay packets. The rest of the classes listed in the table were used by different threads in our system and used up insignificant amount of memory.

Figure 5.17 : Memory (heap)

The overall memory that the sample application required for its execution is shown in figure 5.17. The lighter shade represents the overall memory that the JVM allocated for its heap (2 Megabytes in the graph). The heap is basically the amount of memory that is allocated dynamically during the program runtime. The darker shade in the graph

represents the amount of heap that is actually in use by the program, which stays close to 0.5 Megabytes. The amount of heap allocated and used does not include the JVM.

The amount of the heap being used by our system could slightly rise depending on its overlay state. For example, our middleware which was running in 10.1.1.1 formed a neighbor relationship with 10.1.1.3, and it was the client side of the connection. Had it been the server, one more Timer thread would have been allocated to perform LCS functionality. Moreover, the memory needed by our system could rise depending on the user application's requirements. If the application specified a higher number of neighbors, more threads will be spawned to manage the connections to these neighbors. The more threads our system spawns, the more CPU and memory resources will be used up. Therefore, it is important that the application specifies a maximum number of neighbors that does not overwhelm the capacity of the application's target device. The following equation estimates the number of the threads, t, spawned by our system:

$$t = (1 + c)\, n + k$$

Where c is the percentage of neighbors for which the node serves as a client for LCS. n is the number of neighbors, and k represents a constant number of threads used for administrative purposes.

Finally, figure 5.18 shows two important heap statistics [37]:

- The lower line is the percentage of execution time spent by the JVM doing garbage collection and is graphed against the y-axis on the right edge of the graph. Time spent by the JVM doing garbage collection is time that is not available for it to run the user application. Since the line is close to zero during the lifetime of the application, our middleware did not exhaust heap memory, which proves that our system is memory-efficient.
- The upper line is surviving generations and is graphed against the y-axis scale on the left edge of the graph. The count of surviving generations is the number of different ages of all the Java objects on the JVM's heap, where "age" is defined as

the number of garbage collections that an object has survived. When the value for surviving generations is low it indicates that most of the objects on the heap have been around about the same amount of time. If, however, the value for surviving generations is increasing at a high rate over time then it indicates that our middleware is allocating new objects while maintaining references to many of the older objects it already allocated. If those older objects are in fact no longer needed then the application is wasting (or "leaking") memory. In our middleware, the allocated resources are mainly the databases that are needed throughout the lifetime of the application. Therefore, our middleware does not leak memory.

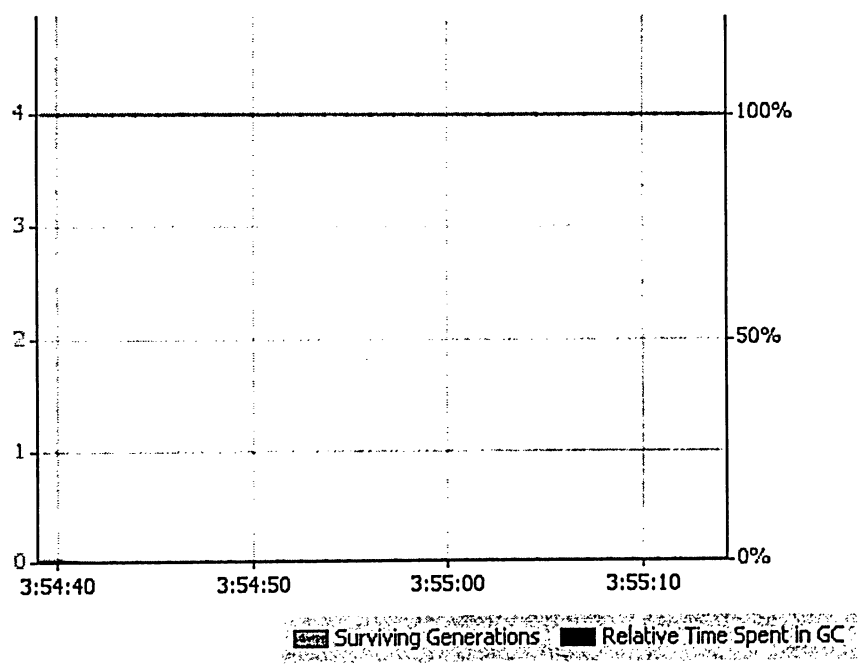

Figure 5.18 : Memory Garbage Collection

5.5 Testing on HP iPAQ 1950

Because our middleware was developed in accordance with the J2ME CDC, Personal Profile 1.0 specifications, and since CDC personal profile is supported by most Personal Digital Assistants (PDAs) and many mobile phones, we decided to experiment executing it on the Hewlett Packard PDA, iPAQ 1950 in order to prove that we have achieved our portability requirement.

HP iPAQ 1950 is a pocket pc powered by windows mobile 5.0 Premium Edition. It is based on new Samsung SC32442 processor which runs at a speed of 300 MHz. It has 64 MB ROM and 32MB SDRAM (up to 33 MB user available persistent storage memory). It is also equipped with a WiFi 802.11b connectivity.

In order to be able to run our java program on the PDA, a J2ME run time environment was installed first. We have installed IBM's J9 runtime environment [39], which is an implementation of the J2ME platform and is composed of a JVM and some Java Class Libraries (JCL). Microsoft's ActiveSync [40] program was used to launch J9 on the iPAQ.

Because print statements can no longer be used to trace the execution of our program on the PDA, events were saved on a log file. IBM's Websphere Studio Device Developer [41] IDE was used to generate a Java Archive (JAR) file of the application. The JAR file was then ported to the PDA, and was tested for execution. To test our middleware on the PDA, we carried out the same demonstration application of section 5.3. Only this time, the PDA was given the role of 10.1.1.1 in the previous experiment and its execution was no different from the laptop.

5.6 Implementation Limitations

One limitation encountered while using J2ME platform is that it does not support any API that can retrieve local host device information, such as current battery or power level, or any CPU information. To overcome this problem, two methods were added to Node interface, which are SetPower(), and SetCapacity(), which can be used to set the power and capacity values for the node respectively.

At the time of development, the user might have an idea which platform is being targeted. In that case, platform specific native API can be used in order to read the power and capacity level of the node and feed it to our system. Another get around that the

developer can do is to use PersonalJava platform which supports some API that can retrieve battery level and capacity. We have not used PersonalJava because it has reached its end-of-life-cycle which means that it will no longer be supported in next generation java-enabled devices.

5.7 Performance Optimization using AODV

In section 3.2.4, we have compared between flooding and random walk techniques as resource discovery techniques for MANETs. We have chosen to employ random walks rather than flooding because flooding generates excessive traffic, and because biased random walks can find the best path based in link stability and neighbor capacity, which suits MANET nodes properties.

However, in order to perform MBR, our system generates some overhead traffic. For example, nodes exchange LCS packets in order to measure the link stability to their neighbors. A question that rises itself is whether we have achieved what we are hoping to achieve when we favored biased random walks over flooding. One might think that instead of generating LCS traffic, we can simply flood the network with queries, which might have the same overhead, if not less. We can argue that we can achieve better results than that of flooding, in terms of decreasing network load, if our system was used on top of Ad hoc On Demand Distance Vector AODV routing protocol [31].

AODV is a reactive on-demand routing protocol designed for ad hoc mobile networks. It discovers routes when the node needs to send the data packets to an unknown destination. Hello messages are used to detect and monitor links to neighbors. If Hello messages are used, each active node periodically broadcasts a Hello message that all its neighbors receive. Because nodes periodically send Hello messages, if a node fails to receive several Hello messages from a neighbor, a link break is detected [31].

A node wishing to discover a new route broadcasts a route request (RREQ) to all of its neighbors. The receiver of a RREQ searches its cache for a match of the request. If it

finds a hit, it unicasts a route reply (RREP) to its next hop in the direction of the source; otherwise, it continues to broadcast the RREQ after saving the return route to the source in the cache. Each node that propagates a RREP saves the route to the destination. After the source receives the RREP, it begins sending data. AODV maintains routes for as long as they are active [33]. This means that data packets destined to a specific route will refresh the route in the AODV cache.

How does AODV help optimize the performance of our system? If we set our LCS and multicast timers to be less than of the AODV cache expiry timer, then our LCS and multicast traffic will refresh the routes in AODV cache and will save the network from RREQ packets that AODV floods the network with when it needs to discover a route.

On the other hand, if we consider the scenario of query flooding, the ad hoc network does not only get flooded at the overlay-level, but it also gets flooded with AODV RREQ that some nodes send to rediscover routes to their neighbors , since queries are issued by the application in at unpredictable times.

Performance of our system can be further improved if it can get the RTT and power information of the overlay neighbors from the underlying routing protocol. For example, some routing protocols are power-aware. Because we prefer not to constraint our middleware with such routing protocols and because it is important for our middleware to be independent of underlying layers, the LCS functionality was fully implemented at the overlay layer.

5.8 OverlayMiddleware and KBR

In section 2.2, we summarized the work of [7] which introduces a common unified API for structured overlay networks. Despite the fact that their work targets a different class of overlay networks than ours, some methods in our middleware implement similar functions that their API suggests, as shown in table 5.2.

Table 5.2 : KBR Functions Implemented by OverlayMiddleware

KBR function	OverlayMiddleware method
route(), forward()	Implemented by our ConnManager.Forward() method
deliver()	Implemented by MessageHandler.ProcessMessage()
locallookup()	Implemented by our NeighborDB.ReturnCandidates()

6 Concluding Remarks and Future Work

Because of the flourishing market of high-end mobile devices, MANET have gained an enormous popularity over the past few years, which led to an increasing need for development of applications to run on these devices. Developing MANET application, however, is not an easy task. It faces challenges due to network dynamics including ad hoc and unstable connectivity, abrupt disconnection; and due to resource constraints including limited power and computing resources.

Solutions such as, overlay networks or middleware have been suggested to facilitate software development over MANETs. Middleware usually refers to a layer that acts an intermediate layer between different application components, providing a more functional set of Application Programming Interfaces (API) than the underlying platform. An overlay network, on the other hand, is basically a peer-to-peer virtual network that is logically built on top of the physical topology. Both structured and unstructured overlay networks have shown a promising approach for designing and deploying applications over MANET.

In this thesis, the design of a middleware that abstracts the functionalities of an unstructured overlay network is proposed. The most important functionalities that our middleware supports include the following:

- Bootstrapping or joining the overlay network: multicast is used so that overlay peers discover each other.
- Neighbor establishment: TCP connections are established between overlay peers simultaneously, where each neighbor is assigned a separate thread to monitor the connection and to forward the overlay traffic coming from that neighbor.
- Resource discovery: queries and query replies are used to discover a resource. The forwarding decision of each node is based on MBR.
- Path establishment: during the overlay query forwarding phase, each hop caches the next and previous hops in order to establish the path and the reverse path between source and destination.

- File download: A simple TCP file server is implemented.

The design of the middleware is influenced by the fact that a MANET is formed out of resource-challenged devices of different platforms. In addition, our middleware is designed so that it supports the implementation of a wide range of applications. These facts mainly indicate that our middleware has to meet three key non- functional requirements, which are namely, portability, efficiency and usability.

Firstly, portability is achieved by our middleware because it is implemented according to the J2ME CDC platform specifications, which allows it to run on several platforms. Secondly, using java threads allows us to achieve speed efficiency. Memory efficiency is taken care of by JVM because of its garbage collection facility. Finally, usability is achieved by implementing the simple MBR-based file-search and path establishment mechanisms which we have proved to be the basis for implementing a wide range of applications, such as file sharing, ALM and gaming applications.

There are some future improvements that can be incorporated into our middleware. One improvement that would be desirable for our system is to implement a proactive neighbor replacement algorithm based on link RTTs. Selecting neighbors based on RTT can ensure that each node will remain connected to its physically closest nodes, which can help build a more locality-aware topology. The reason is that links to physically closest nodes are more likely to have shorter RTTs. This can be advantageous because if overlay neighbors are also physical neighbors, then messages sent between neighbors will be reduced. It is true that at the overlay level, a packet from a node to its neighbor is one hop away, but in reality, at the transport layer, this neighbor could be multiple hops away.

One hop replication might be considered as a future improvement as well. If each node keeps track of the files that exist at each of its neighbors, the overall systems performance can be improved. The reason is that it increases the number of nodes that know where a file is located, which increases the query success rate. Actually, exchanging file names should not introduce more traffic in the network, because such information can be

piggybacked on other overlay packets. However, there are some issues that should be studied first before introducing one hop replication to the system. For example, if each overlay node were to keep track of the file indices of all of its neighbors, this might exhaust the memory of the node.

One important future improvement is securing the overlay network against malicious nodes. For example, a malicious node can listen to the overlay joins that overlay nodes send during bootstrapping. The malicious node can then use the information in the join packet, such as the IP, in order to connect to the server port. This can cause a denial-of-service attack. A malicious node can also attack the TCP file server. Therefore, a thorough investigation on overlay middleware must be made in order to guard against such attacks.

Finally, one improvement to the system is to use a credit-based flow control such as the one used in Gia. In Gia's flow control algorithm, a node can only send a query to its neighbor if it has credits or tokens from that neighbor. This approach guards against the problem of hot spots which could occur because each node takes capacity into consideration when forwarding a query.

References

[1] Doval, D ,O'Mahony, D, "Overlay Networks: A scalable alternative for P2P," IEEE Internet computing, jul-aug 2003.

[2] K. Lua, J. Crowcroft, M. Pias, R. Sharma and S. Lim. "A Survey and Comparison of Peer-to-Peer Overlay Network Schemes,"in *IEEE communications survey and tutorial*, March 2004.

[3] F. Araujo, J Kaiser,C. Mitidieri, C Liu and L. Rodrigues. "CHR: a Distributed Hash Table for Wireless Ad Hoc Networks," in *25th international conference on distributed computing schemes*, Columbus, Ohio, USA, June 2005.

[4] R. Harbird, S. Hailes and C. Mascolo. "Adaptive Resource Discovery for Ubiquitous Computing," in *2nd workshop on Middleware for Pervasive and Ad-Hoc Computing*, Toronto, Canada.

[5] G. Lau, M. Jaseemuddin, and G. Ravindran "RAON: A P2P Network for MANET," *Proceedings of 2nd IEEE/IFIP International Conference on Wireless and Optical Communications Network (WOCN)*, March 2005.

[6] Y. Chawathe, S. Ratnasamy,L. Breslau, N. Lanham and S. Shenker. "Making Gnutella-like P2P Systems Scalable," in *Proceedings of SIGCOMM 2003*, August 2003.

[7] F. Dabek, B. Zhao, P. Druschel, J Kubiatowicz and I. Stoica. "Towards a Common API for Structured Peer-to-Peer Overlays", *Second International Workshop on Peer-to-Peer Systems (IPTPS 2003)*, February 2003.

[8] I. Chlamtac, M. Conti, and J. Liu, "Mobile Ad Hoc Networking: Imperatives and Challenges," *Ad Hoc Networks,* vol. 1, no. 1, 2003, pp. 13-64.

[9] Antony Rowstron and Peter Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," in *Proceedings of the IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content Addressable Network," in *Processings of the ACM SIGCOMM*, 2001, pp. 161–172.

[11] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17– 32, 2003.

[12] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A Resilient Global-Scale Overlay for Service Deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, January 2004.

[13] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. "Freenet: A Distributed Anonymous Information Storage and Retrieval System," In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, 2000.

[14] B. Cohen, "Incentives Build Robustness in BitTorrent," May 22, 2003. http://www.bittorrent.org/bittorrentecon.pdf

[15] Gnutella Development Forum, "The Gnutella v0.6 Protocol," http://www.gnutellaforums.com/

[16] Napster, http://www.napster.com/

[17] D. Bakken, "Middleware," Chapter in *Encyclopedia of Distributed Computing,* J. Urban and P. Dasgupta, eds., Kluwer Academic Publishers, 2002, http://www.eecs.wsu.edu/~bakken/middleware.pdf.

[18] S. Nilsen, "A CORBA Service for the OSA+ Real-Time Middleware," M.S. thesis, University of Oslo, October 2005.

[19] H. Pinus, "Middleware: Past and Present a Comparison," http://www.st.informatik.tu-darmstadt.de/database/seminars/data/middleware.pdf?id=79 June 2004

[20] C. Mascolo, L. Capra, and W. Emmerich, "Middleware for Mobile Computing (A Survey)," *Networking 2002 Tutorial Papers,* 2002.

[21] A. Gaddah and T. Kunz. "A Survey of Middleware Paradigms for Mobile Computing," Technical Report SCE-03-16, Carleton University Systems and Computing Engeneering, July 2003.

[22] E. Vollset, "Extending an Enterprise Messaging System to Support Mobile Devices," M.S. thesis, University of Newcastle upon Tyne, September 2002

[23] Sun Microsystems, 1995-2006. "The Java Tutorials: Concurrency, Processes and Threads," http://java.sun.com/docs/books/tutorial/essential/concurrency/procthread.html

[24] S. Oaks and H.Wong, Java Threads, O'reilly, January 1997

[25] I. Sommerville, Software Engineering, Addison-Wesley Publishing Company, 5th edition, chapter 12.

[26] M. Castro, P.Druschel, A. Kermarrec, A. Rowstron. "SCRIBE: A Large-Scale and Decentralized Application-Level Multicast Infrastructure".in *IEEE Journal on selected areas in communications*, Vol. 20, NO. 8, October 2002

[27] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Application-Level Multicast using Content-Addressable Networks," in *Proceedings of the Third International Workshop on Networked Group Communication*, London, UK, Nov. 2001, pp. 14–29.

[28] S. Kandula, J. Lee, and J. Hou."LARK: A Light-weight, Resilient Application-Level Multicast Protocol," in *Proceedings of IEEE Computer Communication Workshop*, November 2003.

[29] P.Engelstd, G. Egeland, S. Bygdas,R. Geers and T. Urnes. "Middleware Supporting Adaptive Services in On-Demand Ad hoc Networks,"in *Proceedings of 9th International Conference on Intelligence in service delivery Networks (ICIN'2004)*, Bordeaux (France), October 18-21, 2004.

[30] Q. Lv, P. Cao, E. Cohen, K. Li and S. Shenker. "Search and Replication in Unstructured Peer-to-Peer Networks," in *Proceedings of the 16th international conference on Supercomputing*, pp. 84-95, 2002.

[31] RFC3561, Ad hoc on Demand Distance Vector Routing, July 2003. http://www.ietf.org/rfc/rfc3561.txt

[32] C. Perkins, E. Belding-Royer, and Ian Chakeres. "Ad Hoc On Demand Distance Vector (AODV) Routing," In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, New Orleans, LA, February 1999, pp. 90-100.

[33] Mobility Management and Networking (MOMENT) Laboratory, "AODV," http://moment.cs.ucsb.edu/AODV/aodv.html.

[34] Tool Report: NetBeans Profiler, January 2007, http://www.javaperformancetuning.com/tools/netbeansprofiler/.

[35] NetBeans Profiler, http://profiler.netbeans.org/.

[36] NetBeans IDE, http://www.netbeans.org/products/ide/.

[37] G. Sporar and R. Kusterer. "NeatBeans IDE tutorial". http://www.netbeans.org/kb/articles/nb-profiler-tutor-8.html.

[38] Ethereal, www.ethereal.com.

[39] J9 IBM JVM for PocketPC PDA's. http://www.berka.name/stan/jvm-ppc/little_j9_howto.html.

[40] Microsoft's ActiveSync 4.5 http://www.microsoft.com/windowsmobile/activesync/activesync45.mspx.

[41]Websphere Studio Device Developer www.ibm.com/software/wireless/wsdd/.

[42] "Link State Routing Protocol – OSPF," class notes for CN 8813, Department of Electrical and Computer Engineering, Ryerson University, Fall 2006.

[43] M. Abolhasan, T. Wysocki, and E. Dutkiewicz, "A review of routing protocols for mobile ad hoc networks," Ad Hoc Networks, vol.2, no.1, pp.1–22, Jan. 2004.

[44] M. Bhuiyan, and M. Jaseemuddin, Congestion-Aware Overlay Networks, accepted to appear *in Proceedings of the 64th IEEE Vehicular Technology Conference Fall 2006 (VTC)*, Sept. 25-28, 2006 Montreal, Canada.