Prototyping a Peer-to-Peer Session Initiation Protocol User Agent

Submitted in fulfilment
of the requirements of the degree
Masters in Computer Science
at Rhodes University

Mosiuoa Tsietsi

March 10, 2008

Abstract

The Session Initiation Protocol (SIP) has in recent years become a popular protocol for the exchange of text, voice and video over IP networks. This thesis proposes the use of a class of structured peer to peer protocols - commonly known as Distributed Hash Tables (DHTs) - to provide a SIP overlay with services such as end-point location management and message relay, in the absence of traditional, centralised resources such as SIP proxies and registrars. A peer-to-peer layer named OverCord, which allows the interaction with any specific DHT protocol via the use of appropriate plug-ins, was designed, implemented and tested. This layer was then incorporated into a SIP user agent distributed by NIST (National Institute of Standards and Technology, USA). The modified user agent is capable of reliably establishing text, audio and video communication with similarly modified agents (peers) as well as conventional, centralized SIP overlays.

Acknowledgements

This thesis is dedicated to my family who have been with me through thick and thin, who believed in me when times were rough and tough. I am grateful for the support they gave me not just over the last two years, but all through my life. You mean a lot to me.

I would like to thank my supervisors, Alfredo Terzoli and George Wells for their tireless effort and patience when I needed it. This thesis is testimony to the quality of their stewardship and ability to motivate students to do their best, and dig just a little bit deeper.

I would also like to thank the staff members and students of the Department of Computer Science - thank you all.

Finally, thank you to my sponsors: Telkom SA, Business Connexion, Comverse, Verso Technologies, THRIP, Stortech, Tellabs, Mars Technologies, Amatole Telecommunication Services, Bright Ideas 39 and the National Research Foundation.

Contents

List of Figures

List of Tables

Glossary Of Terms

AOR: Address of Record.

API: Application Programming Interface.

CAN: Content Addressable Network.

DDNS: Dynamic DNS.

DHCP: Dynamic Host Control Protocol.

DHT: Distributed Hash Table.

DNS: Domain Name System.

DNS NAPTR: Domain Name System Naming Authority Pointer.

DNS SRV: Domain Name System Service Record.

HTTP: HyperText Transfer Protocol.

IANA: Internet Assigned Numbers Authority.

IETF: Internet Engineering Task Force.

IM: Instant Messaging.

IP: Internet Protocol.

IPv4: Internet Protocol version 4.

IRIS: Infrastructure for Resilient Internet Systems.

ISP: Internet Service Provider.

JAIN: Java APIs for Intelligent Networks.

JCP: Java Community Process.

JSAP: JAIN SIP Applet Phone.

JSR: Java Specification Request.

KBR: Key Based Routing.

LAN: Local Area Network.

LDAP: Lightweight Directory Access Protocol.

MEGACO: Media Gateway Control Protocol.

NAT: Network Address Translator.

NIST: National Insititute of Science and Technology.

Node ID: Node Identifier.

P2P: Peer to Peer.

P2P SIP: Peer to Peer Session Initiation Protocol.

pCAN: Passive CAN.

PIDF: Portable Information Document Format.

Resource ID: Resource Identifier.

RFC: Request For Comments.

RTP: Realtime Transport Protocol.

RTSP: Realtime Streaming Protocol.

SDP: Session Description Protocol.

SEDA: Sandstorm Event Driven Architecture.

SER: Sip Express Router.

SIP: Session Initiation Protocol.

TU: Transaction User.

UA: User Agent.

UAC: User Agent Client.

UAS: User Agent Server.

URI: Uniform Resource Identifier.

URL: Uniform Resource Locator.

VoIP: Voice over Internet Protocol.

XPP: eXtensible Peer Protocol.

Chapter 1

Introduction

> *Begin at the beginning ... and go on till you come to the end: then stop.*
>
> - Lewis Carroll, Alice's Adventures in Wonderland, 1865

The growth of the Internet, and IP networks in general, has seen improvements and increased diversity in the possible mediums of communication. Traditionally, technologies such as email and the web have been the predominant means of information exchange. At present, newer forms of communication such as Internet telephony, multimedia conferencing and multimedia streaming are fast becoming commonplace as services that can be delivered to users in realtime as opposed to the classical store and forward method.

The Session Initiation Protocol (SIP) is a popular choice for the delivery of realtime services over IP networks. In most settings, SIP is deployed using one or more central servers which perform dedicated tasks on behalf of users. While this setup has proven effective, there are many real-world scenarios where it would be beneficial or even necessary to provide the means of establishing point-to-point communication between endpoints themselves. The resulting architecture is known as Peer-to-Peer SIP (P2P SIP). The standardisation of the protocols that will be used to enable P2P SIP is currently work in progress in the Internet Engineering Task Force (IETF), where a working group has been established. A summary of their contributions is the subject of chapter 4.

This thesis looks at using peer-to-peer protocols as tools for the creation of decentralised networks upon which a telecommunication overlay based on SIP can be constructed. The SIP overlay consists of SIP-enabled endpoints that perform traditional server functions in a cooperative manner. This chapter serves as an introduction to the area of study, starting with a non-exhaustive

enumeration of the possible usage scenarios for decentralised communications, followed by an account of the services that must be provided by a decentralised SIP network. The class of peer-to-peer protocols that were selected to achieve this purpose is presented, followed by an outline of the objectives of the research. Finally, this chapter details the steps that led to the development of a decentralised framework for P2P SIP and gives the scope of the project.

1.1 Usage scenarios for P2P SIP

The IETF working group on P2P SIP released an Internet draft that classified the possible scenarios for the usage of P2P SIP [1] into four main groupings, which are described below:

1. Global Internet Environment - P2P SIP can be used as a means of creating a global, openly available VOIP network with no (or little) central administration.
2. Security Demanding Environments - For security or privacy reasons, users or organisations may not want their data to traverse the infrastructure of service providers or may not be allowed to use the infrastructure, thus making a peer-to-peer solution more attractive.
3. Environments with Limited Internet Connectivity and Infrastructure - For environments such as the developing world, ad-hoc and ephemeral groups or disaster situations, where Internet connectivity may not be present or may be unreliable, a local and inexpensive self-configuring solution would be necessary.
4. Managed Private Network Environments - For the purpose of cost reduction or improved scalability of VOIP systems, organisations may decide to complement their client-server based systems with a peer-to-peer solution.

1.2 Services offered in a SIP network

If one considers what is needed in order for a simple conversation between two SIP endpoints (called user agents) to occur, it is possible to understand how the centralised theme permeates SIP networks. Firstly, each user agent must initiate the registration process which ultimately provides the network with information that allows it to make an association between the user's identity and the user's locality. This often means that each user agent must be pre-configured to interact with

a specific SIP server, typically a SIP proxy, which knows the location of another server called a registrar, which authenticates them. The registrar, upon receipt of a registration request, will compare the credentials supplied by the clients with those stored in a central database (which can be hosted by the registrar itself or by another server). The database is called a ”location service” because it associates a user’s SIP address with a locality record which points to the actual network location of the user’s device. The SIP proxy server assists in facilitating the exchange of SIP messages (a process called signalling) between the endpoints. Once the endpoints are successfully registered and the signalling process has succeeded, then media streams can be established in both directions.

The processes described above are underpinned by two main activities. Firstly, there is the administrative effort which goes into running a SIP network. This is reflected in the need for the configuration of both SIP clients and servers, and the setup and maintenance of a central database system. Secondly, SIP relies on a naming service based on domains which is usually provided by an implementation of the Domain Name System (DNS).

It is possible to enumerate the services provided by a SIP network. These services are not discarded by a peer-to-peer solution, but must be implemented in a distributed manner. These services are:

Proxy A service which is able to handle requests on the behalf of others.

Registration A service which accepts registration requests from user agents.

Location Service A database of locality mappings crucial for session establishment.

Administration It must be possible to perform actions such as create new networks and add new users.

1.3 Decentralised protocols for P2P SIP

In order to perform the services listed above in a decentralised manner, a logical choice is to borrow from the realm of peer-to-peer protocols. By nature, peer-to-peer systems are able to pool the bandwidth, processing power and storage capabilities of its constituent nodes. In addition, it is typical for these systems to be able to sustain and coordinate themselves with little or no external supervision, hence their popularity in recent years as evidenced by such systems as KaZaA [2] and BitTorrent [3].

Peer-to-peer protocols can be roughly divided into two classes, structured and unstructured. Structured protocols place certain constraints on the node graph so that searches for resources stored in the network follow a deterministic path. Unstructured protocols impose no such constraints, allowing nodes to be organised in a random graph where there are random joining processes and neighbour selection algorithms.

It would seem that interest has been growing in a class of structured peer-to-peer protocols commonly known as Distributed Hash Tables (DHTs). The name is derived from their ability to provide a decentralised key-value based lookup service, which is supported by a structured set of node links. In this thesis, the term DHT is used in reference to both the protocols as well as the hash table abstraction they provide.

DHTs have been identified by the research community working on protocols for P2P SIP as a possible platform for providing the distributed location service for SIP. However, when the work towards this thesis commenced, it was not clear that DHTs would become part of the standard.

1.4 Objectives

DHTs may satisfy the requirements presented in section 1.2, but there are a number of implementation challenges with their use. Firstly, there are many DHT protocols in existence today [4, 5, 6, 7]. It would be beneficial to provide the means for any DHT to be used so that a P2P SIP client does not have to be molded to use a single, must-implement DHT. Secondly, while all DHTs provide a similar basic lookup service, there are some differences in how they provide this service and in the way they expect to interact with the applications which make use of them.

Given these points, the overall objectives presented in this thesis are:

1. To investigate the use of peer-to-peer protocols, and in particular DHTs, for use with SIP.
2. To provide a DHT-agnostic method for many DHT modules to be plugged into a single P2P SIP system.
3. To investigate the possibility of providing interoperation between heterogeneous overlays.
4. To provide a solution for decentralised overlays to interoperate with conventional SIP networks.

1.5 Method

An applied experimental approach was followed in this research, that involved the development of a peer-to-peer framework followed by its incorporation into a SIP user agent. The framework was named OverCord. The name consists has two distinct parts: over and cord, which refer to the system's ability to support many types of overlays in a similar way to an electrical extension cord that supports many devices.

1.5.1 Identification of open source DHT implementations

An investigation into existing DHT implementations revealed that there is much activity in this area, and it was possible to access stable, open source code largely in C/C++ and Java. This investigation was done in parallel with the one described in section 1.5.3 below, which covers the identification of an open source SIP user agent. This led to the selection of the Java based implementations, from which two DHTs called OpenChord and Bamboo were chosen. The two DHTs are discussed in detail in section 6.3.1 and section 6.3.2 respectively.

1.5.2 Development of a generic interface

DHTs commonly export an application programming interface (API) - a software interface composed of functions - through which an external application can control it. The functions necessary for P2P SIP would include those for message routing as well as for the lookup, insertion, update and removal of resources from a location service. To achieve the objective of DHT pluggability, given that each DHT exposes a different API, it would be beneficial to create a single, generic interface which could be used across all DHTs that the application has access to. This means that a middleware layer needs to be implemented that is able to translate the generic API methods to the proprietary APIs of each DHT. This middleware layer was implemented via the creation of software ”wrappers” which perform the translation. In OverCord, these wrappers are called plug-ins. After the two DHTs had been selected, plug-ins were developed for each one. The generic interface was tested and found to be successful in interacting with the plug-ins.

1.5.3 Identification of an open source SIP user agent

Since the objective of the research was to construct a peer-to-peer layer for currently existing SIP user agents, it was necessary to identify an open source application into which such a layer could be inserted. The investigation identified the JAIN SIP Applet Phone [8] which uses a Java based SIP stack known as JAIN (Java APIs for Intelligent Networks). The result of the incorporation was tested and proved successful.

1.5.4 Interoperation with conventional SIP networks

Having produced a peer-to-peer layer for SIP and tested its ability to support communication across heterogeneous overlays, it seemed interesting to try to provide similar support for interoperating with conventional SIP networks. It became evident that an asymmetry existed: it would be more difficult to provide communication from centralised networks to decentralised networks than the other way around. The difficulty arises from the fact that most centralised networks rely on DNS to calculate the next hop destination for messages. Though it is possible to create DNS resource records for elements in peer-to-peer overlays, conventional DNS is unsuitable for peer-to-peer environments due to the high churn rates, which would require unusually frequent DNS record updates. Dynamic DNS (DDNS) was identified as an appropriate alternative, since it supports dynamic updates and low TTL (time-to-live) periods for resource records. A node in the overlay would thus be able to play the role of an ephemeral proxy for incoming and outgoing messages.

1.6 Scope of research

The main focus of the research is to investigate the requirements of a decentralised framework for SIP and to produce a prototype application that shows how this framework can be used to support realtime communication using SIP.

The scope of the research explicitly excludes the following:

1. Proving why P2P SIP may be better than centralised SIP.
2. Making conclusions about which class of peer-to-peer protocols is best suited for P2P SIP.

3. Considerations for enabling communications in environments where some or all of the endpoints are behind Network Address Translators (NATs).

4. Security concerns which include issues such as the secure allocation of unique user credentials, user authentication and preventing malicious attacks on either stored data or the routing infrastructure of the overlay.

1.7 Document overview

The rest of the thesis is organised as follows:

Chapter 2 This chapter examines the SIP protocol within the limits of what is necessary to understand the rest of the thesis. It defines the different roles and message types that the protocol uses and explains how SIP supports multimedia services. The chapter uses the description of SIP to make the point that SIP is naturally supportive of distributing server roles among overlay nodes. The next chapter explores the different peer-to-peer protocols that can help perform this decentralisation.

Chapter 3 This chapter gives a taxonomy of peer-to-peer protocols, as an initial step towards identifying an appropriate platform for decentralising SIP. This chapter divides peer-to-peer protocols into two classes: structured and unstructured. Examples of each class are given in order to identify common characteristics and lastly, a comparison between the two is made which evaluates the suitability of each to provide the services needed for SIP.

Chapter 4 This chapter summarises the protocol proposals and analyses the prototype implementations of the recently established P2P SIP working group in the IETF. Some of the early work is described as well as the newer and more mature concepts that are likely to feature in the final standard.

Chapter 5 This chapter details the design of a peer-to-peer architecture called OverCord. The design is multi-layered, modular and supports the plugging in of new modules. The architecture can be used to construct user agents for SIP, but due to the separation between application and service, it is anticipated that it can be used for other purposes beyond P2P SIP.

Chapter 6 This chapter describes a possible implementation of the OverCord system. The implementation is based on implementations of DHTs and shows how OverCord, in isolation from any service application, can be used to create decentralised overlays.

Chapter 7 This chapter describes the process of incorporating OverCord into a Java based SIP client called the JAIN SIP Applet Phone. The different options that were available when choosing an appropriate environment for performing this incorporation, namely the SIP stack and client application, are outlined and justification is given for the successful candidates. The steps that were followed in performing the incorporation are detailed, giving insight into the different portions of the client that needed to be reworked in order to perform the integration. The results of test are given at the end of the chapter which show SIP clients communicating not only within the same overlay, but with heterogeneous overlays.

Chapter 8 This chapter deals with the interoperation of overlays based on peer-to-peer protocols and conventional SIP networks, based on SIP servers and DNS. Possible methods to achieve this are given and critiqued, followed by a discussion on the method that was employed in this research. The proposed solution is based on dynamic updates to DNS records which are more suitable for chaotic, decentralised environments characterised by high churn rates.

Chapter 9 This chapter concludes the thesis, summarising the work done and the significance of the contribution towards the development of P2P SIP. Some comments on the opportunities for future work on the design and implementation are also given.

Chapter 2

Session Initiation Protocol

Our life is frittered away by detail ... Simplify, simplify.

- Henry David Thoreau, Walden, 1854

The Session Initiation Protocol (SIP) is an application layer protocol standardised by the Internet Engineering Task Force (IETF) which is used for creating, modifying and terminating sessions such as Voice over Internet Protocol (VoIP) and multimedia conferences [9]. Intended as a member of a larger multimedia framework, SIP was designed with the native ability to incorporate a range of IETF protocols such as Realtime Transport Protocol (RTP) [10], Realtime Streaming Protocol (RTSP) [11] and Media Gateway Control Protocol (MEGACO) [12].

In most deployment environments, a SIP network is supported by a server, or servers, which perform certain functions for the network. Clients are largely dependent on the servers, without which they would have difficulty communicating with, and locating each other. Despite this centralised viewpoint of SIP, many aspects of the protocol are essentially peer-to-peer, such as audio and video conversations between SIP clients, which do not normally involve the server. By definition, SIP also fosters an abstract association between entity and function, providing an opportunity for services to be provided in a distributed and cooperative manner. This chapter does not give a thorough overview of SIP, for which the reader may refer to the protocol standard [9] or other literature such as [13, 14]. However, this chapter does define terminology and concepts required to understand the identity and function of SIP entities, and the multimedia services they are able to support. In addition, it highlights the inherent opportunities that SIP provides for decentralising services without violating the syntax or semantics of the protocol, by using techniques to distribute the services over a number of participating endpoints.

2.1 SIP entities

SIP entities are divided into two groups, SIP endpoints (called user agents) and SIP servers, which are described in the next two sections.

2.1.1 User Agents

SIP, like the Internet Protocol (IP) upon which it is based [15, 16], is itself a layered protocol consisting of layers that define the behaviour of all SIP entities. Its architecture is depicted in Figure 2.1. The lowest layer defines the message structure and grammar of the protocol. The next layer is the transport layer, which defines how an entity sends requests and receives responses as a client, and how it receives requests and sends responses as a server. The transaction layer is built above the transport layer and defines how an entity handles a SIP transaction. In SIP, a transaction encompasses a request sent by a client and all subsequent responses to that request by a server. The topmost layer is the transaction user (TU) layer. This layer ties all the lower layers together in that it is responsible for creating requests through the transaction layer. All SIP entities conform to this model, except for the stateless SIP proxy discussed in section 2.3, which only differs in that it does not have a TU layer.

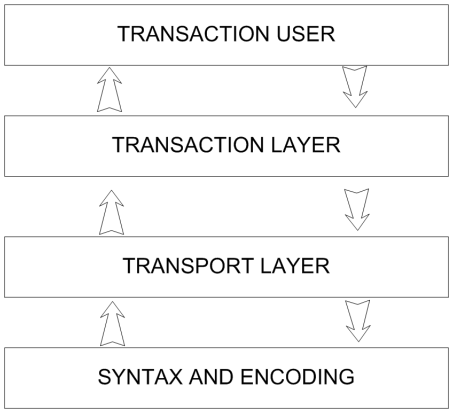

Figure 2.1: SIP - A layered protocol. Derived from [9].

Endpoints in SIP are known as user agents (UAs). A UA is an example of a TU and is host to two discrete components, a User Agent Client (UAC) and a User Agent Server (UAS). The UAC

is capable of creating requests and a UAS is capable of generating responses to requests. Some common SIP request messages are given in Table 2.1. Table 2.2 defines the classes of response codes that are generated to SIP requests.

Message Name	Meaning
REGISTER	Used by a user agent to notify the SIP network of it current Contact URI and the URI that should have requests routed to the Contact.
INVITE	Used to establish media sessions between user agents.
MESSAGE	Used to carry instant message content.
PUBLISH	Used to publish event state and distribute it to interested parties.
SUBSCRIBE	Used to request asynchronous notification of an event or set of events at a later time.
NOTIFY	Used to notify a SIP entity that an event which has been requested by an earlier SUBSCRIBE method has occurred. It may also provide further details about the event.
OPTIONS	Used to indicate user agent capabilities.
CANCEL	Used for canceling a pending request.
ACK	Used to indicate acknowledgment of the reception of a message.
BYE	Used to terminate a session.

Table 2.1: A subset of the SIP request messages in SIP.

Class	Description	Action
1xx	Informational	Also known as provisional - the server is performing an action but does not yet have a definitive response.
2xx	Success	Request was successful.
3xx	Redirection	Gives information about the user's new location, or about alternative servers that might be able to satisfy the call.
4xx	Client error	The client should not retry the request without modification (for example, adding appropriate authorisation). However, the request may succeed at a different server.
5xx	Server failure	The server has erred.
6xx	Global failure	The server has definitive information about a particular user, not just the particular instance indicated in the Request URI.

Table 2.2: SIP response codes. Derived from [9].

Figure 2.2 shows how a UA consists of a UAC and a UAS. When creating a request, the UAC creates a client transaction, and sends it to a UAS, which creates a server transaction to handle the request. An important concept related to user agents and message exchange is a dialog. This is a peer-to-peer relationship between two UAs that persists for some time. It is useful for properly

handling the sending and receiving of messages between two entities that intend to communicate. Of the request messages listed in Table 2.1, only the INVITE and SUBSCRIBE messages create a dialog. Other messages are exchanged within a dialog.

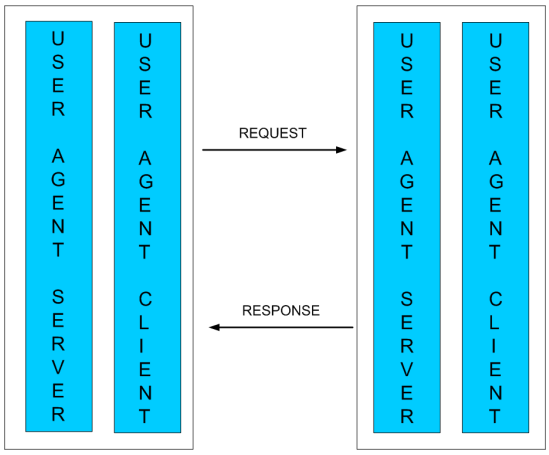

Figure 2.2: SIP user agent design showing how messages are sent, received and responded to.

2.1.2 Servers

SIP defines three main types of servers, namely the proxy, registrar and the redirect servers, whose functions are given in Table 2.3. Some of the terms given in these definitions are explained in the next three sections. SIP servers, like UAs, are also TUs that create server transactions to handle client requests. For example, a registrar can be defined as any UAS that creates a server transaction for registration requests it receives from a client transaction. UAs, proxies, registrars and redirect servers all have cores that differentiate one from the other, but those cores are TUs that can respond to SIP messages in similar ways.

Since SIP servers, like all SIP entities, are defined as logical elements, servers can be co-located on the same hardware. This means that from an implementation point of view, a registrar can be co-located with a proxy, or a redirect server with a registrar. In fact, a SIP server can be defined as such on a purely transaction by transaction basis [9].

Server name	Function
Registrar	A server that accepts REGISTER requests and places the information it receives in those requests into the location service for the domain it handles
Proxy	An intermediary entity that acts as both a client and a server for the purpose of making requests on behalf of other clients
Redirect	A user agent server that generates 3xx responses to requests it receives, directing the client to contact an alternate set of URIs

Table 2.3: SIP servers and their functions.

2.2 Registration

Human users of SIP networks use some kind of SIP UA device such as a software phone (called a softphone), hardware phone (called a hardphone) or a videophone, to communicate. Such a user, say named Mosiuoa, possesses a unique SIP address such as sip:mosiuoa@ru.ac.za, which is known as a SIP Address of Record (AOR). Mosiuoa uses this address to identify himself to the rest of the world. This AOR is structured in a similar way to an email address, where the mosiuoa portion of the AOR identifies the user's name, and the ru.ac.za portion identifies the domain in which he belongs.

In order to participate in the domain and begin to interact with others, Mosiuoa's UA needs to formally join the ru.ac.za domain. In SIP, this process is known as registration, and is formally described as the process through which a user associates their AOR with another form of address known as a contact address or contact URI, which more accurately informs the network of where this user can be reached. For example, the AOR sip:mosiuoa@ru.ac.za may be associated with the contact address sip:mosiuoa@146.231.123.55:5060, which indicates the IP address and listening port of Mosiuoa's SIP UA. This process of registration is accomplished by the creation of a SIP REGISTER message (see Table 2.1), which the UAC sends to a UAS which can handle the request. A user may associate a single AOR with multiple contact addresses so that messages can be sent to various SIP devices that the user owns. The process of registration is depicted in Figure 2.3.

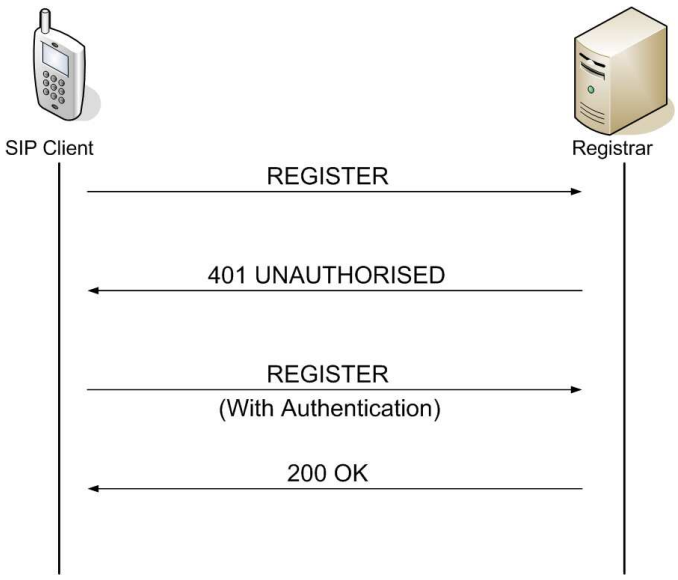

Figure 2.3: The SIP registration process.

In order for a user to join a SIP domain, the UAC must know beforehand the destination to which to send the REGISTER message. There are several methods available to the UAC to do this, which are discussed in [9]. These are:

Manual Configuration A UA can be manually configured by an administrator or experienced user, or may have built-in knowledge of the location to send requests.

Default Domain If a UA is not manually configured, it may choose to send the request to the host part of its AOR, such as sip:ru.ac.za for a UA on the domain ru.ac.za.

Multicast Registration A UA can send the request to the multicast address sip.mcast.net, which is known as the *all SIP servers* address (224.0.1.75 for IPv4), which will be responded to appropriately by a UAS listening on this multicast address.

When the registrar receives the registration request, it performs certain administrative checks on it. For example, the registrar must check that the host portion of the AOR of the originating UA falls in its administrative domain. A registrar may also challenge unauthenticated requests as shown in Figure 2.3. If all requirements have been met, the registrar adds the binding between the AOR and the contact address into an abstract data service called the location service. The location

service stores all the bindings on the behalf of the network and exposes these to other servers such as the redirect server. It can be implemented in various ways since the protocol specification does not standardise its structure or the exact format of the resource records it contains. Implementers of location services have traditionally used technologies such as relational databases and the Lightweight Directory Access Protocol (LDAP) [17]. Figure 2.4 shows how the location service is populated and how it assists servers such as proxies to perform their functions. The behaviour of the proxy is described in detail in the next section.

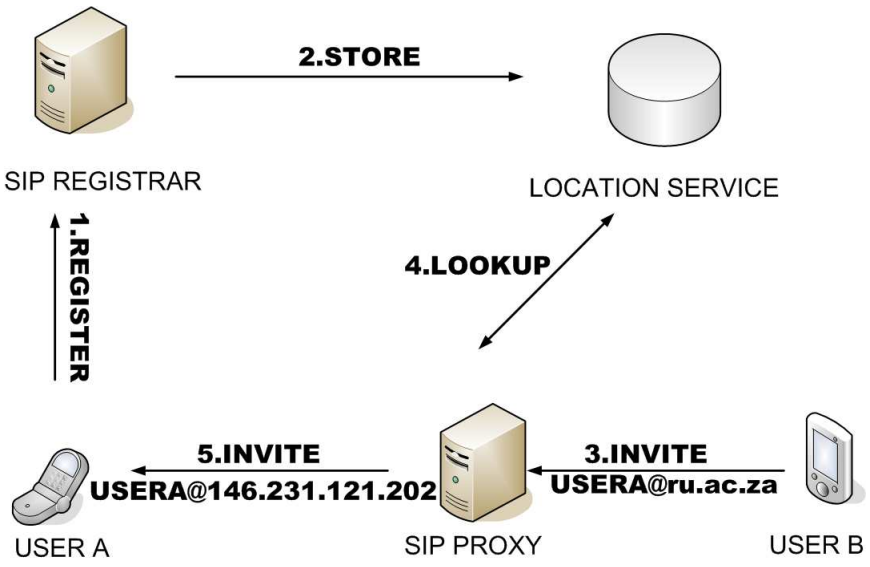

Figure 2.4: How the location service is used in establishing sessions.

The use of the REGISTER message is not limited to facilitating the process of adding bindings to the location service, but can also be used to modify or remove them. Each binding in the location service has an associated expiry interval, after which the binding will be automatically removed. This interval is stipulated in the class 2XX OK response sent back to the client after a successful registration. It is the responsibility of the client to refresh bindings before the record expires, which it accomplishes by sending an appropriate REGISTER message back to the registrar. A client may also remove bindings by modifying the Expires header in the REGISTER message by setting it to 0, which results in the removal of the binding from the location service.

2.3 Proxy service

Once a user has successfully registered with the registrar, that user is free to establish sessions with other users in the network. These sessions are established through the exchange of messages between endpoints, which in telephony is known as signalling. To aid in the routing of messages between endpoints, SIP defines the role of an entity called a SIP proxy. SIP distinguishes between a *stateful* and a *stateless* proxy. According to [9], a stateful proxy maintains the client or server transaction states of a request, and as such is alternatively known as a transaction stateful proxy. A stateless proxy does not maintain such state, and as such, simply forwards every request it receives downstream, and every response it receives upstream. Apart from message relay, SIP proxies can also perform other functions such as:

Authentication The proxy can challenge user requests to make sure they are authorised to use the service.

Loop Detection Messages could sometimes loop between a set of proxies forever, and a proxy can perform actions to prevent this from occuring.

Forking A proxy can route messages to several destinations at which a user has indicated availability.

Record Routing A proxy can request to remain in the signalling path in subsequent messages in a dialog.

Administrative As flexible elements in a SIP network, policies based on traffic load, security and user preferences can be taken into account when performing message routing.

An element intending to proxy a request it has received from a UAC must perform four key tasks. Firstly, it validates the request, which includes checking for correct message syntax. Secondly, it inspects the destination of the message, which is contained in a SIP header called the Request URI. Thirdly, it determines the target, or targets of the message. Lastly, it forwards the message to the target SIP entity. Needless to say, since a stateful proxy handles requests on a transaction basis, it can handle functions such as retransmissions when performing these steps.

This discussion on the role of the proxy completes the description of Figure 2.4 by showing how, after the process of registration has succeeded, messages from other users can be proxied to the user's location for the establishment of sessions. Session establishment is discussed in greater detail in the next section.

2.4 Establishment of multimedia sessions using SIP

SIP is able to support the establishment of a wide range of multimedia sessions through signalling mechanisms. Figure 2.5 shows the sequence of messages that are exchanged in order to setup and terminate a multimedia session between two UAs in separate domains. The session is established through the use of the INVITE message, which was defined in Table 2.1.

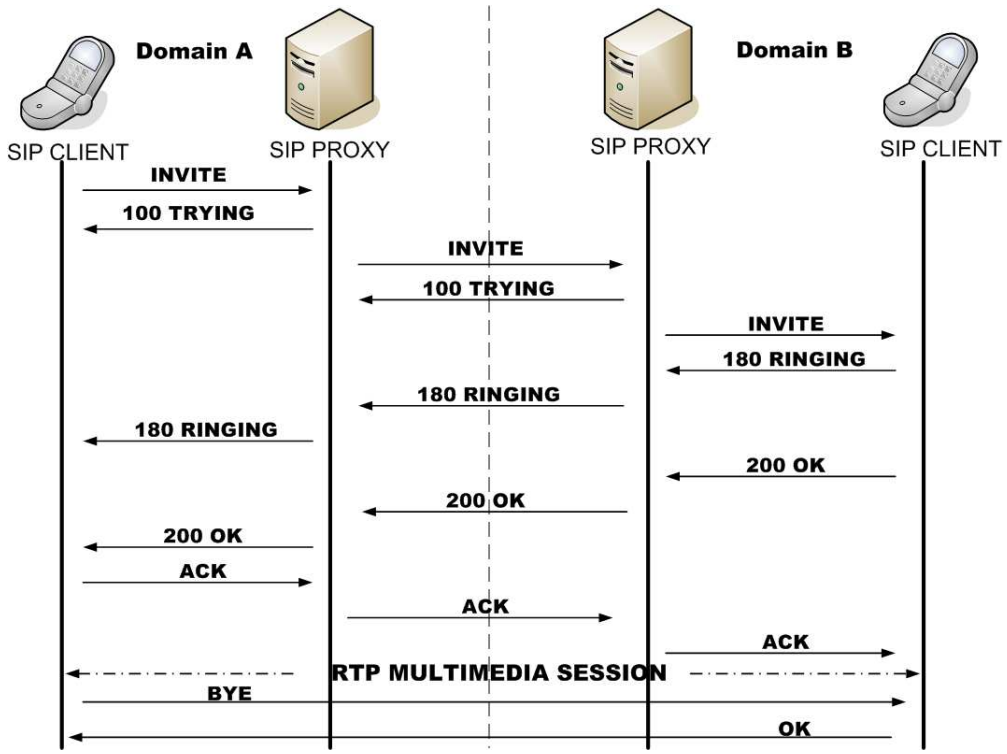

Figure 2.5: Session establishment across two SIP domains.

The UA generating the INVITE can use the methods described in section 2.2 to locate the SIP proxy in its own domain. When both the sender UA and the receiver UA are in the same domain, the records in the location service can inform the proxy of domain A in Figure 2.5 where next to send the message, called the next hop. In the non-trivial case where the sender and receiver are not in the same domain, the proxy on the originating side must determine the next hop address, port and transport protocol which it must use to route the message. Local policies may be in place that determine the correct behaviour in this instance, such as reading values from a database or sending the message to a default server, as suggested in [9]. However, the generally used method

is that the proxy will use DNS procedures to retrieve the appropriate SRV and NAPTR records of the SIP proxy serving the foreign domain, as indicated by the domain part of the Request URI in the SIP request [18]. Using this technique, as shown in Figure 2.5, the INVITE message can be routed to the proxy in domain B. There, the location service informs the proxy of the location, or locations of the intended user. The proxy then uses the contact address of the proxy in domain A (which it has appended in a SIP header called a Via header in the original INVITE) to re-route subsequent messages back to domain A.

2.4.1 Audio and video sessions

Figure 2.5 shows that once the three-way handshake of INVITE-OK-ACK has completed, multimedia traffic between those endpoints proceeds in a peer-to-peer fashion. Central to the provisioning of these services is the use of the Session Description Protocol (SDP) [19] which is used to describe sessions between participants including media details, transport protocols and other forms of metadata. SDP can be used, for example, as as basis for the negotiation of media codecs between users in an audio or video session.

In Figure 2.5, at the point where the user agent in domain B sends a 200 OK response to the SIP request, a dialog is created (see section 2.1.1). Using Figure 2.5 above as an example, after the session has been established and a dialog has been created, one of the SIP UAs may choose to modify the properties of the session in some way, such as using a different audio or video codec. A new INVITE message called a re-invite is generated and sent, which does not need to traverse the proxies since the signalling can be accomplished through the use of the already established dialog, and the UAs can negotiate the new properties of the session themselves.

2.4.2 Instant messaging

The introduction of the MESSAGE request added the ability to convey text information in near-realtime between users. This form of communication is known as instant messaging, or IM. Figure 2.6 shows how messages would be exchanged between two users in an IM session.

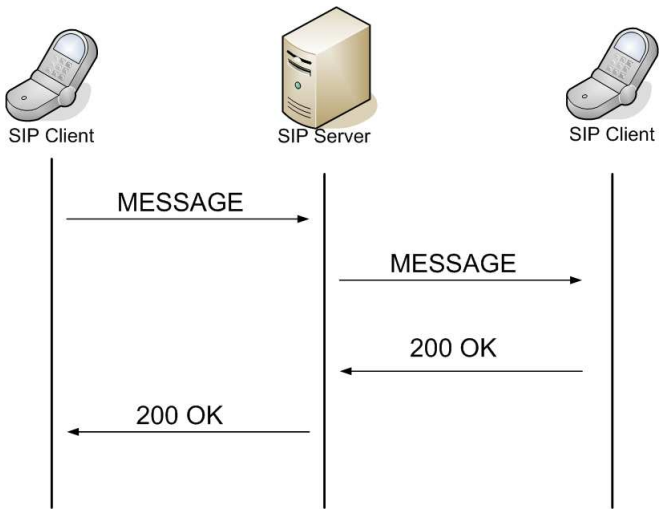

Figure 2.6: A typical IM session between two users with an intermediary proxy.

2.4.3 Presence

IM is a useful tool for communicating, but the initiator of an IM request, unlike an audio or video session, is unable to determine if the user at the other end is able to respond to the IM request or not. Presence refers to the ability of a user to communicate their willingness or availability to interact with others. Presence is supported in SIP with the SUBSCRIBE and NOTIFY request messages [20].

An entity (called a watcher) wishing to obtain a lease for presence updates of another entity (called a notifier) uses the SUBSCRIBE message. The notifier responds with an SIP 200 OK response followed by a NOTIFY, which contains the status information in the body of the message. A notifier will resend NOTIFY messages every time its presence status changes, as well as sending updates at regular intervals. Watchers are responsible for refreshing leases on presence updates, which is also facilitated by the SUBSCRIBE message. An example of a successful presence subscription and notification is given in Figure 2.7.

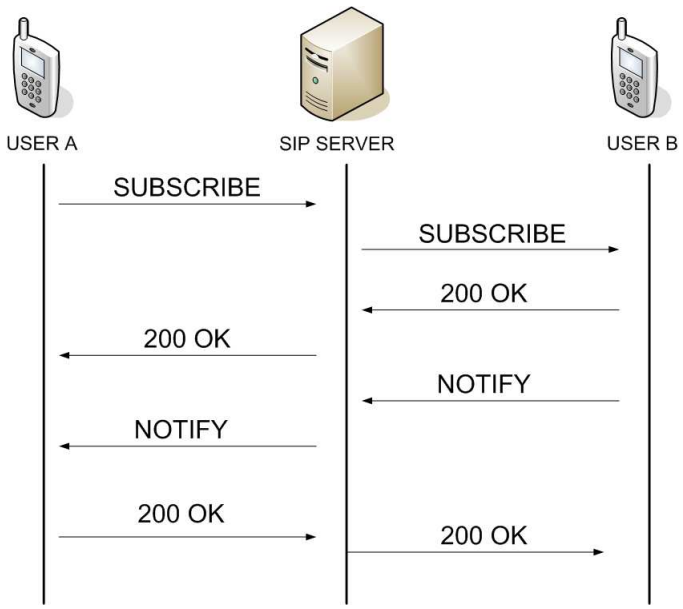

Figure 2.7: A typical presence subscription between two users with an intermediary proxy.

Subsequent to [20] which defines the SUBSCRIBE and NOTIFY messages, a further extension to SIP was defined with the introduction of the PUBLISH method [21]. Use of this method is meant for delivery to a special UAS which is responsible for collecting state and distributing it to subscribers. In this manner, a notifier can publish state to an event compositor and valid watchers can obtain the notifications from the compositor.

2.5 Summary

SIP is an application layer protocol which can be used to create multimedia sessions such as audio, video and instant messaging, as well as support presence subscription and advertisement. SIP defines a number of abstract entities that provide services to the SIP network. However, it is evident from the protocol specification that there is no rigid binding between a service and the identity of the entity providing that service, since almost all SIP entities but one, have the ability to both initiate and respond to messages that create these sessions as TUs. Many facets of realtime communication using SIP are essentially peer-to-peer, and support the transfer of information without intermediate and centralised elements. This includes the peer-to-peer RTP sessions that are established subsequent to session establishment, and the dialogs that are created

between user agents. Decentralising SIP means decentralising the roles of the proxy, registrar and redirect server, and the implementation of a decentralised location service. The resulting design is known as P2P SIP. The next chapter looks at some of the peer-to-peer protocols that may assist in performing this decentralisation.

Chapter 3

Peer-to-Peer Protocols

> *All animals are equal.*
>
> - Animal Farm, George Orwell, 1945

Peer-to-peer networking is a concept that has risen in popularity in recent times, as evidenced by the large collection of computer applications that are based on peer-to-peer techniques and support services such as file sharing, instant messaging and peercasting. The power behind peer-to-peer networking lies in its ability to pool the bandwidth, processing power and storage capacities of a number of participating hosts that are spread over a potentially large geographic area. This can be achieved in a self-sustaining and self-organising manner, without any or much centralised control or intervention.

The purpose of this chapter is to give a brief overview of peer-to-peer systems as an initial step towards selecting an appropriate platform for decentralising SIP. A classification of peer-to-peer systems is given, after which two classes of peer-to-peer systems are compared, highlighting the advantages and disadvantages of each for the purposes of P2P SIP.

3.1 Classifying peer-to-peer systems

There are numerous definitions for the term peer-to-peer systems. A detailed definition is given in [22]:

> *peer-to-peer systems are distributed systems consisting of interconnected nodes able to self-organize into network topologies with the purpose of sharing resources such as content, CPU cycles, storage and bandwidth, capable of adapting to failures and accommodating transient populations of nodes while maintaining acceptable connectivity and performance, without requiring the intermediation or support of a global centralized server or authority*.

In the definition above, network topologies refer to the arrangement of nodes in a network and the connections between them. Four popular network topologies are shown in Figure 3.1, followed by a brief description of each.

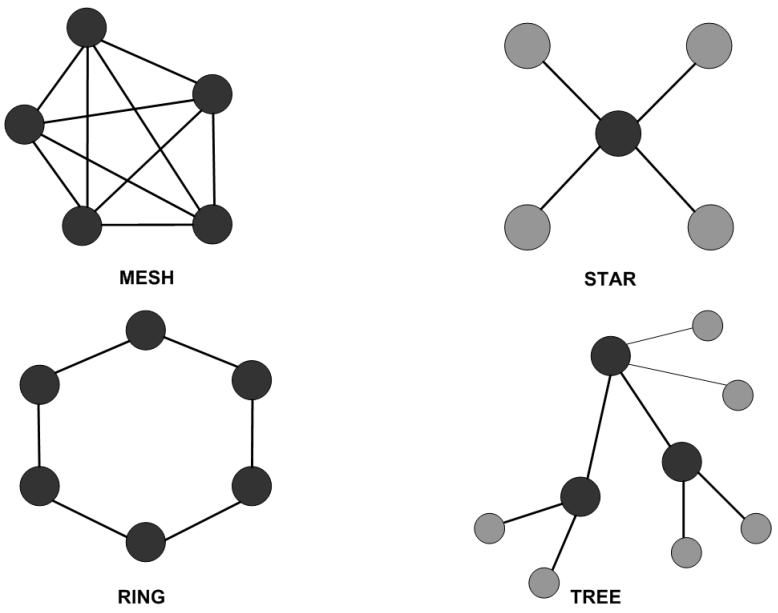

Figure 3.1: A classification of peer-to-peer systems.

Mesh Where all nodes in the network are connected to all other nodes in the network.

Star Where all nodes in the network are connected to one central node such as a network hub.

Ring Where the nodes in the network are arranged in a circular fashion.

Tree Where sets of nodes are connected to one central node in a hierarchical pattern, and in turn, the central nodes are connected to each other.

It is often possible to layer networks over others. Such networks are known as *overlay networks* or simply *overlays*, of which the Internet is a prime example as it is layered over other telecommunication networks. The next section discusses the different types of overlays that exist and some of the protocols that are used to create them.

3.2 Overlay networks

There are many different possible ways to classify overlays, which is due to the sheer number of protocols in existence today and the rate at which new ones are being invented. However, a popular classification of overlays distinguishes between *structured* and *unstructured* overlays.

3.2.1 Structured overlays

Structured overlays are efficient at mapping a key to a node [23]. Usually, cryptographic functions are used to generate node identifiers (or *node IDs*) for nodes in the overlay such that every node is identified by a unique identifier. Nodes in a structured overlay can host resources, whereby each resource is identified by a resource identifier (or *resource ID*), which is in the same address space as the node IDs. In many contexts, the resource ID is also known as a key. It is this relationship between the two types of identifiers that allows structured overlays to make an association between a resource and the node that hosts that resource. The node with the identifier whose value is nearest to the value of the resource identifier is selected to host the resource. The hosting of resources is defined in an abstract way such that the incumbent node may physically store the resource (such as a file) or simply have a pointer to where that resource can be found (such as a URI).

Node IDs are used to establish logical relationships between nodes with identifiers that are near each other by some protocol-specific definition of nearness. Nodes in structured overlays keep track of a limited number of neighbours through the use of a routing table, which is typically of size $log(N)$, where N is the total number of nodes in the network. These routing tables are consulted when the overlay handles a query for an identifier. A query is passed from node to

node aided by these routing tables. At each hop, the query is routed nearer to the target until the node to which the query maps is obtained. Due to their ability to efficiently map keys to nodes, these protocols are often called Distributed Hash Tables (DHTs). There are many DHTs in existence and three popular ones are described in the next three sections.

3.2.1.1 CHORD

Chord is arguably the best known DHT and was developed at the Massachusetts Institute of Technology (MIT) in the United States [4]. At its core, Chord is a simple lookup substrate which maps a key to a node ID. A Chord overlay has a ring topology where nodes are assigned successive node IDs which are generated by using a variant of the consistent hashing function [24]. It is sufficient for a Chord node to maintain the $< nodeID, IPaddress >$ tuple of its immediate successor in the ring to ensure the correctness of the lookup algorithm. However in addition to this, to improve the performance of the algorithms, a Chord node can keep record of (at most) m neighbour nodes in a routing table called a ***finger table***, where m is the number of bits in a node ID.

A node n with an m-bit identifier builds up a finger table of successor nodes using the formula $s = successor(n + 2^{\text{i-1}})$, where $1 \leq i \leq m$ and all arithmetic is done modulo 2^{m}. The i^{th} finger table entry at node n contains the identity of the first node s that succeeds n by at least $2^{\text{i-1}}$ on the identifier circle. A Chord finger table entry contains a mapping between a node ID and the corresponding IP address and port number of the successor node. The $lookup(key)$ algorithm proceeds either iteratively or recursively, where the query is propagated through the overlay, at each step being routed to the node whose identifier has a numerical value equal to or closest to key, until that node is found. This information allows successful lookups to be made in $O(log(N))$ time in a Chord overlay of N nodes.

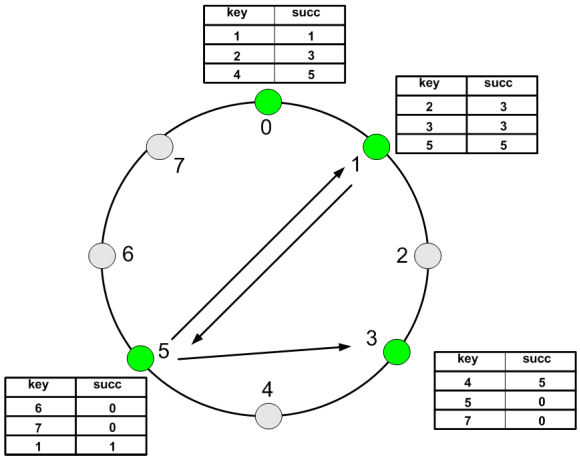

Figure 3.2: A typical Chord ring.

Figure 3.2 depicts a Chord ring where only the nodes with identifiers 0, 1, 3 and 5 are online. Nodes in the ring have three-bit identifiers (i.e. $m = 3$) and as such, there can only be up to eight nodes in the ring, each with up to three entries in its finger table. The IP addresses and port numbers have been omitted in the finger tables for simplicity. As an example, the node with identifier 1 stores the successors of nodes 2, 3 and 5. In the figure, a lookup is performed by the node with identifier 5, which searches for a resource which happens to be stored at the node with identifier 3.

Chord was complemented by an implementation of the protocol which was written in C++ and was coupled with a module called DHASH [25] to provide an efficient data management system. DHASH exploits the powerful lookup protocol it is layered over to expose a simple $< get, put >$ interface to the DHT for storage and retrieval of data units. Data is stored in blocks of 14 units at the neighbour nodes to provide high availability of data. A node requesting a file needs only to retrieve 7 of those data blocks to recreate the original file. The combination of Chord and DHASH has formed the basis of the development of distributed and cooperative file systems [26, 27].

3.2.1.2 Pastry

Pastry, like Chord, is a lookup and routing substrate for peer-to-peer systems [5]. An overlay based on Pastry is also arranged in a ring topology and executes lookups in $O(log(N))$ time,

where N is the number of nodes in the overlay. The two main differences between Pastry and Chord are in the structure of the routing tables, and the modified way in which Pastry defines the closest neighbour.

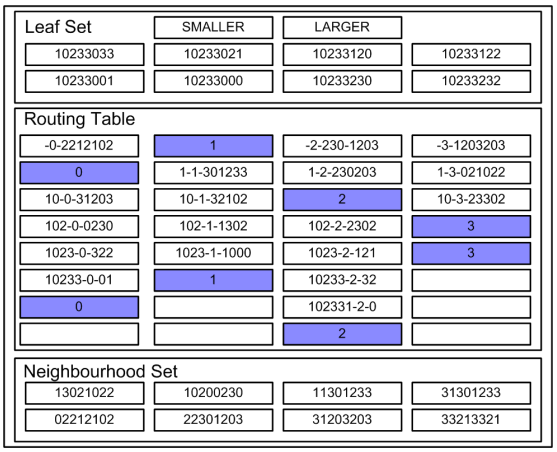

Figure 3.3: An example of a Pastry node's routing table. Adapted from [5].

Figure 3.3 shows an example of a hypothetical Pastry node. This node has node ID 10233102. The shaded cell in each row of the routing table shows the corresponding digit of the node's ID. The node IDs in each entry have been split to show the common prefix with 10233102 - next digit - rest of node ID. The associated IP addresses have been omitted for simplicity.

The routing table in Pastry is more complex than in Chord. In Pastry, a node's routing table contains $\lceil log_{2^b} N \rceil$ rows with $2^b - 1$ entries in each row, where b is a configuration value, usually 4. The entries in row n refer to nodes whose identifiers share the local node's ID in the first n digits, but whose $n + 1$th digit has one of the $2^b - 1$ possible values other than the $n + 1$th digit in the local node's ID [5].

In addition to routing tables, Pastry defines *neighbour sets* and *leaf sets*. A neighbour set M contains the location bindings of a set of $|M|$ nodes that are closest to the local node by an application defined proximity metric. The neighbour list is not explicitly used for routing, but rather ensures that the nodes chosen in the routing table are close to the local node according to the proximity metric. The leaf set L contains a set of $|L|$ nodes numerically closest to the local node in the number of common prefix digits of the identifier. Half of these have identifiers smaller than the local node, and the other half larger. These extra features are optimizations that improve routing performance.

Chord uses consistent hashing to generate a node ID based on the IP address of the node. This means that a Chord node may have neighbours with node IDs that are numerically close to it, but are far removed from it in a geographic sense, or in the number of IP routing hops necessary to reach them. This maybe undesirable, since it has the potential to increase the latency of message exchanges between neighbour nodes if neighbours are geographically dispersed as a result of the proximity metric. Pastry is able to embed within the routing table, knowledge of the distance of a remote node from the local node, so the local node can optimise its routing tables appropriately.

When routing a message, a Pastry node checks if the key falls in the range of nodes in its leaf set. If so, it routes the message directly to that node. If this fails, it will look for an appropriate node in its routing table and routes the message to the node in the overlay with the most number of common prefix digits to the key that it is aware of.

There are two known, open source implementations of the Pastry protocol, which are FreePastry [28] and SimPastry/VisPastry [29]. FreePastry was developed at Rice University in the United States and is written in Java. At the time of writing, it is at version 2.1. Since its release, there are a number of distributed applications that have been built using FreePastry including SCRIBE [30] (a large-scale, decentralised multicast infrastructure), PAST [31] (a peer-to-peer archival storage utility) and SQUIRREL [32] (a peer-to-peer web cache). SimPastry and VisPastry were both developed by Microsoft research. SimPastry is a simulator for the Pastry protocol and was developed using C# and the Microsoft Common Language Runtime (CLR). VisPastry is a visualisation tool for the Pastry protocol.

Another very popular DHT is Bamboo [33] which is loosely based on Pastry. It was developed at the University of California, Berkeley in the United States. It is particularly interesting as it was chosen as the routing substrate of choice at OpenDHT [34], which is a globally available DHT service for peer-to-peer researchers. A detailed description of this DHT implementation is provided in section 6.3.2.

3.2.1.3 Content Addressable Network

The specification of the Content Addressable Network (CAN) was the first among the new generation of structured protocols, such as Chord and Pastry, to use the term distributed hash table [6]. Rather than having a circular keyspace, CAN features a d-dimensional Cartesian coordinate keyspace on a d-torus. The coordinate space is partitioned to nodes, with each node being responsible for its own zone. As nodes join and leave the overlay, the coordinate zones can change dramatically. An example of a CAN is given in Figure 3.4.

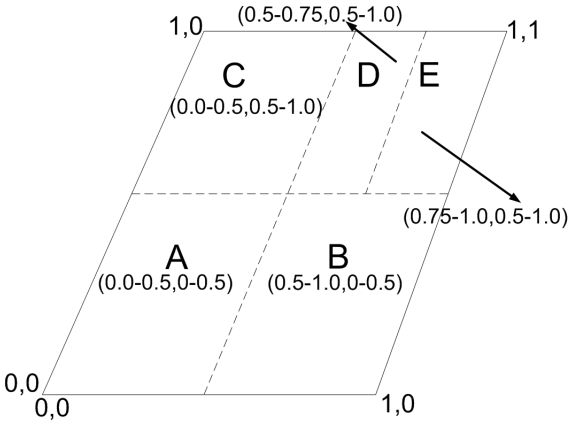

Figure 3.4: Example of 2-d [0,1] x [0,1] coordinate space with 5 nodes. Source: [6].

In order to join, a new node p must discover another node which is currently in the CAN. The Domain Name System (DNS) is used to discover the IP addresses of a number of currently online CAN nodes. The joining node must choose, at random, some point called P in the coordinate space and communicate this point to the CAN. The CAN will assist in locating the node responsible for the zone in which P lies (say x) through routing procedures, and thus the `JOIN` request reaches node x. This node will then admit node p into the CAN, supply p with a list of its own neighbours including x itself, splits its zone in half and allocate half the zone to the new node. Any nodes directly affected by the admission of p into the CAN, that is those in adjacent zones, are notified so they can also update their neighbour lists. Periodic refreshes are also sent between nodes in these adjacent zones.

In order to successfully store the record $< K_1, V_1 >$, the key K_1 is hashed using a hashing function and mapped to a specific point called P in the coordinate keyspace. The record will be stored at the node which is responsible for the zone in which P belongs. The same hashing function can be used by a subsequent node which is searching for the record V_1 identified by K_1, by hashing the key which will map to the owner of key K_1.

The investigation into structured protocols and their implementations found that interest in the CAN protocol is significantly less than in other protocols such as Chord and Pastry. However, since January 2005, CAN was incorporated into the Meteor component of the JXTA suite of protocols [35]. Meteor provides a simplified CAN implementation for creating overlay networks consisting of transient peers. More recently, a passive version of CAN called pCAN was used in the SIPDHT [36] application described in section 4.3. It is called passive since clients do not

participate actively in the overlay until they are invited to do so by a node that is already in the overlay [37].

3.2.2 Common APIs for structured overlays

The previous section gave examples of some popular structured protocols. These protocols share the common trait of being able to map a key to a node, though even a glancing examination uncovers discrepancies in the way each protocol achieves this. The Infrastructure for Resilient Internet Systems (IRIS) [38] is a project that is driven by academics from several institutions in the United States with the purpose of developing an infrastructure based on structured protocols, that will give birth to a host of large scale distributed applications. One of the areas of research at IRIS focuses on the similarities between structured protocols that allows for the definition of an API that can be used to access a common set of services that the protocols provide. The approach is summarised in Figure 3.5.

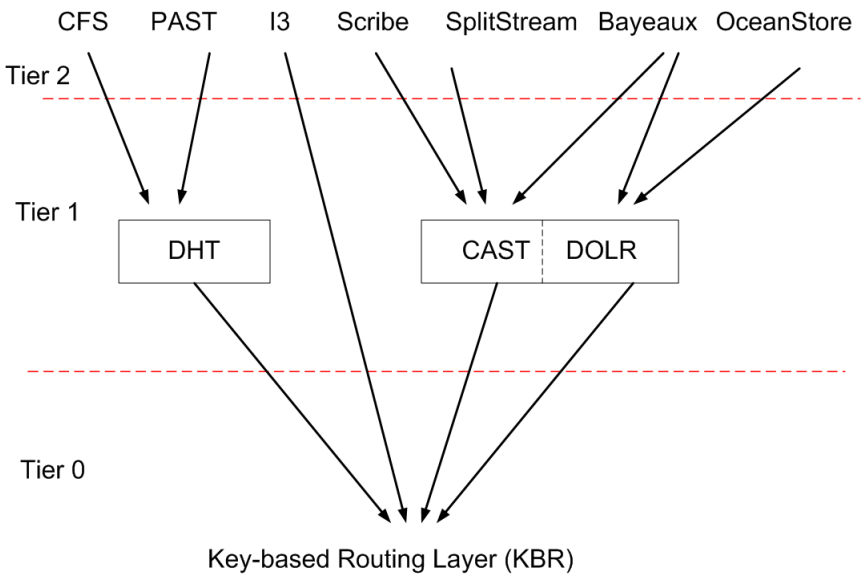

Figure 3.5: Basic abstractions and APIs for structured protocols. Source: [39].

The researchers at IRIS have defined an API called Key Based Routing (KBR), which represents all the capabilities common to structured protocols, which is located at tier 0 in the model. KBR becomes the common denominator for other services located at tier 1 such as structured protocols

themselves (DHTs) and anycast/multicast (CAST) and decentralised location and routing protocols (DOLR). Lastly, most of the advanced services at tier 2 rely on the abstractions provided at the tier 1 level. The major contributions of this work are that it encourages the development of applications based on structured protocols by third parties, and it shows how a common API can be used to interact with different kinds of DHTs. The overlay construction toolkit called Overlay Weaver [40] which was designed primarily for application developers to test their applications on different kinds of DHTs, is based on this design.

3.2.3 Unstructured overlays

Unlike structured overlays, nodes in unstructured overlays are organised in a random graph where no constraints are enforced. The protocols used to locate resources in the overlay are less deterministic than in structured overlays, where broadcast, flooding and random walks are usually employed [22]. There is greater variance in the characteristics of unstructured protocols than in structured ones. Three popular types are examined in the next three sections.

3.2.3.1 Gnutella

Gnutella [41] is a file sharing protocol which has evolved considerably throughout the years of its existence and has had many modifications to improve efficiency and lower bandwidth consumption [42, 43]. Initial bootstrap nodes on a Gnutella network are discovered via out-of-band mechanisms such as a manual search on the Internet, where recently live nodes are listed on public caches known as GWebCaches [44]. Once an existing node has assisted the new node to join the network, the new node broadcasts `PING` messages to the network to announce its presence. `PONG` messages are re-sent to the node, so it can gain knowledge of other nodes in its vicinity.

A resource such as a file is published on the network with a `PUT` method. Searches for the resources stored by nodes in the network are facilitated by `QUERY` requests, which are broadcasted throughout the network. The broadcast queries are limited to a certain number of hops before the query fails. A successful query match called a `QUERY RESPONSE` proceeds along the same path as the query through back propagation and contains details as to where the file can be found. A `GET` request directed to the target node is used to download the desired resource.

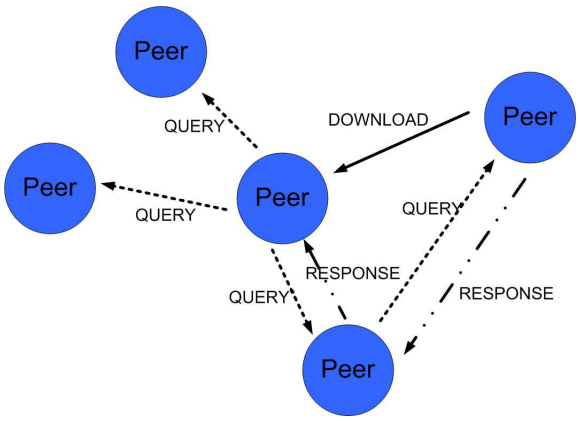

Figure 3.6: Document location and retrieval in Gnutella. Adapted from [45].

3.2.3.2 KaZaA

KaZaA is a peer-to-peer protocol which is similar to Gnutella in that it does not use central servers [2]. However, unlike Gnutella, the KaZaA protocol defines two different types of nodes in the network: ordinary nodes and super nodes. This means that KaZaA is a distributed but hierarchical peer-to-peer system, exhibiting a tree topology (see Figure 3.1). The role of the central server is emulated across a set of super nodes, each with an associated set of ordinary nodes that communicate with it. Each super node keeps record of the locations of its ordinary client nodes, as well as those of its tier 1 peer neighbours.

When a client node wishes to share a file, it registers the details of the file (metadata) with its related super node. The file metadata contains keywords that will be matched against a search string during a query request. When a node wishes to download an indexed file, it forwards the request through its super node. The super node will in turn proxy the request to the rest of the network using broadcast mechanisms. A list of successful matches are sent back to the super node, which returns the results to the requesting node. That node can then use the location details contained in the set of results obtained on its behalf to download the file from the network. Figure 3.7 shows ordinary nodes with their respective super nodes, and how a file is published and downloaded by peers.

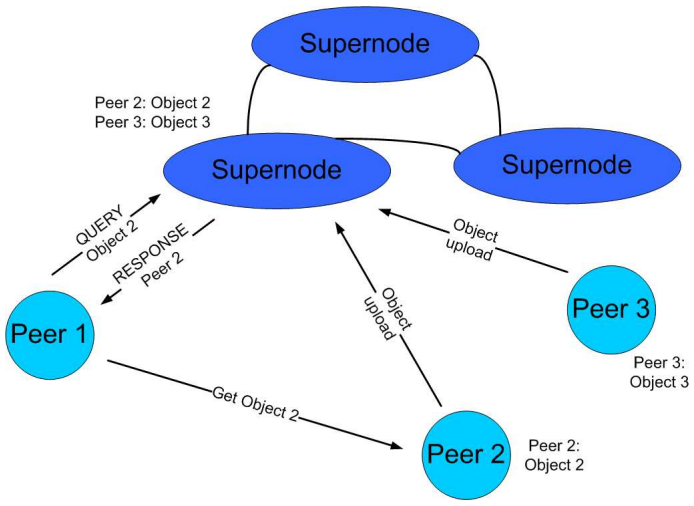

Figure 3.7: Super nodes and ordinary nodes in a KaZaA network. Source: [45].

3.2.3.3 BitTorrent

BitTorrent is a file sharing system that has both centralised and peer-to-peer components [3]. Its hybrid design is similar to the star topology in Figure 3.1, where a central server is used to manage user downloads. A file that is to be shared is broken into data blocks and is stored on the network. A server called a *tracker* keeps record of a file that is shared and of the users that either have a complete copy or only blocks of the file [46, 47].

When a client wishes to share a file to the rest of the network, it divides the file into blocks of up to 256KB in size which can be distributed to other peers. It also creates a file known as a *torrent* which contains metadata that describes the file. The torrent is then made available to the public and registered with a tracker. Peers that have blocks of the file are called *seeders*, and the initial peer that introduces the file to the network is called the *initial seeder*. A client that wants to download the shared file, must first obtain the associated torrent for the file. The torrent directs the client as to the location of the tracker, which the client proceeds to contact, requesting the desired file. The tracker returns a list of seeders for this client to contact. The client can then start downloading blocks of the file from the indicated peers. BitTorrent is sophisticated in that it allows for nodes to *choke*, meaning a temporary disabling of uploading connections so as to maintain a consistent downloading rate, since uploading causes congestion on the node's bandwidth allocation. Also, the network rewards nodes with high upload speeds with high download speeds. BitTorrent is unusual because it employs what are called *tit-for-tat* algorithms to discour-

age freeloaders (that is, people who seek to benefit from the system but do not wish to contribute resources) by providing quicker download times for those that share data for others [45]. Figure 3.8 shows the relationship between a tracker, a torrent and peers in a BitTorrent network.

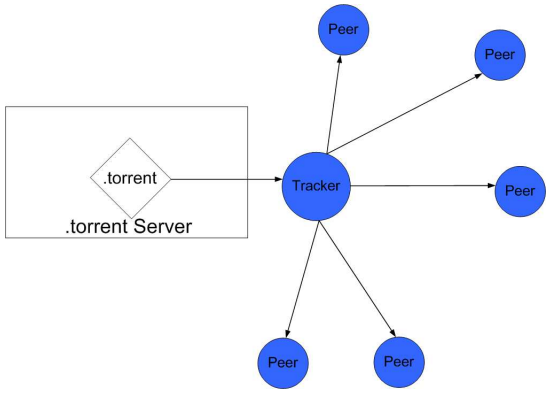

Figure 3.8: BitTorrent architecture. Source: [45].

3.2.4 Comparing structured and unstructured protocols

Having classified and presented examples of the two classes of peer-to-peer protocols, it is now possible to analyse and compare them. A thorough discussion would involve examining them from several perspectives, but only the aspects that make the protocols desirable or undesirable for P2P SIP are of concern here. As such, this section iterates through the SIP services that have been described, and discusses the suitability of structured and unstructured protocols for performing each of them.

3.2.4.1 Registration

Registration is the joining process through which a user associates their AOR with a contact address that indicates to the rest of the network the locations at which the user can be reached. In a peer-to-peer setting, it is important that the system support this process with as little administration or oversight as possible.

In structured overlays, a joining node first obtains the location of a node that is currently in the overlay (called a *bootstrap node*). After this node has bee contacted, it subsequently assists the joining node in becoming part of the overlay, either directly, or by referring the join request to a more suitable node (called the *admitting node*). DHTs do not specify how to locate the bootstrap

node, which means that some other protocol or mechanism must be used to achieve this. In the joining process, the new node learns who its neighbours are and populates its routing tables so it can reach them, and those nodes in turn learn of the existence of the new node and update their routing tables as well. This establishes reachability for passing messages to the new node, but does not yet perform the SIP address mapping. DHTs are lookup and routing systems and do not natively support insertion of data sets. However, in section 3.2.1.1, a module called DHASH was described that implements a distributed storage system based on Chord. This proves that it is possible to exploit the routing algorithms in DHTs by using an application layer to insert data at the node where the hash of the key is closest to the node ID. Building upon this logic, it would be possible to store a copy of the contact address of the joining node in some abstract form, at the node whose ID is closest to the hash of the AOR.

Unstructured overlays are also challenged by this first node problem, but they usually do provide discovery mechanisms. A technique that is commonly used is that of a central host cache such as the GWebcache in Gnutella networks which caches a list of nodes that are known to have been online in the recent past. This suffers from the problem of having stale entries in the host cache that causes either delays or the inability of nodes to join. Unstructured overlays often leverage these types of systems by keeping a local cache of reliable nodes which have been discovered which can be used in subsequent attempts to join the system, such as the UDP Host Caches [48] in Gnutella networks. Unstructured overlays are very well suited for distributed resource storage, which is what makes them popular in file sharing applications.

3.2.4.2 Location service

The location service is an abstract data service that stores the location bindings of UAs and exposes them to entities that need to access them. For structured protocols, an application layer can be built over the DHT and be used to facilitate the insertion of a registration record which can be appropriately stored in the overlay. Over a period of time, as more nodes begin to join and register their addresses, a larger location service is built up. Hashing functions such as consistent hashing also ensure that the load is distributed equally among all the nodes that are participating in the overlay. However, in addition to simply populating the location service with registrations, nodes also need to retrieve those bindings which are necessary for session establishment (see Figure 2.4). The application layer could be extended to provide a retrieve function that returns the values stored at the owner of the key. Additionally, a location service must also be able to update and remove those bindings, such as when users become available elsewhere or when they

leave the network. Further extension of the application layer could be made to provide update and remove functions to perform these duties.

Unstructured overlays are primarily used for content distribution, and therefore in theory, can also be used to implement a location service for SIP. Nodes would be able to use unstructured protocols to store bindings, in file format for example, across the overlay. However, the storage of data items in these types of overlays is independent of any structured rules, therefore there is no deterministic way of locating stored data. Typically, flooding methods are used to perform the search for data items in the network, which means that query requests consume much bandwidth. Such a solution does not scale well with system size [45]. Unstructured overlays are more suited for content distribution applications such as file sharing, in that they can support complex queries to match patterns in a resource name, such as a search for a data file based on a substring of its actual name [49]. SIP location queries are not usually in the form of a complex query, but in the exact SIP AOR of the user who is to be contacted such as when a user selects an entry in a contact list.

3.2.4.3 Proxy

The function of a proxy is to behave as both a server and a client in order to make requests on the behalf of other clients. To support the different types of multimedia services that were described in section 2.4, essentially the role of the proxy is to route messages between users and where necessary, consult the location service to determine the location of the target user. In a DHT environment, it is possible to decentralise the role of the proxy by distributing it over all the nodes in the overlay such that any node is able to receive a message in the form of a key, and use its routing tables to route the request closer to the owner of that key.

Unstructured overlays that do not use central nodes such as BitTorrent trackers are more appealing since they avoid the problem of a single point of failure. It was for this reason that a recent BitTorrent client named Azureus decentralises the traditional BitTorrent tracker, by embedding DHT logic in the client nodes [50]. Where a central node is not used, flooding techniques are used to flood the neighbour nodes that proxy the request further through the network. Some unstructured overlays such as KaZaA use super nodes which could be used as a better proxy solution whereby the super overlay of super nodes decentralise the SIP proxy server among a limited set of overlay nodes. This would ensure that only a small number of nodes are flooded.

3.3 Summary

There are diverse types of peer-to-peers systems which differ in both their topologies. The protocols that create and maintain these different topologies can be described as either structured or unstructured, depending on whether or not they place constraints on the node graph. It is conceivable that either class of protocols can be used to provide a decentralised overlay that distributes the SIP services among the participant nodes. However, the brief examination of the protocols provided in this chapter does show that a DHT that is coupled with an application layer may provide a simpler, cleaner solution to the problem. In addition, due to the similarities that exist between structured protocols, it is possible to derive a common interface to them that will allow nodes to use different DHTs. This would be beneficial for comparing different DHTs for SIP without having to rework the interfaces to each. The next chapter summarises the standardisation effort in the IETF for developing protocols for P2P SIP, and it describes how DHTs have become the ideal peer-to-peer platform for SIP.

Chapter 4

Standardisation Efforts

> *Without counsel purposes are disappointed: but in the multitude of counsellors they are established.*
>
> - Proverbs 15:22, NKJV

A few months prior to the commencement of this project, interest had began to grow among some academics and computing professionals who were interested in investigating the possible benefits of conducting SIP-based communication in a decentralised, cooperative environment. As interest continued to grow, an informal group was formed that communicated mainly through mailing list postings and regular IETF BOF meetings. After much debate and the development of a reasonable amount of consensus on some key issues, the informal group was able to gain recognition within the IETF as an official working group. The P2P SIP Working Group as it is called, is a collection of people who see opportunities for the use of SIP in new and diverse scenarios, where dedicated SIP servers are either unnecessary or simply unavailable. The group has brought forth many proposals for protocols that could be used to achieve this goal, through various Internet Drafts and prototype implementations, in order to communicate and gather support for their ideas. This chapter serves as a guide to some of the current approaches that are being proposed and uses the fruits of the standardisation effort in the IETF to define popular terms and design choices. It also highlights the major consensus points that are likely to feature in the final protocol specification.

4.1 Early work

Dialogue around the development of P2P SIP protocols first began to surface in the open in the form of a mailing list that was hosted by Columbia University in United States in late 2005 [51]. As part of the growing research agenda in peer-to-peer Internet telephony at that institution, researchers there released a paper on what was to become one of the earliest candidate P2P SIP protocol descriptions [52]. The system is based on an hierarchical overlay, built using a structured protocol, that consists of super nodes and ordinary nodes. The motivation behind the decision to distinguish between ordinary nodes and super nodes was that it was believed that some nodes would not have sufficient resources to participate in, and maintain state for a DHT. Thus only stable, high capacity nodes are elected as super nodes. The architecture of a P2P SIP node that follows this design is depicted in Figure 4.1.

A discovery module assists the node to discover and join the overlay by sending a SIP REGISTER message to one or more super nodes that are already in the overlay. One of several possible methods can be used to achieve this, such as DNS lookup, multicast or using cached entries of previously online nodes. When a super node receives a new REGISTER request, it proxies the request to another suitable super node that can admit the new node into the overlay. It determines the identity of that super node by hashing the SIP AOR indicated in the SIP REGISTER message to obtain a key, and requests the overlay for the ID of the closest node to that key. When this super node has been found, it can either redirect the new node to that super node, or proxy the registration request on the new node's behalf. The super node that acts as the admitting node for the new node, adds this node to its list of clients, and creates a resource registration that maps the SIP AOR to a contact address, and stores this value (the discovery module is also used later for locating nodes that are behind NATs).

A user location module interfaces between the application logic and the DHT by forwarding application requests, such as requests for locating buddies, to the DHT layer. If Bob is a user in the overlay and wants to initiate a session, he generates a SIP message such as an INVITE or MESSAGE, and sends it to his super node. The super node uses methods provided by the DHT API interface to obtain the contact address of the intended target node (see Figure 4.1). Once found, Bob can then send the SIP message directly to the appropriate destination. A super node sends SIP OPTIONS messages periodically to its clients in order to detect node failures. A SIP REGISTER message is sent by Bob to his super node when he gracefully leaves the overlay.

The design described above was complemented by a subsequent paper by the same authors, which describes their implementation of a SIP client adaptor called SipPeer [53]. The adaptor

was designed for use by conventional SIP UAs that could not be modified by inserting modules similar to those described above, but needed to participate in decentralised overlays nonetheless. The client adaptor behaves as an outbound SIP proxy for a conventional SIP UA and helps it establish sessions with P2P SIP UAs.

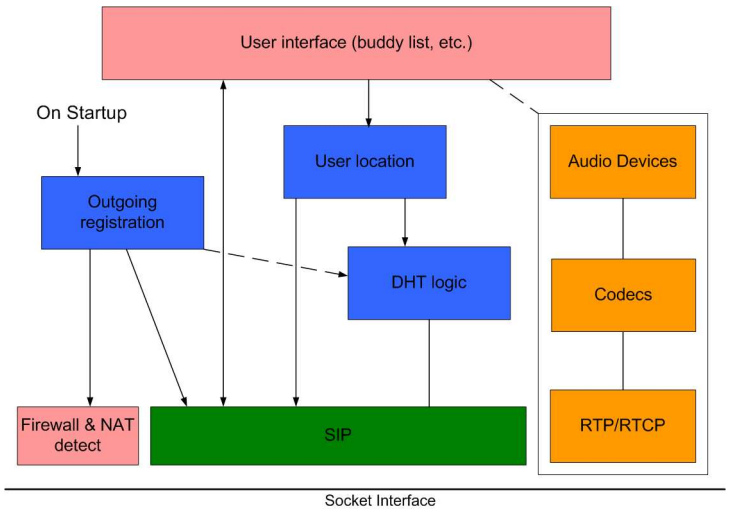

Figure 4.1: Block diagram of a P2P SIP node. Source: [52].

At around the same time, another paper was authored which details a P2P SIP VoIP/IM system known as SOSIMPLE [54]. The paper describes a system which is slightly different from [52]. The similarities include that SOSIMPLE also embeds a DHT, namely Chord, which it uses to provide the DHT functionality. SOSIMPLE also decouples a node registration (orientation into the overlay) from a resource registration (insertion of a contact address binding). That said, however, there are many differences. SOSIMPLE does not define super nodes, but assumes that all nodes equally participate in the overlay. A REGISTER message is modified for peer-to-peer such that new headers are added to it, in order to convey information such as node identifiers, resource identifiers and the overlay name. Resource lookup and session establishment are performed using SIP messages since SOSIMPLE does not have a DHT API interface, unlike [52]. When a node wishes to lookup the location of a buddy, it hashes the SIP AOR to produce a key, and routes a message such as an INVITE to the neighbour with the closest ID to the key. This process is repeated iteratively or recursively in the overlay until the node which hosts the location binding for the intended user returns the actual location address to the callee in a SIP 200 OK response. Thus the final INVITE can be sent directly to the intended user. Figure 4.2 illustrates

how SOSIMPLE supports node joining and session establishment.

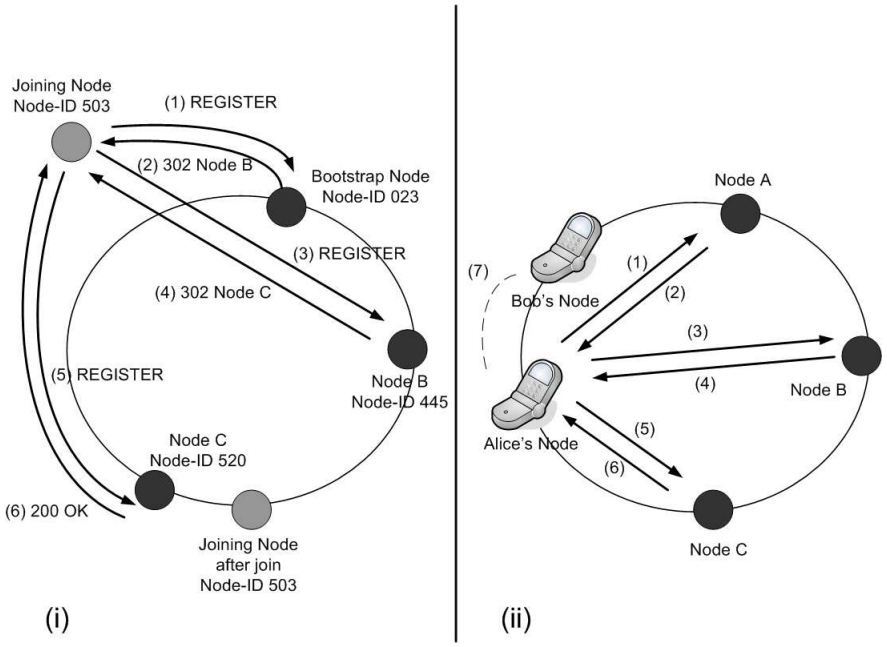

Figure 4.2: (i) Node joining and (ii) session establishment in SOSIMPLE. Adapted from [54].

Although the authors of SOSIMPLE mentioned in their paper that a prototype of the architecture was being tested at the time the paper was published, private correspondence with the authors revealed that they decided ultimately not to release their code to the public. It is possible, however, that SOSIMPLE is used in some of the commercial telephony solutions provided by a company owned by some of the authors [55].

The creation of the SOSIMPLE architecture was important for two main reasons. Firstly, it reinforced the idea that DHTs could be used instead of unstructured protocols as lookup and routing substrates for resource location and session establishment in P2P SIP. Not only was it a proponent of a DHT solution, but it was one of the first in a long line of proposals and prototypes that had a particular bias towards Chord, making it the unofficial DHT of choice. It is likely that due to this early adoption of DHTs, little effort has been invested in the use of unstructured protocols for P2P SIP.

Secondly, it introduced a point that would continue to be debated in the P2P SIP community as to what protocols would be used to maintain the DHT, and what the exact nature of the node operations for lookup and session establishment would be. In [52], SIP is used to maintain the

DHT but node operations such as lookups (executed by super nodes), are supported by a DHT API interface. In SOSIMPLE, SIP is used both to maintain the DHT as well as to perform node operations. These two approaches have come to be known as *P2P over SIP* and *SIP over P2P* respectively in the P2P SIP community. Their benefits are contrasted in section 4.2.2.

4.2 Concepts and terminology

Over time, certain ideas have come into prominence in the P2P SIP community and a degree of consensus has begun to develop around certain key areas. However, members in the group still have different opinions about certain aspects of how the protocols for P2P SIP should behave. An important milestone that needed to be reached was the maturing of the informal group into a formal IETF working group, a step which would give the group more credibility and recognition as they strove to formalise the P2P SIP protocols. The group achieved this goal in February 2007. There is no doubt that the recognition of the group by the IETF was influenced largely by the acceptability of the proposed working group charter [56]. However, second to the charter, a document that similarly vouched for the intellectual capacity of the group is a document known as the *Concepts and Terminology* document [57]. This document has the primary aim of defining a general framework for P2P SIP by utilising many of the design decisions that were already popular in the group, while acknowledging alternative views held by others.

The next five sections will give a summary of the Concepts and Terminology document (hereafter called the concepts document) and at each juncture, will tie in the main items of literature mainly in the form of Internet drafts, that support common protocol behaviours.

4.2.1 Peers and clients

A P2P SIP overlay consists of a number of nodes that collectively provide services for each other. An optional realisation of this model is one whereby peer-to-peer protocols are used to organise the overlay in a hierarchical tree topology, with super nodes and clusters of ordinary nodes. In the P2P SIP lexicon, these types of nodes are called *peers* and *clients*, respectively. Peers are those that provide routing, storage and retrieval services, and clients are those that do not. The concepts document does not indicate that the P2P SIP protocols must enforce this type of segregation, but does specify that where it is employed, only peers must participate actively

	Overlay Maintenance	**DHT Operations**
ASP	`join`,`leave`,`update`	`store`,`fetch`,`find`,`remove`
P2PP	`join`,`leave`,`lookupPeer`, `exchangeTable`,`query`, `replicate`,`transfer`	`publish`,`lookupObject`, `removeObject`
RELOAD	`peer-join`,`peer-search`, `resource-transfer`	`resource-put`,`resource-get`

Table 4.1: Examples of proposed P2P SIP Peer Protocols.

in the overlay and maintain the DHT. Clients do not participate actively in the overlay, and can only make requests by forwarding them to the overlay through their associated peer nodes.

The use of this distinction raises questions as to how the P2P SIP protocols manifest themselves in each type of node. In order to accommodate possible differences between them, separate *P2P SIP Peer* and *P2P SIP Client* protocols have been proposed. At the time of writing, there is still no clear consensus on the issue of peers and clients, and much debate is still ongoing as to how to differentiate the two protocols, or if the client protocol is even necessary at all. There are those who do not distinguish between the two, and suggest a generic protocol definition that is used by all nodes. Examples of such descriptions include dSIP (distributed SIP) [58], ASP (Address Settlement by Peer-to-Peer) [59] and RELOAD (REsource LOcation And Discovery) [60]. At the time of writing, there is not yet a protocol proposal that defines how the client behaves. If the P2P SIP client protocol does formally emerge in future, it is likely to either be a subset of the peer protocol or just pure SIP [60].

A summary of some of the proposed APIs for P2P SIP peers is given in table 4.1. The APIs are divided into two main categories, overlay maintenance and DHT operations. Overlay maintenance refers to the procedures used to manage the DHT itself, whereas DHT operations are those that deal with managing the resources that are stored in the distributed database.

In the overlay maintenance column in Table 4.1, the joining and leaving operations are supported by the appropriate join and leave methods. Methods such as `peer-search` in RELOAD and `query` in P2PP can be used to lookup the locations of certain nodes, while `update` in ASP can be used to send messages to neighbours, such as prompting them to perform some action. It is also possible to manage resources using methods such as `resource-transfer` in RELOAD which re-locates resources to other nodes, such as when a node leaves the overlay and must commit its resources to an appropriate neighbour. The DHT operations normally consist of three methods for inserting, retrieving and removing resources from the overlay, with the exception of

ASP which has two variants of a retrieval method.

The syntax of the protocols outlined in Table 4.1 are quite different, though the semantics are similar. Some of the protocols may appear to be more expressive than others, such as P2PP which has a larger set of operations than the others. However, this is misleading since some of the protocols re-use the same operation to achieve the same purpose. For example, RELOAD re-uses the `peer-join` operation to both join and leave the overlay by varying how the message is constructed.

4.2.2 Protocol layering

In section 4.1, it was explained that there are two main types of protocol layering in P2P SIP, that is P2P over SIP and SIP over P2P. In P2P over SIP, overlay operations are defined as extensions to SIP whereas in SIP over P2P, a lower level transport protocol defines overlay operations [61]. It is the proposals supporting P2P over SIP that seem to be less popular. When justifying the use of P2P over SIP, *Bryan et al* [54] cite the following advantages:

1. It prevents the necessity of forcing applications to incorporate an extra peer-to-peer stack in addition to a SIP stack.
2. There is no clearly defined, standards based solution for implementing the messages for a peer-to-peer system, whereas SIP is a mature, established protocol.
3. Many border devices such as firewalls and traffic shapers already recognise SIP traffic, and often ban peer-to-peer traffic.

Among the recent proposals, dSIP [58] and the peer protocol for P2PSIP networks [62] are based on the P2P over SIP method. The latter introduces a new SIP method called LOCSER that embeds peer-to-peer information in XML-like payloads that are exchanged between peers. The former, like SOSIMPLE, defines new SIP headers and tags to convey peer-to-peer information such as node IDs and hashing algorithms.

P2PP [63] is an example of a protocol that fits the SIP over P2P approach. In this design, a new node discovers the properties of an overlay by sending a `Query` request to an existent node in the overlay, which returns information such as the name of the overlay and hashing algorithms the node needs in order to participate in it. The joining node then sends a `Join` request to a bootstrap node as a request to join the overlay. The `Join` request is routed through the overlay

until a suitable admitting node is found, which returns a 200 OK message to the new node to confirm its insertion. A `Leave` operation removes the node from the routing tables of its neighbours. A `Publish` operation inserts a resource record into the overlay and a `LookupObject` operation retrieves a record. If a node wishes to invite another node for a session, it executes a `LookupObject` request and uses the contact address it retrieves to send a SIP INVITE message to the destination.

The authors of the RELOAD document [60] cite the following advantages of SIP over P2P:

1. The SIP protocol is misused by making it do things it wasn't designed to do, such as resource lookup.
2. The SIP protocol is "too heavy" to be used for a P2P system.
3. Avoiding SIP means that a new protocol can be designed that is built from the ground up with peer-to-peer in mind.

The debate is still ongoing among members of the P2P SIP working group as to which form of protocol layering is the most appropriate. It is however the opinion of this research that SIP over P2P is the better approach not only due to the reasons given above, but also because it offers an interesting separation between service layer and application. The benefits of this separation are discussed in more detail in section 5.2.2. That said, both SIP over P2P and P2P over SIP are designed to enable nodes to implement services in a distributed fashion. The next section describes in detail what those service are and how some proposals provide solutions for them.

4.2.3 Location service

The SIP protocol never standardised the location service, as was described in section 2.2. The advantage of this abstract conceptualisation of the location service is that it allows SIP to be used with various protocols that provide data persistence services. This trend has continued in P2P SIP, where resources have been defined in an abstract way and no association with existing protocols is made with regard to the location service.

A possible solution however, is described in [64] which details a data format and a location service interface for P2P SIP. This document is unique since it addresses the issue of the structure of the location service in connection with a DHT API similar to those that were outlined in Table 4.1. This document describes a location service that is secure from attacks by storing keys in

signed form or making them immutable so they cannot be changed. A key is represented by the XML key element such as,

```
<key>sip:bob@example.net</key>
```

The peer will hash this key, and the key-value pair that will be inserted will resemble the following (the signature element is omitted for simplicity),

```
<key Id="One">
     <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
     <DigestValue>Y8eVClzbJ8Wk315al7pW6/RZsvM=</DigestValue>
</key>
<value Id=”Two” expires=”2006-12-31T18:00:00Z” nonce=”82771583613">
   <contacts xmlns=”urn:ietf:params:xml:ns:p2p-sip”>
      <contact displayName=”Bob W.” expires=”2006-12-31T18:00:00Z”>
         sip:bob@192.1.2.3:5060
      </contact>
   </contacts>
</value>
<Signature>
….
   <KeyInfo><KeyName>bob@example.net</KeyName></KeyInfo>
</Signature>
```

4.2.4 Distributed database function

Irrespective of how it is structured, a location service provides access to the resources needed for successful session establishment in the overlay. There are two particular models that can be followed to achieve this: the data model and the service model [65]. In the data model, any node is able to access the resources for itself, whereas in the service model, a node must first locate a service node which is responsible for a given resource.

In the data model, a user named Bob inserts his contact address under his SIP AOR sip:bob@ru.ac.za when he registers with the overlay. When the user Alice tries to create a session with Bob, Alice will first hash Bob’s SIP address to obtain a key, and use a DHT operation to obtain Bob’s contact address. This method is the easier of the two models to implement, but is not well adapted for locating advanced services in the overlay (advanced services are the subject of section 4.2.5).

In the service model, instead of trying to obtain the SIP AOR mapping herself, Alice performs a lookup of the service node for Bob's resources, and sends the SIP message directly to this node. This special node acts as a SIP proxy for all the nodes for which it is responsible for. The service model is based in part on the data model, but is viewed as more extensible than the data model, due to its support for certain nodes to provide more advanced services to the overlay besides routing and storage.

4.2.5 Advanced services in the overlay

Nodes in a peer-to-peer overlay offer services such as routing, storage and retrieval of resources. These are the basic services that are necessary to support overlay operations. Apart from these basic services, peers can also offer more advanced services. An example is the discovery of peers that are behind NATs and firewalls. Due to the use of NAT and IP masquerading, the discovery and reachability of nodes in the overlay becomes difficult for processes such as correctly populating routing tables, performing lookups and establishing sessions. One approach that attempts to solve this problem utilises peers which possess public IP addresses to provide NAT traversal. Peers would maintain connections between themselves, forming a super overlay, with clients establishing sessions through their respective peer nodes. This however relies on the presence of numerous peers with public addresses and fails when all nodes are behind NATs [57].

There is a second approach that has been favoured by members of the P2P SIP working group and accepted by most P2P SIP protocol proposals. The consensus in the group has been to employ the use of Interactive Connectivity Establishment (ICE) [66] which is a technique for NAT traversal based on the offer/answer model [67], which is a protocol for negotiating media sessions using SDP (see section 2.4.1). ICE works in concert with Simple Traversal of UDP over NATs (STUN) [68], which is supported by some NATs, and allows nodes to discover the type of NAT it is behind and the public address of its outgoing connection. This allows peers to test and select valid IP address and port tuples in SDP offers and answers.

Another example of an advanced service in a P2P SIP overlay is offline message and voicemail storage. If a peer node that has been participating in the overlay signs off, any communication intended for that user will fail. Specifically, the lookup operation that is executed by the calling node will fail to retrieve the location binding for that user, since when that user signs off, their registration record is removed from the location service. It is possible for one or more of the peers that have high capacity storage capabilities to offer offline message storage and voicemail services for peers that received calls whilst they were offline.

Irrespective of what service an overlay offers its nodes, the P2P SIP protocol must provide the ability for the peer nodes to be able to advertise those services and in turn, the ability for other peers to be able to discover those services that are being offered. One possible solution for advertising services is to recycle the SIP addressing scheme to create a set of standard service addresses. For example, it is possible to create a STUN service binding that relates a contact address or addresses that map to the SIP address sip:stunserver@p2pdomain.org. Nodes that desire to make use of this service can execute lookup requests to retrieve the locations of STUN servers and subsequently use ICE to establish direct connections with other nodes, such as nodes to be inserted as neighbours in a routing table. This naming convention could be applied to other services such as offline message storage.

The data format described in section 4.2.3 can provide a more sophisticated way of handling this problem. By using a DHT key such as offline:bob@example.net, a retrieve operation for this key can be executed when a Bob's UA becomes active, and can retrieve messages that were stored in the overlay in his absence.

4.3 P2P SIP implementations - SIPDHT

The protocols that are to be used for P2P SIP have not yet been standardised, though there is growing consensus in the areas described above. It is quite likely that implementations are being developed and tested behind the scenes in academic institutions and companies, but there is a clear lack of code that has been made available to the P2P SIP community. At the time of writing, the only project that has been released to the public, is called SIPDHT, and is described in this section.

SIPDHT is an open source project that was started by an active member of the working group, Enrico Marocco from Telecom Italia, and others in May 2006 [36]. The aim of the project is to provide running code for the working group which implements some of the design decisions that are under debate and provide feedback to the rest of the members. The software is now in its version 2 release.

The initial release was based on an earlier proposal for distributed SIP registration and resource location [69] and employed a simple text based interface for creating virtual nodes on a single host machine or on a network. It embeds a SIP stack to provide SIP functionality and uses a modified implementation of the CAN protocol described in section 3.2.1.3 called pCAN (passive CAN), to provide the distributed hash table functionality. The modified CAN algorithm is called

passive since nodes don't participate in the DHT by default, but must be invited to do so by another node that is already in the overlay. The API exported by pCAN is described below:

`PUT` Inserts a key-value pair into the DHT.

`GET` Retrieves a value associated with a key.

`JOIN` Sent by a joining peer to a node in the DHT as a request to join the overlay.

`QUERY` Queries the status of a peer in a given pCAN zone for a given key. The request is routed in the DHT until the node responsible for the key is located.

`UPDATE` A message exchanged with a node's neighbours when the state of that node changes.

`TAKEOVER` Sent by a node to a neighbour as a request for the neighbour node to add that node's zone to its zone of responsibility.

The maintenance of the DHT and session establishment is aided by a protocol called eXtensible Peer Protocol (XPP) [70]. XPP is a binary transfer protocol which uses UDP for transport and is designed to be NAT friendly by using ICE and STUN servers to establish connections between peers, using simultaneous session establishment through SIP or SDP. XPP uses keep-alive messages to regularly refresh NAT bindings between peers. Figure 4.3 shows how a typical XPP session is established between two users.

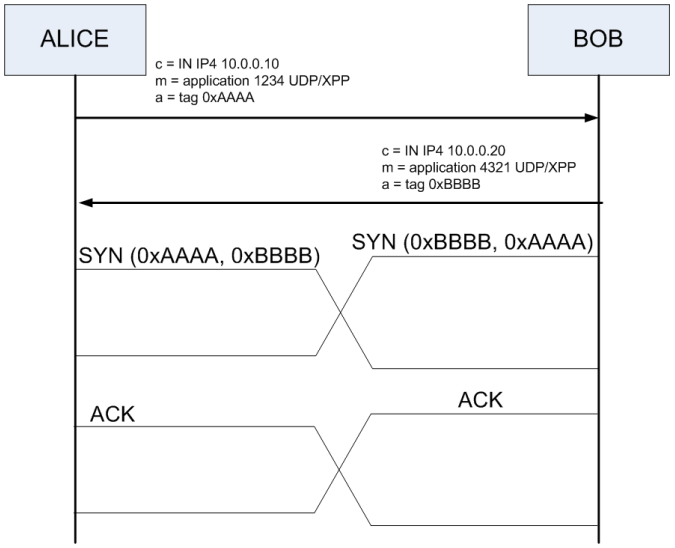

Figure 4.3: An example XPP session between two users. Source: [70].

4.4 Summary

The P2P SIP working group within the IETF is driving the effort towards the standardisation of protocols for P2P SIP. There is a wealth of proposals that have been produced by members of the group, often describing very different behaviours of P2P SIP entities, which reflects contrasts in the way people conceptualise the protocols. There is however, a growing consensus in some core features, and it is possible to use some of the areas of agreement, to sketch an outline of what the standardised protocol may look like in future.

The following chapter begins the next phase of this thesis by describing the design of a candidate peer-to-peer framework for SIP which has been the focus of the work reported in this thesis. The framework is based on a SIP over P2P approach and it is possible to discern how some of the ideas that have been presented in this chapter have had an influence on the design. However, the design is novel in nature and represents a new contribution to the debate on the P2P SIP protocols.

Chapter 5

Designing the Peer-to-Peer layer

> *The introduction of suitable abstractions is our only mental aid to organize and master complexity*
>
> - E.W. Dijkstra

The previous chapter provided an overview of the most popular protocols and algorithms that have been proposed for P2P SIP, some of which are likely to feature in the final standard. The work undertaken towards this project was strategically placed at a time of high activity in the P2P SIP working group, and as such, was able to benefit from exposure to several design options for a possible framework for a peer-to-peer system for SIP. At the time of writing, there are several protocol proposals that describe the higher level functionality of a P2P SIP system, but there is a lack of design implementations that can be used to test popular ideas and provide results that can be used to better inform the standardisation process. The development of such a system is the main objective of this project, but a good implementation must originate from a good, well structured model.

This chapter describes the design of a modular peer-to-peer framework for decentralised object storage and location based on either structured or unstructured protocols. The original intention was to use this framework in a P2P SIP setting, but it is believed that it is possible to use it with a broader range of applications beyond telephony based on SIP. The modularity property manifests itself in discrete, functional layers, most of which can have different physical implementations. Much like an extension cord that has many sockets to power many devices, the model supports the use of many types of overlays, and as such the model has been dubbed *OverCord*.

5.1 OverCord: A modular, layered framework

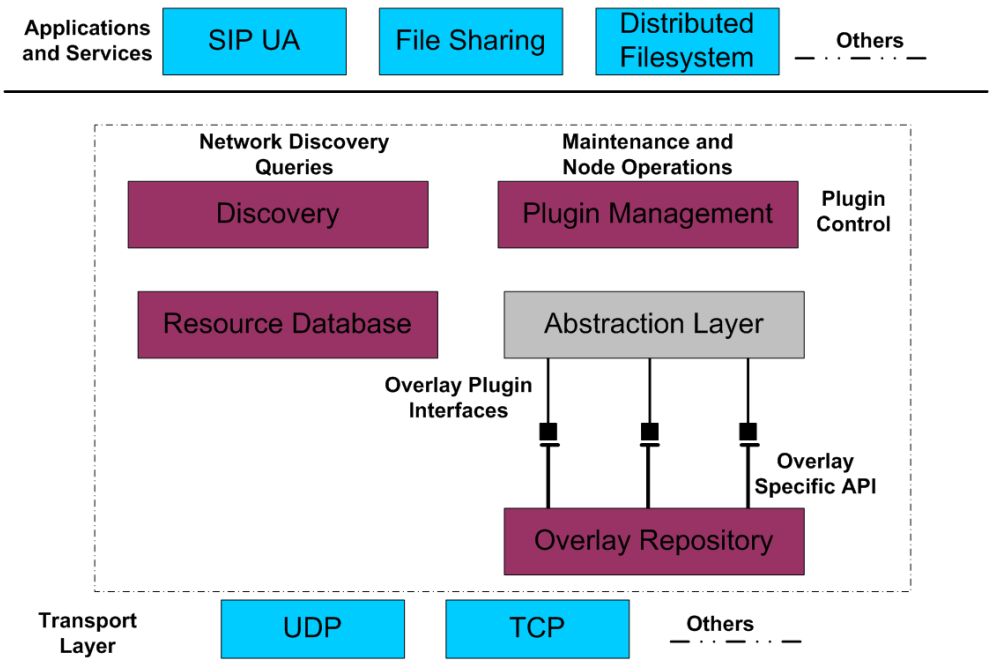

Figure 5.1: The layered architecture of the OverCord framework. Adapted from [71].

The architecture of OverCord is shown in Figure 5.1. The applications and services are located at the topmost layer, and access the lower layers in a transparent fashion. The core of the OverCord framework consists of the resource database, discovery, plug-in management, overlay plug-in and overlay repository layers, which are contained within the bounding box in the figure above. The various layers are described in the next six sections.

5.1.1 Discovery layer

The discovery layer assists the application to discover or join overlays, by communicating and establishing connections with live nodes in those overlays. In previous chapters, it was emphasised that the different types of overlays rely on certain mechanisms that allow them to search for, discover and join an overlay through a special node called a bootstrap node, once the locus of that node has been obtained. The discovery layer therefore acts as a *helper* layer to the other overlay-aware elements of the system, by using other protocols and mechanisms to discover

potential bootstrap nodes.

This layer is critical because the system's ability to function relies almost entirely on the successful discovery of overlays. In order to improve the discovery layer's effectiveness, instead of using a single protocol, several protocols and mechanisms can be incorporated into the discovery layer in order to provide a multiplicity of options for the node to choose from. The discovery layer can therefore be described as a storehouse for many types of protocols and mechanisms for performing discovery, which can be initiated in parallel or sequentially, depending on the implementation. The greater the number of options, the greater the probability of finding many candidate bootstrap nodes. Also, when more results are obtained, the chances increase of being able to successfully join the overlay. This is important in situations such as a node crashing or becoming unavailable before it can completely service the join request, whereby other candidate nodes may be attempted afterward. Figure 5.2 shows a possible internal structure of the discovery layer in an OverCord node. OverCord may contain an initial set of discovery mechanisms, but it is designed to allow developers to add others afterwards.

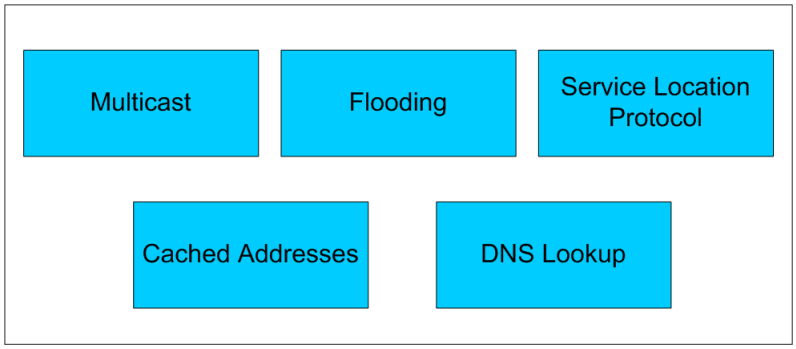

Figure 5.2: A possible construction of the discovery layer.

5.1.2 Resource database

The resource database is an abstract storage space for resources that are shared in the overlay. No physical implementation is tied to this layer, and it can take on many different forms. It is expected that developers would implement a common data format that defines how the resources in this layer are stored across all nodes in the system. The database layer does not enforce any constraints that pertain to how data is stored, retrieved, updated or removed. It is up to the developer to reconcile the storage and the data access protocols in a common way across all OverCord nodes.

5.1.3 Overlay repository

The overlay repository is reserved for all structured and unstructured protocol implementations that are used by OverCord to provide the routing and lookup services in the overlay. OverCord's agnostic approach towards the overlays it interacts with allows for *overlay pluggability*, which means that different protocols can be inserted into the system, so that the user agent that embeds OverCord can participate in several different overlays.

The results obtained through the discovery procedure are particularly relevant to the overlay repository layer, as the overlays need this information to accomplish tasks such as joining. Figure 5.1 shows that each overlay has its own properties and APIs, and as such, expects to interact in a standard way with any application that is layered over it. This poses a challenging problem in that the application must know how to communicate appropriately with the underlying overlays. Section 5.1.4 explains how OverCord solves this problem through the use of a generic interface.

5.1.4 The generic interface

The effect of overlay pluggability is that the manner in which OverCord interacts with the overlay repository layer can change depending on the specific overlay which it needs to communicate with at any given time. "Hard-wiring" the procedures to be followed on a per overlay basis would seriously hamper the scalability of the system, as a change in the core of OverCord would need to be made every time a new overlay is incorporated into the system. A better design would be to define a standard way in which OverCord interacts with the embedded overlays, in a manner which is independent of the type of overlay that is being used.

OverCord provides this functionality in the form of an abstraction layer. The layer manifests itself in a generic interface which is used across all types of overlays. Since the protocol that determines the interaction is defined in one single place, and can be reused across all overlays, the full benefit of overlay pluggability can be realised. It is possible to draw on the protocols that have been proposed by members of the P2P SIP working group to compose this interface, since those protocols cover many of the types of interactions that an application must have with the overlay. The outline of the generic interface is given in the listing below, defined in pseudocode:

```
define interface:
    define Overlay Management Methods:
        joinOverlay(bootstrapNode)
        leaveOverlay()
        updateStatus(type, value)
    define Node Operations:
        put(key, value, options)
        get(key, options) returns vector
        remove(key, value, options)
```

In the pseudocode above, the `joinOverlay` procedure is used to insert the new node into an overlay, given the locus of an existing bootstrap node. The `leaveOverlay` procedure is used to exit the overlay, and restore any resources to the rest of the overlay in a manner defined by the overlay itself, and not by OverCord. The `updateStatus` procedure allows designers to define a set of network related messages identified by a message type which takes on a specific value that conveys information for maintaining the overlay. The `put`, `get` and `remove` procedures are responsible for inserting, retrieving, updating and removing resources from the location service respectively. A `put` operation indicating a key that already exists in the overlay is interpreted as a request to update the value associated with the key. The options parameter allows designers to define options such as a TTL (time to live) for a resource or security related information.

5.1.5 Plug-in layer

The interface described in the previous section is detached from any kind of overlay. There needs to be a component in the system that makes use of this interface and is able to translate between the API it exposes to the application, and the corresponding proprietary APIs exported by the underlying overlay implementations. OverCord achieves this through the plug-in layer. The plug-in layer is a repository for software constructs known as plug-ins that help interface between the overlay and the application. Figure 5.3 illustrates the interplay between the plug-in layer and the underlying overlays.

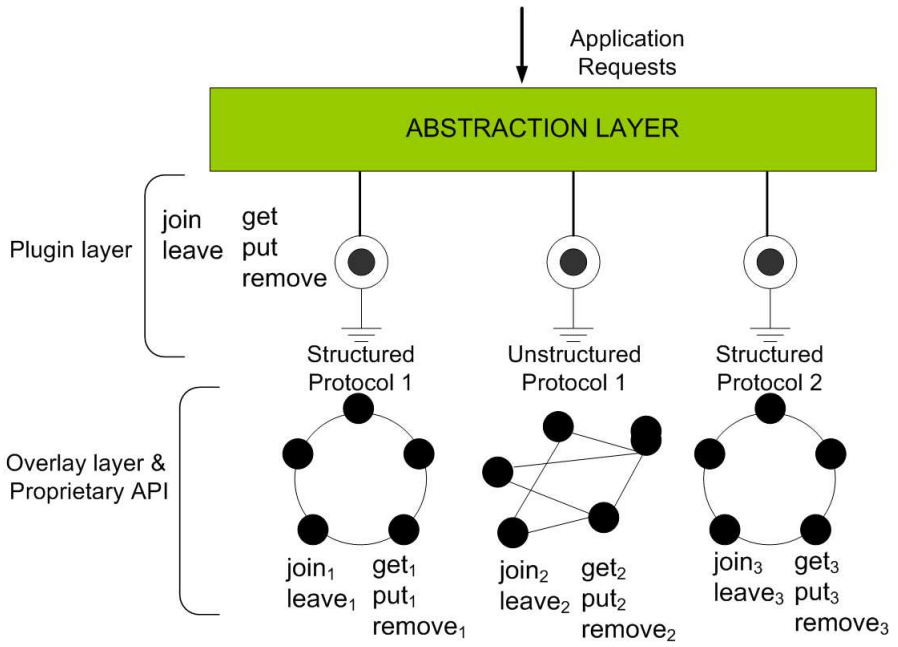

Figure 5.3: Plug-in layer and overlay layer with proprietary API.

A plug-in can be associated with one or more overlays, which means that a single plug-in can be used to join different overlays, albeit one at a time. Where there is such a one-to-many mapping of plug-ins to overlays, it is anticipated that the different overlays would have to be very similar, and have similar internal mechanisms, since the plug-in would use a common instruction set to interact with the overlays. Such a situation would arise in instances such as where there exists two implementations of a single structured or unstructured protocol that have similar APIs, but represent different release versions of the software, whereby perhaps one has more advanced features than the other, such as efficiency. OverCord also allows a user to join more than one overlay by using different plug-ins operating at once.

It is possible to define properties for plug-ins so that they can provide descriptive information about themselves. Many kinds of methods are possible, some foreseeable ones are given in the listing below, as an extension to the interface defined in section 5.1.4:

```
define interface:
    import Management Methods
    import Node Operations
    define descriptors:
        getVersionNumber()
        getOverlays() returns vector
        getBootstrapHost() returns vector
```

The `getVersionNumber` procedure is a way of distinguishing the release version of a software plug-in, which may be important for compatibility reasons if a plug-in has been modified. It is anticipated that participants of an overlay will always use a common version of a plug-in, or at least versions of a plug-in that do not introduce anomalies into the system, such as the data format of the resources database. The problem of performing software upgrades for a distributed system such as OverCord is considered out of scope, but one possible solution is discussed in [72]. The `getOverlays` procedure is able to return all the overlays that this plug-in is compatible with. The last procedure returns details about the hosts that were discovered by the discovery layer, and will be attempted when joining the overlay.

5.1.6 Plug-in management

The plug-in management layer performs all administrative functions regarding the management and use of plug-ins in OverCord. There are four main functions that it is responsible for performing:

Plug-in Detection The plug-in management layer must be able to detect the different plug-ins hosted within the OverCord system. It does so by browsing through all the overlay implementations contained in the overlay repository, and extracts details about each. It can choose to display information about the plug-ins to a human user, for example. The detection ability is important in order to perform plug-in verification, a process which is explained next.

Plug-in Verification When the responses to a discovery probe reach an OverCord endpoint, the plug-in management layer must determine which of the overlays in those responses correspond to plug-ins that are supported by the node. It does so by comparing tokens

in the responses against the properties of the individual plug-ins, and can then relay any matches to the application interface so a user can select the overlay they would like to join.

Plug-in Creation It cannot be known before-hand what plug-in will be used when the application starts since this information can only be made available when the user has selected the overlay they would like to join, and an appropriate plug-in has been identified for use in that overlay. Thus when the user has selected an overlay to join, the correct plug-in must be created at runtime. The plug-in management layer must be able to handle this dynamic requirement.

Plug-in Control Once a plug-in has been started, the plug-in management layer must be able to exercise control over the plug-in. All the overlay maintenance and node operations must be passed to the active plug-in through this layer, which is the only layer that has direct access to the plug-ins themselves.

OverCord is a modular framework, intended to support any application that makes use of a distributed location service based on either structured or unstructured protocols. Peer-to-peer Internet telephony based on SIP, is an example of one such service. This assertion is justified in the next section, where the architecture of OverCord is discussed with specific relevance to P2P SIP.

5.2 Analysis of the OverCord framework

The next few sections describe how the modules of OverCord work together to provide the core features and services that constitute a P2P SIP node. An attempt is made to relate the features of OverCord with the protocols that will probably be standardised by the P2P SIP working group, such as those stipulated in the concepts document (see section 4.2).

5.2.1 Peers and clients

In future, a P2P SIP network may be divided into special nodes called peers which participate actively in the overlay, and clients who rely on peers for the provision of services. The aim of OverCord is to support any implementation choice, regardless of whether or not the developer supports the separation of nodes into peers and clients. In this section, we describe how OverCord can be used to support both modes of operation.

The trivial case is where there is no distinction that is made in the types of nodes in a P2P SIP overlay, such that all nodes are peers. OverCord is built with peer functionality in mind, and as such nothing is affected by this design choice: simply, all nodes in the overlay have full access to the interface defined in section 5.1.4.

Where a distinction is made between nodes, two scenarios are possible. The first is where peer nodes are defined to be those that embed the OverCord system and clients are those that do not, such as conventional SIP UAs. In this setting, the unmodified SIP UAs would need to use the proxy and location services of the overlay. In the absence of a central SIP server, the conventional UAs could be manually configured (or discover OverCord peers using the sip.mcast.net address, or using other protocols) to point to an OverCord node, and use it as its SIP proxy server. This is similar to the SIP client adaptor described in section 4.1. In this way, a P2P SIP overlay consists of OverCord nodes as peers and conventional SIP UAs as clients. In this instance, the P2P SIP client protocol is just the SIP protocol itself.

The other scenario is where all nodes embed the OverCord system, but there is a distinction in the way the system is used between those that are peers and those that are clients. Clients by definition do not contribute resources to the overlay, but rely on peers to perform resource management on their behalf. This would mean that the resource database layer in client nodes would be empty, or non-existent. Also, because OverCord treats the client protocol as a subset of the peer protocol, it would be possible to replace the definition of the interface in the abstraction layer with a modified version. This, however would have a cascading effect on the overlay plug-ins and would necessitate the use of different kinds of plug-ins in the same overlay between peers and clients. A neater design would be to keep the interface and the plug-ins intact, and rather have a *switching* functionality that switches the action mode of a plug-in from a *peer mode* to a *client mode*. The switch could be controlled by the plug-in management layer, which could control the mode in which the plug-in operates. A user action supported in the application interface could cause this change in plug-in mode to occur.

5.2.2 Protocol layering

P2P SIP can either be achieved by modifying the SIP protocol to convey peer-to-peer information, or by servicing SIP by using a separate peer-to-peer protocol stack. One of the aims of OverCord is to be extensible to many different applications. For this reason, OverCord is more suited to an approach that layers services, such as those provided by SIP, over a peer-to-peer layer. In so doing, the higher level application can be substituted for another type of applica-

tion, without changing the underlying layer. This is not so where peer-to-peer information is embedded in SIP messages because in this case, the two protocols are merged together in such a way that they cannot be decoupled. This results in a protocol that cannot be reused for purposes beyond communication based on SIP. Furthermore, it is believed it would be more advantageous for the P2P SIP protocol to incorporate two separate protocol stacks, a peer-to-peer stack and an unmodified SIP stack, as it would make it easier to reconcile P2P SIP with pure SIP.

5.2.3 Location service

The location service for P2P SIP stores resource bindings that are needed for session establishment, such as SIP AOR to SIP contact address bindings. Other forms of data that may need to be stored includes binary data such as audio voicemail messages, if the overlay supports offline voicemail storage. In OverCord, this ability is provided by the resource database layer, which represents an abstract repository for the storage of resources that are shared across the overlay. Developers are free to implement this layer in any way they wish, but the solution would have to take into consideration the storage space limitations on the endpoints.

5.2.4 Data and service models

The basic services in a P2P SIP overlay are routing, storage and retrieval. OverCord provides these services through the plug-in infrastructure and the resource database layer. These basic services are the foundation for session establishment, but there are different approaches to using these basic services to support session establishment, as explained in section 4.2.4.

In the data model approach, each peer has access to the DHT and can retrieve resource records from it for session establishment. OverCord supports the data model approach using the embedded plug-ins that implement the DHT operations defined in the abstraction layer, to allow peers to manage keys. A peer can then use the contact addresses extracted from the retrieved resource to send the SIP message to the appropriate destination, or destinations.

In the service model approach, a peer must first locate the service peer that is responsible for the destination user's resources. OverCord supports this model as well through the embedded plug-ins which can be used to locate the service peer through the lookup operation which maps a SIP AOR to the location of the service peer. The plug-in layer can return this location to the SIP layer, so that a SIP message can be constructed and sent to the service peer. When the SIP

message reaches the service peer, the service peer can then use a DHT operation to retrieve the key in the resource database and either redirect the caller to the appropriate destination or use the location information to proxy the SIP message to the destination on the caller's behalf.

5.2.5 Interoperating overlays

Section 3.2.2 described a paper that presents a common API for structured overlays, that involves breaking down the structured protocols into a simpler key based routing (KBR) API, to which all structured protocols reduce to. Other more sophisticated protocols could similarly be reduced to the abstractions provided by the protocols that were based on KBR, and so on [39]. This reduced API would be used to build applications that were DHT agnostic, and would access the underlying layers in a transparent fashion.

The OverCord design provides an alternative solution to the problem of allowing decentralised applications to utilise various types of overlays. The solution that OverCord proposes is also based on an abstraction, which is realised in the generic interface. However, instead of achieving the abstraction at the tier 0 level (see Figure 3.5), the abstraction is achieved between the tier 1 protocols and the tier 2 services. The plug-in layer can be regarded in this respect as a tier 1 abstraction.

The disadvantage with the OverCord framework is that it misses out on the lowest common denominator effect of the KBR API, and as such has to exhaustively implement plug-ins for each type of overlay. This is in contrast with the KBR API, which can be viewed as a super plug-in for all things DHT. However, the advantage that is achieved in OverCord is that overlays with special features in their native APIs are allowed to retain their special functionality, whereas the KBR API can, in certain circumstances, stifle particular characteristics of certain protocols [39].

5.3 Summary

The development of a standard P2P SIP protocol relies on established ideas that ultimately solve the problem of decentralised SIP. In order to complement the work that has been done by the working group on paper, a design was needed that addresses most of the core areas of consensus in order to give a possible framework for a P2P SIP system. The OverCord model achieves this since it extends the applicability of the framework to other services beyond SIP telephony, provides a modular design that can take on many physical forms and also encourages development

from third parties to provide plug-ins for overlay implementations that are already available. The next chapter details one possible implementation of OverCord. To prove its viability, this implementation was used to develop a peer-to-peer system for SIP, as will be described in chapter 7.

Chapter 6

Implementing the Peer-to-Peer Layer

> *First comes thought; then organization of that thought, into ideas and plans; then transformation of those plans into reality.*
>
> - Napoleon Hill

This chapter details an example implementation of the OverCord framework. The aim of the software is to provide a proof of concept system that testifies to the modularity and flexibility of OverCord. An implementation of the discovery layer is presented which uses a combination of multicast and unicast. The overlay repository consists of two well known DHT implementations, OpenChord and Bamboo. Plug-ins were developed for each of these DHTs that implemented a common version of the generic interface that has been proposed. A plug-in management component was developed that shows how the plug-ins can be dynamically created and controlled. Lastly, a simple application is described that creates OverCord nodes, and shows how an application such as a SIP user agent can issue commands to the plug-in layer in a way that is independent of the underlying DHT.

6.1 Overview

The implementation that is described here followed a bottom-up design that began with the identification of suitable DHT implementations that could be incorporated into the system. After two candidate DHT implementations had been selected and a thorough understanding of them had been achieved, plug-ins could be developed for each. A simple command line version of the plug-in management layer could then be realised was able to detect embedded plug-ins, allow the selection of a plug-in to use, dynamically create the plug-in and issue standard commands to it.

Once this had been established and tested, it was possible to create several virtual nodes on a single machine to test the ability for communication. Again, the behaviour of the system was monitored. Afterward, a discovery capability was added to the system to allow nodes to generate multicast requests so that they did not need to be manually fed bootstrap node locations, but could discover them themselves. Lastly, these same interactions that were performed on a single machine were attempted and tested between hosts on a Local Area Network (LAN).

The rest of the chapter is dedicated to describing in more detail how each layer was developed and also explains some of the design decisions that were taken. Appendix A shows output of the sample application that was built over OverCord. Results of operations such as insertion and retrieval of data sets are clearly labelled.

6.2 Identification of overlay implementations

There has recently been a keen interest in the practical use of DHT protocols. This is evidenced by the existence of projects such as IRIS (see section 3.2.2) that are dedicated to researching large scale, fault-resistant, distributed systems that are based on DHTs. As a result, there are a number of DHT implementations that have been released for public use in the form of open source software. In order to populate the overlay repository layer, these needed to be investigated to determine their suitability for OverCord. Moreover, more than one DHT implementation was needed to fully prove the pluggability claim. The suitability of the DHTs that would be imported into the system would be determined by the degree of ease of using these protocols together (i.e. operating system, software dependencies, programming language and other factors). At the end of the investigation, two Java based DHTs were chosen, OpenChord and Bamboo. The two DHT implementations are described in the next section. Afterward, the steps used to develop plug-ins

for each are detailed.

6.3 DHT implementations

6.3.1 OpenChord

6.3.1.1 Overview

OpenChord is a Java based, open source implementation of the Chord protocol which is available for free under the GNU General Public license. It was developed at Bamberg University in Germany by the Distributed and Mobile Systems Group. Originally released in January 2006 under version 1.0.0, development on the OpenChord implementation continued and is now, at the time of writing, at version 1.0.4. The architecture of OpenChord is given in Figure 6.1. A detailed description of these layers follows, derived using information gathered from the manual [73] and knowledge gained through experiments with the code.

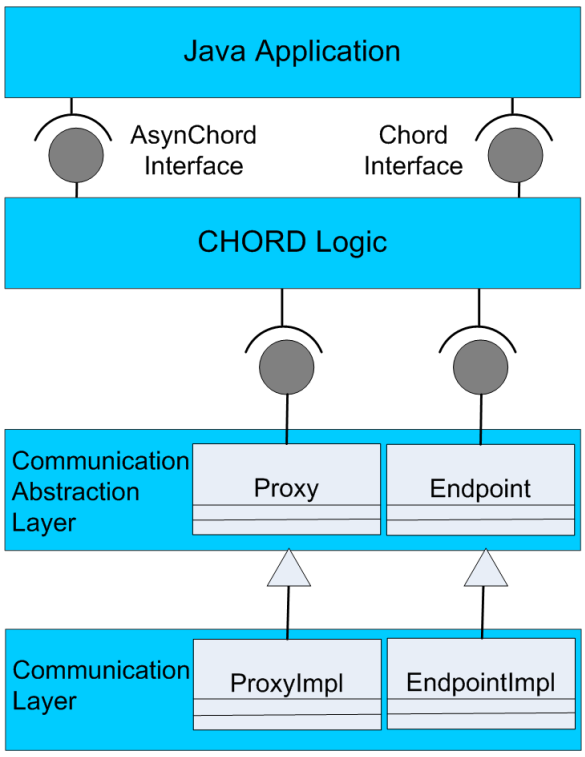

Figure 6.1: Architecture of OpenChord. Source: [73].

6.3.1.2 Communication Layer

The communication layer is the lowest layer in the architecture and forms the basis of the communication between nodes in an OpenChord network. One of the aims of OpenChord is to allow a wide range of communication protocols to be used, depending on the preferences of the developer. However, OpenChord natively supports two forms of communication: one is thread based, where an OpenChord network is run on a single machine; the other uses Java Sockets for nodes that must communicate over a real network such as a LAN. The protocol used by a node is determined by its URL which begins with the string `olocal` or `osocket` for thread based and socket based communications respectively.

6.3.1.3 Communication Abstraction Layer

The communication abstraction layer provides two classes, `Proxy` and `Endpoint` in order to abstract from the underlying communication protocols. This layer provides factory methods for `Proxy` and `Endpoint` to create instances of themselves for specific communication protocols. An instance of the `Proxy` class references all the remote nodes with which the local node communicates. An instance of the `Endpoint` class on the local node provides a connection point for remote endpoints to communicate with the local node. Both the `olocal` and `osocket` protocols described above have their own associated implementations, and developers who wish to add their own communication protocols must naturally create new implementations of these.

6.3.1.4 Chord Logic

This layer is responsible for providing some of the algorithms and procedures described in the original Chord protocol specification such as how to locate successors of the local node, how to replicate data to successors, and other forms of overlay maintenance. This layer is kept neatly independent of the application interfaces defined in the next section which OpenChord exports to applications layered over it.

6.3.1.5 AsynChord and Chord Interfaces

Figure 6.2 illustrates how OpenChord provides Java applications with two types of interfaces to the Chord logic, `AsynChord` and `Chord`. The difference between the two interfaces is

that `Chord` provides synchronous methods whereas `AsynChord` provides asynchronous, non-blocking methods to do the same operations. The methods provided by the two interfaces support overlay management (i.e. joining and leaving) and DHT operations (i.e. inserting, retrieving and removing keys) and internally handle connection management between peers (i.e. neighbours that have crashed and are no longer accessible). Figure 6.2 shows that most of the methods provided by the interfaces are semantically similar.

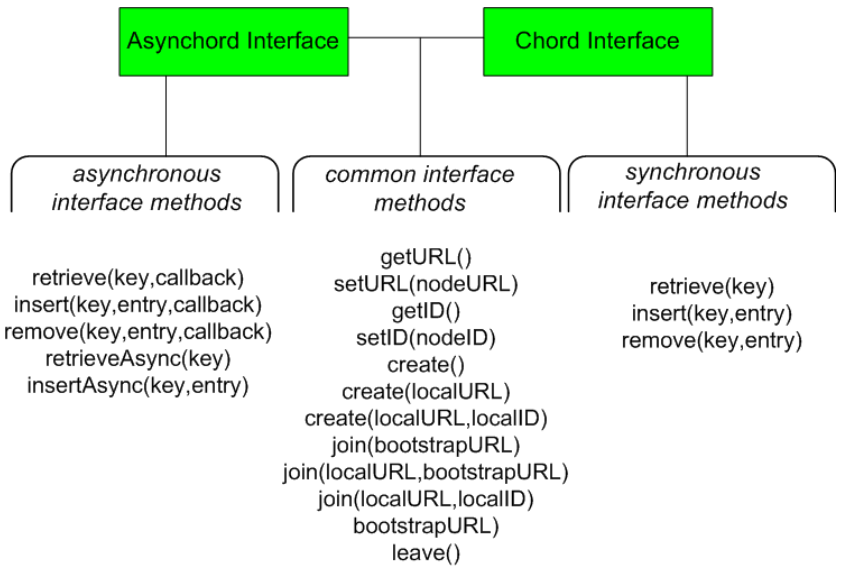

Figure 6.2: Summary of AsynChord and Chord interfaces. Derived from [73].

In an OpenChord overlay, each node has a unique URL which is created by appending the communication protocol type (i.e. `olocal` or `osocket`) to the IP address and port number of the node. A custom hash function is used to generate a unique node ID for the local node by using the URL string. The `create` methods are used to create OpenChord overlays and are invoked only by the first node in the overlay. The `join` methods allow new nodes to join a currently existing overlay. When a new node joins an existing overlay, it must pass the `join` method a string representation of the URL of the bootstrap node it wants to use. A node calls the `leave` method to exit the overlay.

The two interfaces have different methods for the DHT operations due to the fact that one interface provides synchronous methods and the other asynchronous. `Chord` uses basic `retrieve`, `insert` and `remove` methods, each blocking the thread that invokes them. The methods of the `AsynChord` interface are divided into two types. The aim of each type is to determine the suc-

cess or failure of each DHT operation performed. The first set of methods achieves this through the use of an instance of the `ChordCallback` interface, which is notified when an operation has completed, and throws an exception if there was a failure. The second set returns an instance of the `ChordFuture` interface, which returns a boolean value through its `isDone` method, to notify the thread whether the operation has been completed or not, and throws an exception if the operation ultimately fails. Both interfaces store resources in memory.

6.3.2 Bamboo DHT

6.3.2.1 Overview

Bamboo is a mature DHT loosely based on the Pastry protocol. It is similar to Pastry in terms of the pattern of the neighbour links that are formed in a overlay, but uses different joining and neighbour management algorithms [33]. It is claimed that these differences allow Bamboo to perform better than Pastry in certain situations such as limited bandwidth environments and where there is high churn [33]. Like OpenChord, Bamboo is also Java based and was popularised by its adoption in the OpenDHT project described in section 3.2.1.2. The development of the protocol implementation was done largely as part of a PhD research project at the University of California (Berkeley), with additional development done by members of other projects.

6.3.2.2 Sandstorm Event Driven Architecture

The design of Bamboo is based on the Sandstorm Event Driven Architecture (SEDA) . SEDA is an alternative form of design targeted at large, highly concurrent systems [74, 75]. Traditionally, services such as the Web and FTP were powered by high capacity servers which use a threading model to service client requests whereby a new thread is spawned for each request. These types of systems are known not to scale well and generate significant overhead, lowering performance as the number of threads increases. SEDA applications are reported to perform better than applications conforming to traditional service designs, and are robust to large variations in load [75].

6.3.2.3 Bamboo Stages

An application for Bamboo is implemented as a stage in the event driven architecture. When an event is dispatched to the event driven server - which is the main thread on the Bamboo node - it

is passed to all stages that have been registered with the server. An application can receive events it registers to receive through an `eventHandler` method and can also dispatch events either for the local node or remote nodes with a `dispatch_later` method [76]. A Bamboo stage has three basic parts:

Constructor This initialises the data of the stage and registers events the stage wants to be notified of.

Event Registration This occurs in the `init` function of the stage, which fetches configuration parameters set in the configuration file.

Event Handling This occurs through the `handleEvent` method of the stage, which is called every time an event that the stage has registered for arrives and needs to be handled.

6.3.2.4 Configuration File

When a stage is registering for events, there a number of parameters that can be fetched. The values of these parameters are set in a configuration file with a set of mandatory and optional tags that resemble XML markup. The format of the file is borrowed from another project at Berkeley called Oceanstore [77]. An example of a configuration file is given below showing some standard Bamboo stages as well as a custom stage called SimpleStage:

```
<sandstorm>
    <global>
        <initargs>
            node_id localhost:3200
        <initargs>
    </global>
    <stages>
        <Network>
            class bamboo.network.Network
            <initargs>
            </initargs>
        </Network>
        <Router>
            class bamboo.router.Router
            <initargs>
              gateway_count 1
              gateway_0 localhost:3200
            </initargs>
        </Router>
        <SimpleStage>
            class bamboo.apps.SimpleStage
            <initargs>
              debug_level 1
              mode sender
            </initargs>
        </SimpleStage>
    </stages>
</sandstorm>
```

In this configuration file, there are several portions to note:

Global Variables Global variables accessible by all stages are contained in the `global` tag.

This section defines the node identifier of the Bamboo node, which is running on the localhost at port 3200.

Initialisation of Arguments All settings of stage specific parameters are wrapped in an `initargs` tag.

Stages `Network`, `Router` and `SimpleStage` stages are defined here. Each stage must specify the class to be loaded upon execution. For the `Network` stage, that class is `bamboo.network.Network`. The `Router` stage also defines the gateway node to be contacted when joining the DHT (in this case, this node is the first node in the DHT, so it is its own gateway). The example stage `SimpleStage` also defines some custom parameters it fetches in its `init` method.

6.3.2.5 DHT/Data Layer

In staying with the staged, event driven nature of Bamboo the classical DHT operations of inserting, retrieving and removing data from the overlay can be supported with the inclusion of the `bamboo.dht.DHT` class as a stage in the configuration. The inclusion of this stage allows an application to participate in the DHT by storing data. The DHT stage is aided by two other stages, `DataManager` and `StoreManager`. The `DataManager` stage manages the data stored on the Bamboo node and allows a programmer to set parameters such as the data replication factor for data items stored in the DHT. The `StorageManager` stage interfaces between the DHT interfaces and the actual database platform for the storage of values in the DHT, allowing a programmer to set parameters such as the location to store the database file, or the maximum size of the node cache.

The database platform for Bamboo is in the form of the embeddable database called the Berkeley Database [78]. This is a lightweight storage system which stores data in key-value pairs, allowing for high performance and easy deployment. The Berkeley Database is highly configurable and needs no administrative effort to maintain, making it a good environment for developing peer-to-peer systems. In Bamboo, the database is imported as a simple .jar file and can be readily used.

6.3.2.6 Gateway layer

Another important stage in Bamboo is the `Gateway` stage which provides access to the DHT storage infrastructure. Nodes implementing this stage can issue commands using SUN RPC

over TCP/IP. DHT operations are supported by nodes running this stage through the classes `bamboo.dht.Get`, `bamboo.dht.Put` and `bamboo.dht.Remove`. By convention, a Bamboo node listening on port 3630 which runs the `Gateway` stage will listen for DHT requests on a different port, such as 3632. Example uses of the interfaces are given in the code below.

```
java bamboo.dht.Put <server_host> <server_port> <key> <value> <TTL> [secret]
java bamboo.dht.Get <server_host> <server_port> <key> [max_vals]
java bamboo.dht.Remove <server_host> <server_port> <key> <value> <TTL> [secret]
```

In the code above, TTL determines the expiry time for the record. The parameters secret and max_vals that are enclosed in square brackets are optional. The secret parameter associates a password with the insertion or retrieval of data records in the DHT. The max_vals parameter determines how many values associated with the key are to be returned to the requestor.

6.4 Development of plug-ins

A significant amount of time was spent understanding and experimenting with the DHT implementations described in the previous section. Both of the open source DHT projects have mailing lists which were instrumental in solving problems and understanding concepts that were unclear. The Bamboo DHT is shipped with helpful sample code and example configuration scripts that help to orientate the beginner. OpenChord has a manual which describes the interfaces and provides code snippets that show how the interfaces are used, as well as a custom shell that can be used to simulate network overlays on the local machine or a LAN. Equipped with a thorough understanding of the DHTs, it was then possible to create plug-ins for them.

In order to create a system that was consistent and orderly, the namspace overcord.overlays was created which contained each of the two types of DHT implementations. This resulted in the two namespaces overcord.overlays.openchord and overcord.overlays.bamboo. The plug-ins that were developed would be separate from the DHTs themselves, and formed the two namespaces overcord.plugins.openchordplugin and overcord.plugins.bambooplugin.

In creating the plug-ins, a Java interface class needed to be developed from which the plug-ins could inherit a common API. The interface that was used resembles the interface defined in section 5.1.4, in the proper Java syntax. Appendix B.1 shows the full Java source code for the interface.

The next two sections detail the process of development that was followed. The plug-ins that were developed were based on version 1.0.3 of OpenChord and the March 2006 snapshot of the Bamboo source code.

6.4.1 The OpenChord plug-in

The overcord.plugins.openchord namespace consists of three classes: `ChordPlugin`, `StringKey` and `URLCreator`. `ChordPlugin` is the main plug-in class that implements the generic interface. The `StringKey` class is based on the Java `String` class and provides the mandatory implementation of the `Key` interface which represents a unique data element in the DHT. The `getBytes` method of the `StringKey` class allows it to generate a byte array which is used to create a hash value of the data item using a hash function [73]. The `URLCreator` class helps create a custom URL for the OpenChord node.

6.4.1.1 Overlay management

When the OpenChord plug-in is started, it accepts as parameters the bootstrap node IP address or hostname as well as the bootstrap node's port number. If the node is the first node in the overlay, it is passed its own IP address and the port number 0, in which case it creates a local instance of a `Chord` class and invokes its `create` method to create a new overlay. If it is not the first node, it uses the bootstrap parameters to execute the `join` method of the `Chord` instance. The plug-in forces the node to listen for connections on port 49370 by default. When multiple OverCord nodes are started on a single host, the new node first looks for a free port greater than 49370 before it tries to start, in order to avoid port clashes. When a node must leave the overlay, the plug-in executes the `leave` method of the node.

6.4.1.2 DHT Operations

When the plug-in needs to insert a new record into the overlay, it passes two `String` objects representing the key-value pair. It creates a new `StringKey` object and inserts the resource, calling the node's `put` method. When a record, or records under the same key, are retrieved from the overlay, the plug-in calls the node's `get` method. The retrieval operation retrieves a single result or a set of results, which are stored in an array object in memory. A removal operation is performed by passing the key-value pair to the node's `remove` method.

6.4.2 The Bamboo plug-in

The overcord.plugins.bamboo namespace consists of five classes: `BambooPlugin`, `ConfigFileCreator`, `PutValue`, `GetValue` and `RemoveValue`. `BambooPlugin` is the main plug-in class that implements the generic interface. The `ConfigFileCreator` class allows the plug-in to dynamically create a Sandstorm configuration file that is formatted correctly to help the plug-in start. The last three classes assist the plug-in to perform DHT key insertion, retrieval and removal operations.

6.4.2.1 Overlay Management

The Bamboo plug-in obtains its bootstrap node address and port number and starts a Bamboo node using similar techniques to those of the OpenChord plug-in, except that the first Bamboo node in an overlay tries to run on the port number 49360. Node joining is however slightly more complicated in Bamboo than in OpenChord since a configuration file must be created before the node can create or join the overlay, as explained in section 6.3.2.4. The custom `ConfigFileCreator` class assists the node by dynamically creating a configuration file and formats it appropriately. By studying the *run-java* script that is shipped with the Bamboo source, it was understood that the class `bamboo.lss.DustDevil` is responsible for starting a Bamboo node, parsing the configuration file, loading the stages and connecting them. It was then possible to automate this process by automatically running this command in a Java `Process` object. Leaving the overlay involves terminating the process.

6.4.2.2 DHT Operations

The Bamboo plug-in has access to custom created `PutKey`, `GetValue` and `RemoveKey` classes. Each of these wrap the native DHT operations described in section 6.3.2.6. Since every Bamboo node started by the plug-in runs the Gateway stage, they can all send DHT commands using SUN RPC over TCP/IP.

6.5 Plug-in management

After the plug-ins for each DHT had been developed, it was necessary to create sample applications, one for each plug-in, that were able to test the success of the generic interface. The

applications that were developed were relatively simple applications that merely created a specific plug-in and passed commands to it. In the initial stages, it was tolerable to have separate applications for each plug-in, so as to test the success of the generic interface. Once this had been established, the separate applications needed to be merged into a single application, resulting in the development of the plug-in management layer. Section 5.1.6 provided a list of requirements for this layer. It is implemented in the overcord.pluginmanagement namespace, where the `PluginManager` class mainly coordinates the activities of this layer. Appendix A provides terminal outputs of experiments made using the `PluginManager` class as a standalone application.

The detection of resident plug-ins is made possible by the fact that plug-ins are located in a standard location within the OverCord system. As it can be observed from the description of the plug-ins in section 6.4, it is possible to have many non-plug-in, supporting classes within the same namespace. A naming convention is enforced such that every plug-in class must contain the string ”Plugin” in its name in order to be recognised by the `PluginManager`. The plug-in verification function is described in section 6.6.

The creation of the plug-ins is made possible through Java’s support for dynamic class loading (this has long been a feature of the Java programming language and will not be discussed here, but for more information on the topic, the reader is referred to [79]). When the plug-in manager has detected the plug-ins, it is able to present the list of plug-ins to the user, who can then select the desired plug-in and start the node. With the possibility of dynamically creating plug-ins, transparent control of the plug-in is achieved.

6.6 The discovery layer

Up until this stage, the OverCord system still required much administration and support, since the plug-in manager still relied on the human user to supply bootstrap details so it could start the DHT node. It was decided to attempt the use of multicast for the purpose of discovery, so that nodes that wished to discover new DHT nodes could send out multicast requests to the sip.mcast.net address, and receive responses in unicast containing the details needed to join the overlay. The reason multicast was chosen is because it is a sanctioned protocol for locating SIP entities such as registrars [9], is well supported in Java and is easy to implement. However, it is acknowledged that most network routers block such traffic, and thus nodes on opposite sides of a router would not be able to locate and participate in the same overlay when using only this

protocol for discovery.

For the purposes of discovery, the overlay plug-ins were extended to incorporate multicast and unicast capabilities. When a node has successfully created or joined an overlay, it spawns a thread that listens on the next free port after port number 49350 for incoming requests targeted at the sip.mcast.net address. When a new node generates a request, it sends out a UDP packet to the address and appends the message MCAST_BOOTSTRAP as a payload. This distinguishes a join request from any other possible use of the multicast address so the listening node can respond appropriately. When a node detects an incoming message, it parses it. If the MCAST_BOOTSTRAP message is found, the receiving node transmits a single unicast UDP packet containing a message which conforms to the pattern OVERLAYNAME PLUGINCLASS HOST PORT. In this way, the responding node informs the joining node what overlay it is in, the plug-in class the new node will need to join that overlay, what its hostname (or IP address) is and what port it is running on. The discovery module on the joining node collects all unicast responses it obtains and presents all unique overlay names to the plug-in manager. It is then that the plug-in manager can perform the plug-in verification by comparing the plug-in class token from the responses obtained through discovery with all plug-in classes that are detected on the local node. Plug-in creation can then be achieved using the usual procedures.

Section 8.2 discusses how dynamic DNS support was added to OverCord to allow interoperation with conventional SIP networks. When this technique was adopted into the system, it became clear that it could also be used for discovery as an alternative to the use of multicast. If an overlay has a fully qualified domain name (FQDN) and has an entry in DNS, when a node desires to join that overlay it can perform a DNS lookup operation to discover the location of a representative node in the overlay, and use that node to join.

If there is an overlay that a plug-in on the local node can create that was not detected by the discovery module, the node will be allowed to create the overlay as the first node in that overlay. It is important to note, however, that in a lossy network, or if the requesting node waits for too short a time, a new node may attempt to create an overlay that may already exist. Thus the waiting period in OverCord is set to a high value of ten seconds. If the user suspects that there has simply been a delay in the acknowledgment of the request, the user is able to issue new discovery requests.

Appendix A.2 shows an example of how the discovery module is used by the plug-in manager to discover existing overlays and bootstrap a desired overlay.

6.7 Interoperating peer-to-peer overlays

In OverCord, a peer-to-peer lookup serves as the prelude to any form of SIP-based communication, the most common purpose being to obtain the contact address of the user to which a SIP message is to be sent. It has been established that a resource such as a SIP URI must first be hashed, and the hash can used as the key for a lookup operation which propagates through the overlay until it reaches the node whose identifier is closest to the hashed string. Typically, the record sought exists in the same overlay in which the request originates and relates information pertaining to another user in that same overlay. However, if the local user wants to communicate with a buddy in another overlay, the corresponding record only exists in that users overlay, and therefore the lookup will fail to retrieve the correct information. Indeed, if no intervention is taken to distinguish between users in the local overlay and those in foreign overlays, the lookup operation still maps to a node in the local overlay (through the mapping of the key to a node ID), but that node will not possess the binding for the desired user since that binding exists elsewhere. If the lookup operation itself cannot be used to discover resources in other overlays, then support must be provided from elsewhere.

A naive approach to solving the problem would be to attempt to insert keys into foreign overlays so as to allow users who have buddies in other overlays to expose their contact addresses to them, by inserting and refreshing those bindings in the foreign overlay. This approach would not work for DHTs since the insertion operation relies on the native lookup function in order to determine the location of the node where the key is to be stored. Even if some non-DHT protocol were used to facilitate this sharing of keys across overlays, there is still a need to perform hash negotiations when the overlays concerned use different hash functions to produce keys. This is because there is no single hash function which is used across all types of DHTs, though SHA-1 has proven popular in a few protocols such as Chord and Pastry. Also, this solution does not scale well, since users would have to insert and refresh bindings in all overlays from which they would like to be reachable.

In networking environments, in order to allow traffic to flow in and out of one network into another, a gateway is deployed. To devices outside of a given network, the network is represented by its gateway. In turn, to devices inside the network, the rest of the world is only accessible through the gateway itself. The concept of a gateway could be applied to the problem of overlay interoperation whereby a gateway is used to represent an overlay to all other overlays that the local overlay would like to communicate with.

In their paper, Singh *et al* [53] present a design for interoperation between overlays that uses

gateways. The design relies on the existence of super nodes in local overlays, as well as a super overlay (global overlay) of super nodes, which assists the local overlays to interoperate. In each overlay, nodes keep two configuration settings, domain and next-level. The domain is the name of the domain for the node and the next-level is the next domain in the hierarchy, such as the global overlay. Bootstrap nodes in the global overlay are configured with these parameters as empty. When a node joins a domain, it assigns the domain setting to the name of the domain it is participating in and the next-level setting to the next domain in the hierarchy that it is configured with. If the node is a gateway, it registers its location in DNS under the name of the domain. If its next-level setting is not empty, it performs a DNS lookup for the next-level domain and sends a SIP REGISTER message to the resolved host. The next-level domain will then store location records for the local overlay using the details supplied by the gateway. Ordinary nodes in local overlays simply send REGISTER requests to the gateways registered in DNS when joining the overlay. This design is illustrated in Figure 6.3.

When a node in the local overlay wishes to setup a session with a node in the same overlay, it uses the local location service to locate that node and send an appropriate SIP message directly. If a node wishes to create a session with a node that is not in the same overlay, it discovers that its domain setting is not the same as the domain indicated in the destination user's SIP AOR and it sends the request to its local gateway node. The local gateway node then proceeds to proxy the request to the domain indicated in its next-level setting, which may be the global overlay itself. Since all overlays have registrations in the global overlay, ultimately the gateway node that resides in the overlay the user would like to contact will be available, and the request can be sent to that gateway. The gateway in the foreign overlay can then proxy the request to the desired user, and communication can be established.

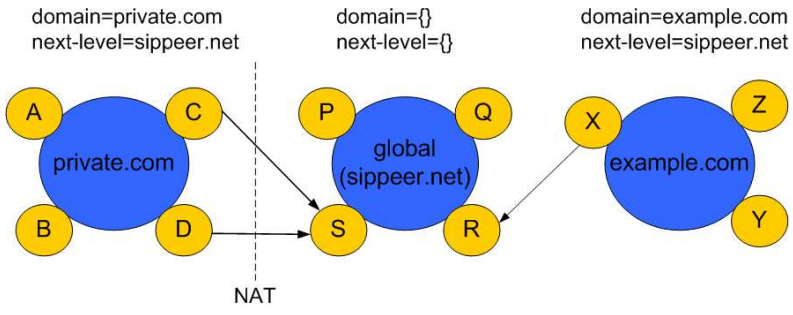

Figure 6.3: A solution for interoperation of overlays. Adapted from: [53].

Gateways are common entities in networking environments and do have some benefits. In their paper [80], Garces-Erice *et al* argue that hierarchical peer-to-peer systems are advantageous because they reduce the average number of lookups and can reduce the lookup latency when peers in the same group are topologically close. When investigating the possibility of using OverCord for overlay interoperation, the use of gateways was considered, but was not used as a solution to the problem of interoperation for two main reasons. Firstly, while the use of gateways does provide interconnection, it does not take full advantage of the properties of a decentralised system since the gateway represents a single point of failure. If the gateway crashes, external queries can no longer be serviced. Secondly, if there is a large number of local nodes that have friends in foreign overlays, the gateway becomes the target of a large amount of query traffic which can put much strain on it. It becomes necessary to select nodes that have long online time, sufficient bandwidth and processing power as gateways. This means that the gateway must generally be a higher capacity node that can handle this type of strain, which can be problematic for interconnecting overlays consisting of low end devices, such as in mobile ad-hoc networks where there may not necessarily be a ”reliable” peer. Ultimately, it is a strict requirement that does not always fit all peer-to-peer models well. It is possible to deploy a large number of redundant gateways to combat this problem, but this does not scale well either, and may require external administrative effort, which defeats the purpose of a dynamic, self sustaining system.

OverCord manages to solve the problem of overlay interoperation, while maintaining a flat, non-hierarchical overlay topology. The solution does not depend on any centralised protocols in order to achieve this, including DNS. It does this by leveraging the DHT protocols by using other peer-to-peer mechanisms (such as those used for discovery and lookup by the unstructured peer-to-peer protocols described in section 3.2.3), to help provide a discovery service for locating foreign overlays, with the purpose of executing the lookup in that overlay. Obviously, the ability to perform the lookup will depend on the node possessing the appropriate plug-in for use in that overlay. The protocols that assist the DHTs, sit at the discovery layer in the OverCord node design given in Figure 5.1.

In order to differentiate between keys that are resolvable within the local overlay and those that are not, before passing the key to the DHT lookup infrastructure, the plug-in management layer of the node examines the domain portion of the key, and compares it with the FQDN of the overlay in which it belongs. If the FQDN matches the domain portion of the key, the key is hashed and resolved locally. If not, the key is preserved as is, and is passed to the discovery module for query in a foreign overlay. The procedure from this point is similar to that which is followed for bootstrapping an overlay. What differentiates this scenario with the bootstrapping scenario

is that, rather than leaving the overlay in which the node is currently participating in to join a foreign one, the node creates a temporary plug-in that is legal for the foreign overlay, and uses this plug-in to join that overlay. The other differentiating factor is that the life-time of the plug-in is very short, as its purpose immediately after it has joined is to execute a lookup and return the appropriate record to the application. The temporary plug-in is subsequently terminated, and the node can use the obtained binding to setup a new SIP session. Appendix A.3 shows examples of resource retrievals that occur between nodes in heterogeneous overlays.

6.8 Summary

This chapter has described an example implementation of the OverCord system that was outlined in the previous chapter. This implementation is only one of several possible realisations of the OverCord model, since OverCord is not bound to any specific platform or programming language. The implementation is limited by the discovery problem, since there is limited support for multicast across routers. It is anticipated that developers would implemented their own, more effective protocols to improve this. Subsequent implementations may also attempt to secure the data in the DHT to avoid attacks from malicious nodes. It also remains to be seen how easy it is to implement plug-ins for unstructured protocols given that this implementation benefited from the fact that structured protocols have many similarities, and it is easy to conform them to a common API. The next chapter details the next step towards proving OverCord's suitability for P2P SIP, where its incorporation into an open source SIP UA is documented.

Chapter 7

Incorporating OverCord into a SIP User Agent

> *It is no paradox to say that in our most theoretical moods we may be nearest to our most practical applications*
>
> - Alfred North Whitehead

The previous chapter has described in detail the development of a peer-to-peer system based on the OverCord model. Our attention now focuses on the incorporation of OverCord into a currently existing SIP user agent and how such a modified user agent behaves. To demonstrate this, an open source SIP user agent that was developed by researchers at the National Institute of Science and Technology (NIST) called the JAIN SIP Applet Phone was obtained, and an incorporation of OverCord was performed on it. This chapter details how the incorporation was achieved and what modifications were necessary. It also shows how various forms of multimedia communications were made possible using this modified user agent, thus validating the effectiveness of OverCord as an appropriate peer-to-peer system for SIP. The chapter also shows how the OverCord implementation supports interoperation between user agents in heterogeneous overlays.

7.1 Implications for conventional SIP user agents

Figure 7.1 shows the configuration frame of a popular voice and video phone called WengoPhone [81]. Many user agents such as WengoPhone participate in closed networks which usually require the user to create an account via a website. Alternatively, advanced users can configure the client to communicate with any other SIP service of their choice. Figure 7.1 shows the SIP account details of a user with extension number 7521 who is served by a SIP server reachable on the hostname sip.ict.ru.ac.za and port 5060.

Figure 7.1: WengoPhone client configuration frame.

Figure 7.1 highlights two important features of any SIP user agent configuration. Firstly, there is a user identity bound to the user agent which identifies the user to the network. When a user joins a SIP network, their user agent uses the credentials in the configuration to create a SIP REGISTER request which is sent to the indicated SIP server for authentication. Secondly, there is the networking aspect which relates to the network identity and location of the SIP proxy or registrar. The hostname or IP address of the host serving the SIP network informs the user agent as to the location to which to send registration requests, as well as all other SIP requests that must be proxied to their destinations. It is this property which is most interesting in the case of P2P SIP. This is so because there is no fixed point to refer to when a node needs a message to be proxied, rather the proxy method is distributed among all overlay peers. Lastly, users may want to be involved in many overlays, thus the user agent must be able to discover multiple overlays, and possibly allow them to join and operate in multiple overlays simultaneously.

These ideas were examined when planning for the incorporation of the OverCord implementation

into a currently existing SIP user agent. To ease the progression into this stage, since the plug-ins described in section 6.4 were developed in the Java programming language, software written in that language was sought. The next three sections describe the status of Java in IP telephony, define the different platform options that Java programmers interested in SIP telephony have to choose from and justify the selection of a particular SIP user agent.

7.2 Java in IP telephony

The National Institute of Science and Technology (NIST) is a non-regulatory body within the USA's Department of Commerce that conducts research in various topics in science and engineering through several research programs. One of the sub-projects in the NIST laboratories program is the IP Telephony Project [82], which deals exclusively with the use of the SIP protocol for Internet telephony and other next generation services. Historically, the C and C++ programming languages have had a monopoly in the telecommunications scene, as evidenced by the popularity of a vast number of SIP stacks such as SOFIA-SIP [83], oSIP [84] and SIPX [85] as well as SIP Application Servers and Service Creation Platforms such as SER [86], CINEMA [87] and VOCAL [88], which are based on those languages. Researchers at NIST however, joined a group of other data communications and telecommunications companies in an effort to bring the Java programming language to the fore in this area.

The Java Community Process (JCP), is a consensus building process which allows interested parties to participate in the extension and development of the Java language [89] through the definition of specifications called Java Specification Requests (JSRs). This process was followed and resulted in the definition of four JSRs dealing with SIP [90]. These are:

JAIN SIP A general purpose stateless, transaction and dialog stateful interface to SIP, for designing user agents and proxies [91].

SIP Servlet API An API for developing application servers which can have SIP and HTTP components for the provision of converged services [92].

JAIN SIP Lite A lightweight specification designed for either the J2SE or J2ME platforms for the development of stateful user agents [93].

SIP API for J2ME A SIP interface specifically for use by lower end devices and is based on the Connected Limited Device Configuration (CLDC) framework within the J2ME platform [94].

The description of the different Java specifications for interfacing with the SIP protocol highlights the fact that user agents are of different levels of complexity. SIP Lite and the SIP API for J2ME are the least complicated and least powerful. The JAIN and Servlet APIs are richer and can be used to develop more powerful and sophisticated user agents or SIP servers. P2P SIP nodes must be intelligent endpoints, capable of performing much processing to fulfill the traditional server roles, which means that the JAIN and Servlet platforms would be the most appropriate to work with for designing these types of SIP entities. The research chose the JAIN SIP platform since it allows designers to develop user agents and manipulate SIP signaling whereas the Servlet applications are service oriented and are in any case built over JAIN SIP itself. The next section describes the architecture of the JAIN platform for SIP.

7.3 Designing applications with JAIN

NIST laboratories currently provides download access to the JAIN version 1.2 SIP library to the open source Java community [95]. The API is fully documented in the reference implementation documentation [96] which defines the structure of JAIN SIP, and is also illustrated in Figure 7.2.

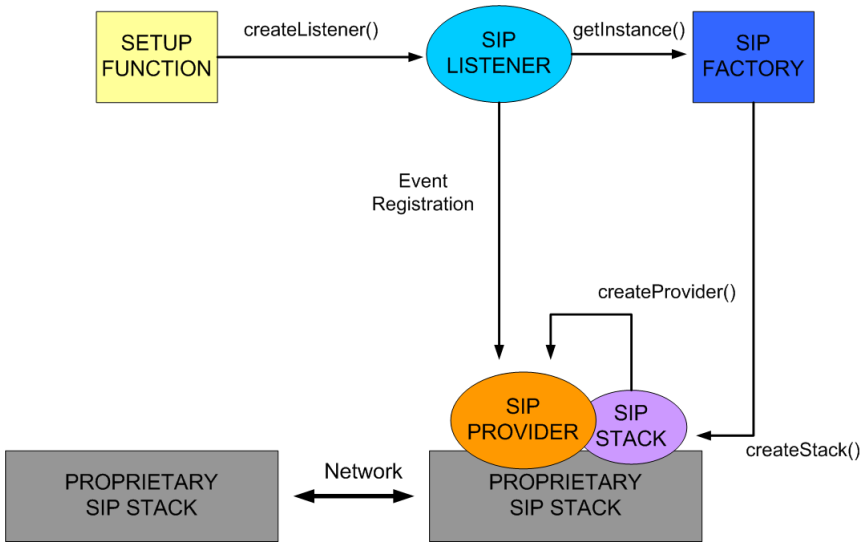

Figure 7.2: Architecture of JAIN SIP. Source: [97].

Figure 7.2 shows that a SIP user agent is started by the creation of a `SipStack` object. The stack object defines the methods used to represent a proprietary SIP protocol stack. Values can be

passed to the `SipStack` that define properties such as the device IP address, the stack name and outbound proxy address and proxy port. The `SipStack` is only accessed through one or more `SipProvider` objects. The `SipStack` also registers one or more `ListeningPoint` objects to a `SipProvider` (one listener per transport protocol, such as UDP and TCP) which define the local addresses at which the application can receive messages. Again, many `SipProvider` objects can be associated with a single `SipListener` object which acts as a proxy, passing all SIP requests and responses as events to the higher level application.

7.4 Choosing a user agent

As an offshoot of the IP telephony project, researchers at NIST Laboratories also provide download access to the source code of several SIP user agents and servers [82] that they have developed. They are listed below:

JAIN SIP PRESENCE PROXY A proxy server with presence support, an instant messaging client and a SIP gateway with RTP bridging between two networks.

JAIN SIP APPLET PHONE A user agent with presence, instant messaging and audio support (either with realtime RTP or voice messaging).

JAIN SIP APP SERVER Allows users to upload and control services written using the JAIN SIP API and control these services using a web frontend.

JAIN SIP 3PCC A third party call controller which can setup and manage sessions between two SIP endpoints.

An investigation was conducted in order to determine which of the applications listed above to use for the insertion of the OverCord implementation. The final decision would be based on three factors: the foreseen ease of performing the integration, the complexity of the code and the ability of the application to provide support for multimedia services. Thus, it was easy to rule out the third party call controller and the SIP application server since they mainly concern themselves with signalling and do not cater for media. The SIP presence proxy bundles a server and a client which can be run with each other. It was found to be less flexible in providing media services (the client only supports instant messaging and the server only handles presence), and so the applet phone was selected to prototype a P2P SIP node using OverCord.

The next section details how the user agent was modified to support OverCord. The prototype designed here uses the February 2007 JAIN SIP Applet Phone (JSAP) source code snapshot, and JAIN version 1.2 libraries.

7.5 Modifying the JAIN SIP applet phone

The JSAP application consists of a large number of source files and as such, was released with the necessary support to allow for the use of the Java build tool called Ant [98] for building and launching the application. JSAP can be launched in a web browser as well. It has multimedia capabilities provided by Java Media Framework (JMF) libraries, and uses an implementation of the XML data format for presence described in [99].

Several parts of the JSAP source code were changed during the process of integrating OverCord. Some of these changes were inconsequential with regard to the integration process, and as such are not discussed here but will be submitted at a later date to the project as patches. However, there were three main aspects of user agent design that were important. Firstly, the configuration aspect, since in a peer-to-peer environment, user agents cannot use static configurations. Secondly, registration takes on a different form since there is no central registrar. Thirdly, the provision of services is different in OverCord since the services are not centralised but distributed across all nodes in the overlay. The next three sections describe how JSAP currently handles these issues and how the application was changed to be more suitable for a decentralised environment.

7.5.1 Configuration

The main class in JSAP is the `NistMessenger` class. This class creates an instance of the configuration class (appropriately named `Configuration`) which is responsible for collecting information necessary for the client to operate on the network. This includes the IP addresses and listening port numbers of both the client machine and the SIP server serving the client, as well as the signalling and media transport protocols (such as UDP or TCP) the client and server will use. These values are needed for the correct instantiation of the SIP stack interface (see section 7.3).

In the original design, users would have to set these parameters manually in the source code. During the investigation, the configuration class was modified in order to detect the host's IP address automatically when the application is started. If the host has more than one network

Stack Property	Description
javax.sip.IP_ADDRESS	Sets the IP address of the SIP stack.
javax.sip.STACK_NAME	Sets a user friendly name for the underlying stack implementation.
javax.sip.OUTBOUND_PROXY	Sets the outbound proxy of the SIP stack.
javax.sip.ROUTER_PATH	Sets the fully qualified classpath to the application supplied Router object that determines how to route messages before a dialog is established.
javax.sip.EXTENSION_METHODS	Informs the underlying implementation of supported extension methods that create new dialogs.
javax.sip.RETRANSMISSION_FILTER	A helper function for user agents that enables the SIP stack to handle retransmission of ACK requests, 1XX and 2XX responses to INVITE transactions for the application.

Table 7.1: JAIN SIP stack properties. Derived from [97].

interface or there are several IP addresses bound to the client's network interface, the application detects this and select one of these at random (the user would be allowed to select a different address to use later if desired). By default, the application listens on port 5060, but if there is a clash, it searches for the next free port greater than 5060. The configuration was also modified to specify the outbound proxy as the bootstrap node that was used to join the overlay.

Once configuration information has been set, the `NistMessenger` class creates an instance of the `MessengerManager` class, which manages all SIP messaging that the client performs. This object creates an instance of the `MessageListener` class which implements the `SipListener` interface described in section 7.3. The `MessageListener` class creates the SIP stack object and uses this to register a `ListeningPoint` to a `SipProvider`. In creating the stack object, the listener class passes the necessary configuration information to it.

In addition to IP addresses and port numbers, there are a number of stack properties that need to be redefined for the purposes of peer-to-peer. The properties of the stack interface are summarised in Table 7.1.

Of those listed in Table 7.1 only the outbound proxy and router path properties of the SIP stack are directly relevant when working in a peer-to-peer paradigm. The settings of the outbound proxy have already been explained. The router settings are important if the node has run out of options and needs to send out a SIP request. By default, JSAP simply passes the outbound

proxy settings to the router class, though other more complex routing procedures are possible. In the modified version, JSAP still uses the outbound proxy settings, but as explained, these are those of the bootstrap node that was used to join the overlay. The advantage here is that the bootstrap node is not static, but changes often since the details of the bootstrap node are obtained by discovery protocols. This allows the node to self-configure itself in a dynamic environment that changes frequently.

7.5.2 Registration

Once the stack has been appropriately instantiated, the process of registration can occur. Section 2.2 defines registration as the process by which a SIP client's AOR is associated with one or more contact addresses. In the modified JSAP application, a new XML file called networks.xml was created which is used to map an overlay to a username. When OverCord has detected overlays, a matching is made between the overlay and the appropriate username to use for that overlay which is derived from the file. Thus when a user selects and joins the overlay, the plug-in is started and it can use the indicated username to create a SIP AOR mapping with a contact address. It achieves this through a DHT put operation which inserts this record into the location service.

This intervention in JSAP's design required a modification of the `MessengerManager` class. This class is responsible for initiating the registration and typically obtains the outbound proxy details and uses them to construct and send a SIP REGISTER message to the proxy. In the new design, the `MessengerManager` class was modified to send a request to OverCord's plug-in manager to create a plug-in that can operate in the overlay the user has selected to join. Afterward, it sends a second request to the plug-in manager, supplying it with a SIP AOR and contact address, requesting it to perform a resource insertion into the overlay.

7.5.3 Services

Currently, when a user creates a session using JSAP, the `MessengerManager` calls a method such as `sendInvite` which creates a SIP INVITE message and formats it appropriately. It then creates a client transaction and passes control to the `MessageListener` which uses a `SipProvider` to send the message to the appropriate destination. This general model is preserved in the modified user agent but with one difference: it must first obtain the location of the target user from the distributed location service. For this, it uses the plug-in data retrieval

method, after which, it can then send the message to the proper destination. This process is illustrated in Figure 7.3 which shows how a lookup is propagated through the network and a SIP message is subsequently sent to the intended target user.

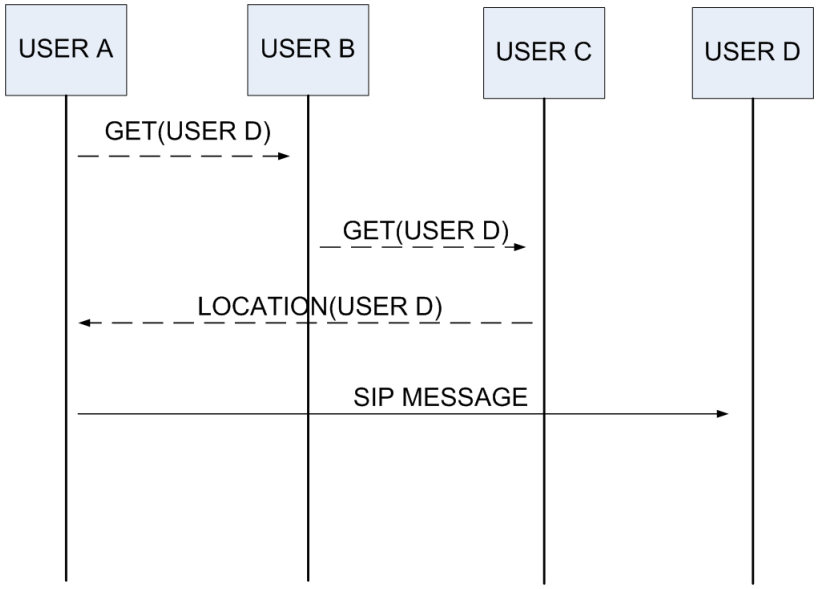

Figure 7.3: Message relay in OverCord.

Voice, instant messaging and presence services are enabled in a peer-to-peer environment with only slight differences between them. The `MessengerManager` has a `sendMessage` method to send an instant message, a `sendInvite` method to request an audio session and `sendSubscribe` and `sendNotify` methods to request and return presence information.

In any case the user agent must first obtain the location details of where the destination host is available. These records are present in the location database distributed among all the nodes. The user agent must therefore then use the active plug-in to perform this query and obtain the location. The `MessengerManager` object obtains a representation of this binding through the plug-in framework, and then uses the binding to reset the next hop destination. This is necessary since in the absence of a static proxy, the sending user agent must continually redirect this property to the intended user. The user agent does this by resetting the `Configuration` object properties outboundProxy and proxyPort accordingly. The user agent's `SipProvider` creates a new client transaction for the request and sends the message.

An extension to the SIP messaging system that the user agent introduces is the use of an overlay

name header. This header is appended to each outgoing message that is sent, where the value associated with the header name corresponds with the user's local overlay name which is obtained by parsing the user's SIP AOR. The advantage of this is that the receiving endpoint is able to parse this header and make an association between a live node and an overlay which can be inserted into a bootstrap list in case the endpoint should want to join this overlay at another time. Over time, each user agent could build up a list of potential bootstrap hosts that can be attempted in addition to the usual discovery protocols.

When a message reaches the destination user agent, the `MessageListener` object detects the incoming request event and must determine how the message will be processed. The listener has `processRequest` and `processResponse` methods which pass control to the instance of the `MessageProcessor` class to handle the specific actions associated with message handling, such as `processInvite` and `processMessage` for processing INVITE and MESSAGE requests respectively. The receiving user agent then needs to parse the SIP message that it has received to obtain the identity of the sending user and obtain the corresponding contact address or addresses associated with that user. Since this user agent may have a different next hop configuration from a previous communication session with another user, it reconfigures them appropriately so that the response can be sent to the initiator of the request. The destination user agent creates a new server transaction for this new request and responds using the processor object.

7.5.3.1 Audio Capabilities

There is a special need for discovering the capabilities of other users in the overlay without ringing the other party through the use of the SIP method OPTIONS as described in [100]. One use of this method is for establishing audio sessions. Some canonical SIP networks allow user agents with incompatible media codecs to communicate by providing media transcoding on their behalf. In the user agent, while some nodes may provide this service on the behalf of the overlay, user agents may send OPTIONS messages to target parties to discover their abilities before they transmit a media invitation. The user will be notified if there is a problem with media compatibility.

7.5.3.2 Presence Capabilities

Table 2.1 defined the SIP method PUBLISH as a request method that is used by SIP user agents to advertise presence status to a presence agent such as a SIP presence server. Other SIP devices

known as watchers subscribe to this status using the SUBSCRIBE method and the presence agent returns a NOTIFY to the watcher as a notification. In P2P SIP, there is no dedicated presence server to receive PUBLISH methods and as such only the SUBSCRIBE and NOTIFY methods are used in a peer-to-peer fashion. In the user agent, when the presence status of a user changes, the list of watchers with valid subscriptions to presence state is collected and NOTIFY messages are sent to each in sequence. Figure 7.4 shows the buddy list of a user in the domain bamboo.dyndns.org. The figure shows that this user is able to obtain presence information about users in their own overlay and in other overlays.

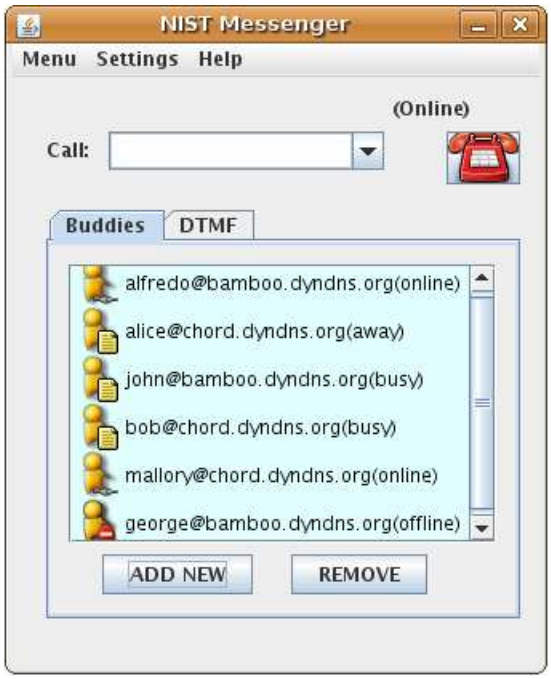

Figure 7.4: Presence support across heterogeneous overlays with the JAIN Applet.

7.5.3.3 Instant Messaging

Table 2.1 defined the SIP method MESSAGE which is used to convey instant messaging information. These messages can be exchanged between each user directly. Figure 7.5 shows an instant messaging session between users in different overlays.

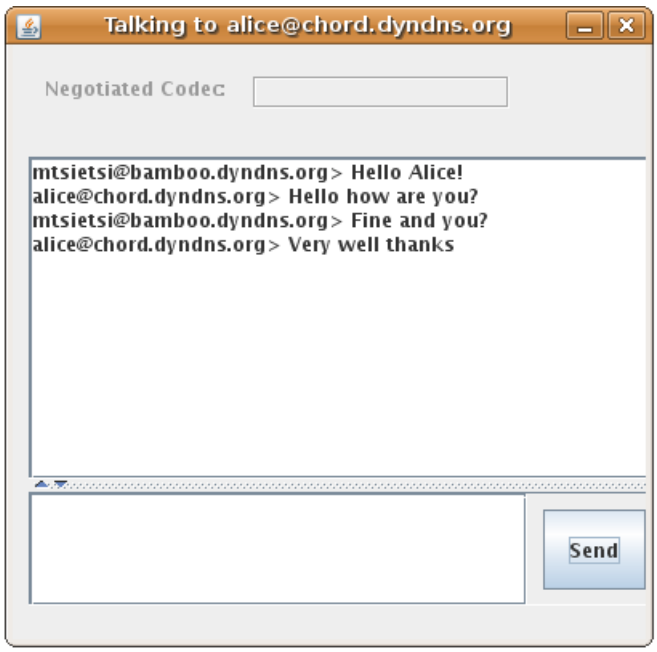

Figure 7.5: Instant messing session across heterogeneous overlays with the JAIN Applet.

7.6 Summary

Currently, there is little support for peer-to-peer communication in most conventional SIP user agents. It is likely that when P2P SIP becomes a standard and interest in the technology grows, developers will be interested in modifying existing SIP user agents to embed peer-to-peer functionality. This chapter has described such a modification which specifically shows how currently existing Java user agents based on the JAIN SIP stack can be re-worked with relative ease to become peer-to-peer enabled. The next chapter takes a step forward and addresses the interoperation of peer-to-peer SIP user agents with user agents in conventional SIP networks.

Chapter 8

Interoperation with Conventional SIP Networks

Only connect!

- E.M. Forster, Howards End

This thesis has demonstrated that OverCord can interoperate between overlays based on heterogeneous DHTs. However, it is very likely that users in these overlays will not only have contacts in other decentralised overlays, but also in conventional SIP networks. It would therefore be beneficial to support interoperation between peer-to-peer overlays and traditional, centralised systems. The crux of any form of interoperation in SIP systems is the ability to calculate the next hop for routing messages from one network to the other. The challenge with peer-to-peer systems in this regard is that due to high rates of churn, the next hop may change frequently, therefore a mechanism must be put into place that takes this dynamic property of decentralised systems into account. Dynamic DNS updates were identified as a possible solution. This chapter describes how this solution can be used to interoperate peer-to-peer overlays with centralised systems and how OverCord was extended to support this new feature.

8.1 Interoperating with conventional SIP

If there has been a shortage of debate and literature that addresses interoperation between heterogeneous peer-to-peer overlays, there is even less concerning explicit solutions that describe how decentralised overlays can interoperate with conventional SIP networks. Popular P2P SIP protocol proposals such as those described in section 4.2.1 do not address this issue.

An early Internet Draft by Shim *et al* [101] was one of the first to propose a design for interoperating P2P SIP and conventional SIP systems, albeit from the perspective of a single composite domain consisting of both peer-to-peer and non peer-to-peer entities. The authors were aware of the assymetry that exists between providing support for nodes in a decentralised network to interoperate with nodes in a centralised network, and vice versa. To understand the reason for this, consider two users: Bob who owns a user agent in a peer-to-peer overlay, and Alice who uses a centralised service. If Bob wishes to send a message to Alice (say to alice@example.com), Bob simply needs to create a SIP message in the usual way, and use the procedures defined in [18] to send the message. Since Alice's user agent simply needs to return subsequent messages to Bob's IP address, there usually is not problem with these users communicating. The opposite case is somwhat more complicated. The solutions that have been proposed are provided in this section.

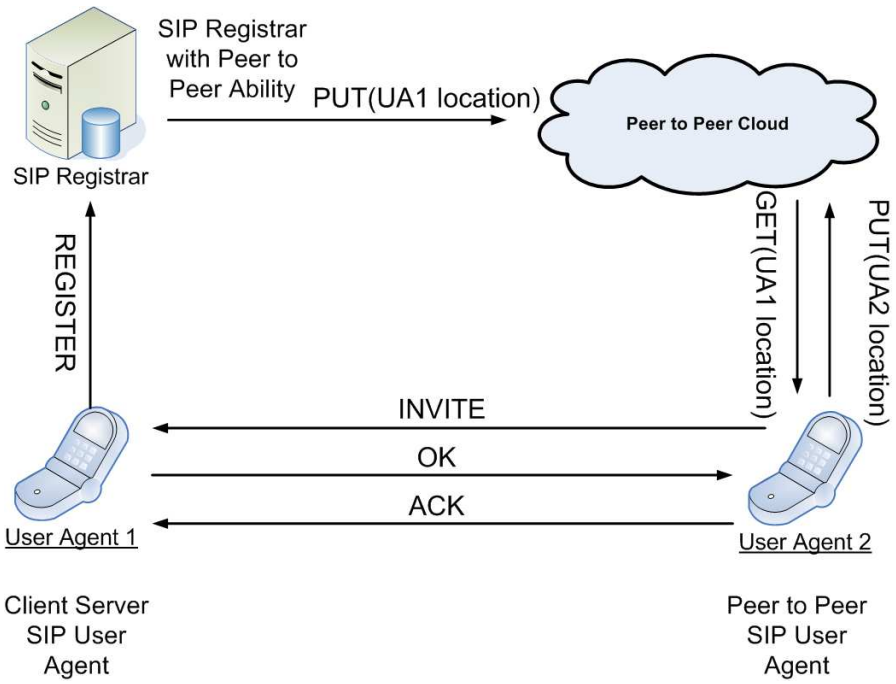

Figure 8.1: A call from a peer-to-peer overlay to client-server system. Adapted from [101].

Figure 8.1 shows how a peer-to-peer user agent creates a communication session with a client-server user agent. Firstly, the peer-to-peer user agent inserts a location record into the distributed location service. Similarly, the client-server user agent constructs a REGISTER message which it sends to a SIP registrar server, which inserts the registration into a centralised location service. This design depends on an improvised registrar server which co-locates a peer-to-peer node that enables it to insert registration records in the decentralised location service for the user agents that are not based on peer-to-peer protocols. In the figure above, when user agent 2 wishes to create a session with user agent 1, it retrieves the binding that was registered on the behalf of user agent 1 by the registrar, and is then able to construct a SIP message and send it directly to user agent 1.

The second part of the design addresses how client-server user agents communicate with peer-to-peer user agents, which is illustrated in Figure 8.2. In order to support this, the design defines a SIP proxy which is similar to the SIP registrar in that it also co-locates a peer-to-peer node in order to perform the proxy service on behalf of client-server user agents. The proxy performs the lookup in the peer-to-peer overlay and forwards the request to the relevant node to help establish the session.

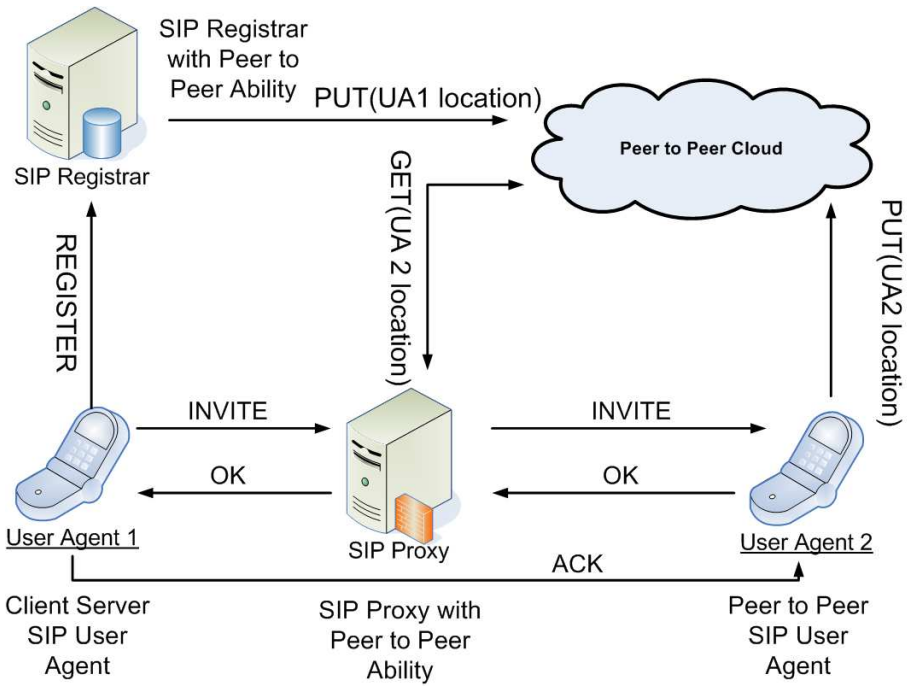

Figure 8.2: A call from a client-server system to a peer-to-peer overlay. Adapted from [101]

The design is interesting in that it recognises the possibility that in the future, SIP networks may consist of both conventional and peer-to-peer SIP user agents, and there will be a demand for the network to support communication between them in a seamless fashion. There are, however, some problems with the design. Firstly, while within a single domain interoperation is possible, the design is not easily extensible for interoperating between user agents in separate domains, that do not expose each other's location services to one other. Even if the design were to be extended to support this, sharing bindings with all possible domains in the world is not a practical way of providing interoperation. Also, the design is problematic because it relies on modified SIP registrars and proxies, which do not currently exist. The viability of the design thus depends on development in the area of open source SIP servers, or market pressure on manufacturers to patch their products to embed peer-to-peer logic. A better design would not demand any change in the current infrastructure, but would be able to work around the differences.

In light of the limitations in this design, our aim was to keep the location services of decentralised and centralised networks separate, and require no modification of current SIP infrastructure. This is more difficult to provide since the design that is described above hides the complexity of interoperation in that it uses a common location service from which SIP servers can obtain

bindings for peer-to-peer user agents. If this convenience is removed, an alternative method is needed.

A second solution for interworking P2P SIP overlays with conventional SIP systems is provided by Marocco *et al* in [102]. The proposed solution is illustrated in Figure 8.3. In this design, special UAs in the peer-to-peer overlay (namely a relay agent peer and a P2P SIP proxy) help provide interoperation between the two systems. The P2P SIP overlay name must be a fully qualified domain name (FQDN) where the P2P SIP proxy peer has a binding in DNS. The proxy peer is responsible for performing the proxy function between the peer-to-peer overlay and the outside world, and thus becomes a connection point into the overlay. The relay agent peer uses protocols such as Traversal Using Relay NAT (TURN) [103] to provide relayed transport addresses for allowing media streaming between UAs without direct connectivity. From the perspective of the outside world, the peer-to-peer overlay is simply another conventional SIP system.

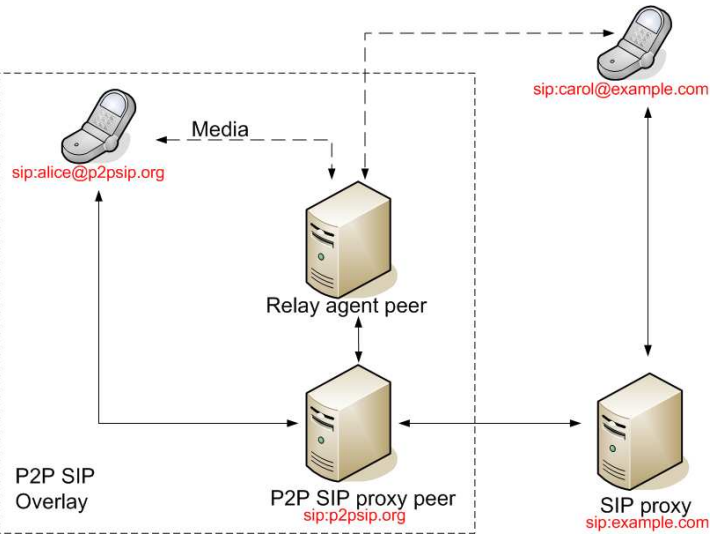

Figure 8.3: A solution for interoperation of overlays using P2P SIP proxy. Adapted from [102] .

As shown in Figure 8.3, when Carol (who is in a centralised domain) wants to contact Alice (who is in a peer-to-peer overlay), she forwards an INVITE to her SIP proxy. The proxy uses mechanisms described in section 2.3 to locate the P2P SIP proxy peer, which then uses its location service to locate Alice and forward the INVITE to her. When Alice receives the INVITE

request, she queries the overlay for one or more relay peers and initialises one for preparing an ICE candidate. Then the session can be established between the two users with media being relayed through the relay peer.

The design that Marocco *et al* proposes should be able to provide a solution to the interoperation problem. However, it relies heavily on DNS to provide this support. Generally, peer-to-peer networks are very dynamic systems, characterised by high rates of churn. Thus the identity of the P2P SIP proxy would need to change quite frequently, as nodes arrive and leave the overlay, since the proxy function would have to shift from node to node, requiring regular updates to the DNS resource records. This property is at odds with DNS, which assumes a static relationship between domain names and the hosts they map to. Thus conventional DNS is more suited to environments where changes to the resource records are few and far between.

The challenge of being able to apply rapid changes of resource records is not unique to peer-to-peer environments. Typically, a user who subscribes to an Internet Service Provider (ISP) for Internet connectivity, finds that the IP address of their machine changes regularly. This is due to the fact that in order to access the Internet, a user's machine must first obtain a lease on an IP address through Dynamic Host Control Protocol (DHCP) [104] procedures from their ISP. This lease persists for a limited amount of time, typically a few hours, after which the lease expires and the machine must generate a new request. The motivation for providing varying IP addresses is that IPv4 addresses are in short supply, which means that the provider must recycle a small set of IP addresses among all their clients. Traditionally, the usage of the Internet has been client based, meaning that users would use the Internet simply to retrieve content from servers. That model is changing into one whereby users are providing content from their own machines to the general public, such as web blogs. In order for other users to access that service, they must keep track of the actual IP address of the server even as it changes, since the ISPs do not create DNS records through which a client's IP address can be associated with a human readable domain name.

In recognition of the need for support for dynamic updates to resource records, a new operation code was defined called UPDATE, which can be used to add and delete DNS resource records [105]. This method allows these operations to be made in the absence of manual administration, and typically associates a DNS resource record with a very low TTL so that clients performing name resolutions would not cache these records for long. The use of dynamic updates to DNS records is often referred to as Dynamic DNS (DDNS). This technique not only suits DHCP environments where clients must update records in realtime, but it also suits P2P SIP environments where the identity of the P2P SIP proxy may change frequently.

The next section describes how the JSAP application was further modified to support DDNS so that it could be a candidate for a P2P SIP proxy peer in a peer-to-peer overlay, and how interoperation with a conventional SIP system was achieved.

8.2 Adding dynamic DNS support to OverCord

8.2.1 DDNS clients

A host that needs to perform dynamic updates of DNS records often uses an external DDNS client. DDNS clients differ in complexity and configurability, depending on the policies of the DDNS provider that renders the service. However, there are many providers that use a common method for making the changes, which allows one client to be used across several DDNS services. One such client is a Perl program called ddclient [106]. The ddclient program uses a simple configuration script which defines several variables whose values a user can assign such as the name of the DDNS provider, the DDNS domain name, the updated IP address and the relevant user credentials for the DDNS account. The client can be run in the foreground or as a daemon and can be invoked periodically through a scheduled computer task such as a UNIX cron job.

The ddclient software can be used with, among many others, the DynDNS.com [107] service. Through their web site, it is possible to create up to five free hostnames on several domains. Since OverCord currently embeds two DHTs, each of which can be used to create an overlay, two such hostnames were created, which could later be bound to each overlay. The hostnames for the OpenChord and Bamboo overlays were chord.dyndns.org and bamboo.dyndns.org respectively. An example of the resource record associated with the chord.dyndns.org domain name is given below, which was extracted from the output of a packet sniffer after performing an nslookup command for the hostname. Note that the TTL is one minute, which is the minimum setting for the provider.

```
chord.dyndns.org: type A, class IN, addr 146.231.123.55
            Name: chord.dyndns.org
            Type: A (Host address)
            Class: IN (0x0001)
            Time to live: 1 minute
            Data length: 4
            Addr: 146.231.123.55
```

This new feature was found to have implications for the DHT plug-ins themselves in that, the plug-ins could be coded to be associated with a DDNS hostname by the plug-in developer, by extending the generic interface implemented by the plug-in, to present this hostname to the application through a new method called `getDomain`. This is not a crucial feature of DHTs, being only necessary for interoperating with client-server systems, and OverCord can still function without it. If DDNS is not used, the plug-in developer can assign the name to `null` or alternatively, the empty string.

8.2.2 A first attempt - Importing a DDNS client

As a first attempt at incorporating DDNS capabilities into OverCord, the configuration script of the ddclient application was customised in the manner given in the sample below. It was discovered that by running the ddclient client, the configuration parameters are collected together, and the query is sent to the provider's server as an HyperText Transfer Protocol (HTTP) GET request along a secure SSL channel.

```
use=ip,   ip=146.231.123.55       # IP address
login=mtsietsi                    # default login
password=dyndnspass               # default password
##
## dyndns.org dynamic addresses
##
## (supports variables: wildcard,mx,backupmx)
##
server=members.dyndns.org,             \
protocol=dyndns2                       \
chord.dyndns.org,bamboo.dyndns.org
```

It is possible to run the ddclient from the OverCord application when the node starts. After it has obtained its IP address, a new class called `DynDNSHost` was created, the purpose of which is to query the availability of the local DDNS host. If the binding is found to be stale, the local instance of this class allows the updating of the DDNS binding to the new starting node's IP address. OverCord executes a command that runs the ddclient command, and uses file stream readers to read the response from the server. The response is either a success acknowledgment or a failure message.

8.2.3 A second attempt - HTTP requests

The above method was easy to implement, but it was considered that it may not be the best way of achieving the aim. Firstly, using programs like ddclient imposes dependencies on the node running OverCord. The ddclient application, for example, depends on a working Perl installation on the node. Secondly, the overhead of an external application is unnecessary considering the relatively simple process of performing a DDNS update for most DDNS providers. By running the ddclient in verbose mode, it was discovered that the client send the request to the URL *http://members.dyndns.org/nic/update* and appends query string arguments for the domain to be updated and the new IP address to this URL. Since it is possible to determine this information from the application itself, a simpler technique could be provided that had with no extra dependencies associated with it. For example, it is possible to create a connection to a URL by constructing the string representation of the URL by concatenating the base URL above with appropriate query string arguments. An `InputStreamReader` object is created which reads

data from the connection to the DDNS provider. This new method was subsequently employed in the OverCord class `DynDNSHost` and included a custom Java `Authenticator` class which handles request authentication across the node's HTTP proxy and for the DDNS user account. The modified user agent has a GUI interface for specifying these values, which become integrated into the node's configuration. In normal use, the authenticator dialog is hidden, but when enabled, each time a new update is made the following image appears:

Figure 8.4: Authentication dialog for DynDNS.com service.

8.2.4 Node behaviour in a DDNS enabled environment

When a new user joins a peer-to-peer overlay, the node running OverCord tries to discover if the DDNS node serving that overlay is alive and accepting connections. Using the FQDN of the overlay, it obtains the IP address of the P2P SIP proxy node of the overlay by a DNS lookup procedure, and subsequently attempts to determine if the node is reachable. The reachability of the P2P SIP proxy is determined by whether or not it responds to a probe within a set amount of time. If it is not reachable, then the joining node assumes that the P2P SIP proxy is no longer available, and it must update the DDNS binding. If the indicated IP address is network reachable, the joining node must determine if the port number 49152 on that host is open. This is the default port used in the system for accepting probes from other nodes in the overlay, and according to the Internet Assigned Numbers Authority (IANA), is a private port number that does not conflict with any registered protocols [108]. If the port is not open, then the joining node must perform a DDNS update and begin itself listening for connections on the 49152 port. If the DDNS node is alive and accepting connections, the node starts as normal, but starts a thread that periodically sends probes to the P2P SIP proxy every ten minutes in order to detect a failure or exit in future.

In addition to listening for probes from neighbour nodes, the P2P SIP proxy is the first point of

contact into the overlay from the outside world. Any request destined for a user on that overlay, reaches this node which bears the responsibility of forwarding the request appropriately. When a message is received by the proxy, the node must examine the message to determine the target. If the Request URI indicated in the message belongs to the node, then it processes it as normal. If the Request URI does not belong to the node, it knows that it must proxy it to the correct node, by obtaining the contact address of the target user through a peer-to-peer lookup procedure, and forwarding it to the intended target.

8.2.5 Results and discussion

In order to test the success of the use of DDNS for interoperation, the popular open source SIP server called SIP Express Router (SER) [86] was used. In an experiment, an OverCord-enabled JSAP application was started which was owned by a hypothetical user called Alice, a member of the chord.dyndns.org domain. Her user agent was the first to join the domain, so the host address of her machine was entered in the DNS records for chord.dyndns.org. The user Alice had a hypothetical friend called linphone, a member of a centralised domain known as deebee.dsl.ru.ac.za, which is served by SER.

Firstly, presence subscription and advertisement were attempted. The user linphone registers with SER and subsequently sends a PUBLISH method to the server. The PUBLISH message attaches a Presence Information Document Format (PIDF) document which contains presence information, which SER will store. The user then sends a request to obtain the presence status of the user Alice who is in the chord.dyndns.org domain. SER proxies the request to the decentralised overlay by accessing the current P2P SIP proxy peer in the chord.dyndns.org domain, which happens to be Alice's machine itself. Signaling was successful, as shown by the message below which promptly appears on Alice's host machine, notifying her of the request from the remote user:

Figure 8.5: Presence request from centralised user agent to peer-to-peer UA.

In this case, Alice agrees to accept the subscription request from the remote user linphone, and sends a notification to that user. Alice will subsequently generate a request of her own, subscribing to the presence status of linphone. SER allocates a lease on the presence status of linphone to Alice and sends a NOTIFY message to Alice, informing her of the present status of this user. The JSAP application must handle refreshing of these leases since the SIP stack does not do this on the behalf of the UA itself. The contact list on the user linphone's UA appears below in Figure 8.6 showing the insertion of the user Alice and her status. Evidence of a successful instant messaging session between the two users is given in Figure 8.7.

Figure 8.6: Presence status of peer-to-peer UA acquired by centralised UA.

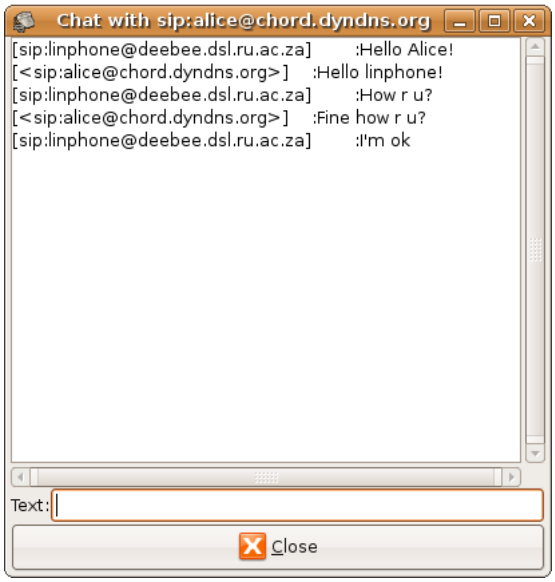

Figure 8.7: Instant messaging session a client-server and peer-to-peer UA.

The screenshots shown above validate the fact that it is possible to achieve the interoperation from centralised to decentralised networks using dynamic updates in DNS. However, as has been explained, OverCord is able to facilitate routing of external messages to overlay nodes, by virtue of the proxy behaviour embedded in the P2P SIP proxy. To prove that the design is able to achieve this, another user is introduced, named Bob, who joins the chord.dyndns.org overlay after Alice, but while Alice's node is still the P2P SIP proxy peer. In the experiment, it was possible to provide presence and instant messaging communication between linphone and Bob through Alice's user agent performing the proxy function, as shown in Figure 8.8 and Figure 8.9.

Figure 8.8: Presence status sharing supported by P2P SIP proxy.

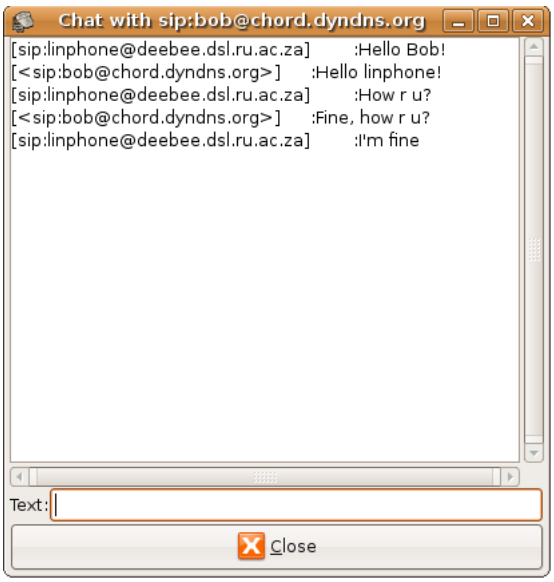

Figure 8.9: Instant messaging session supported by P2P SIP proxy.

8.2.6 Summary

The ability for OverCord to interoperate with client-server systems is a valuable feature, essentially bridging two types of network overlays, decentralised and conventional. The fact that DDNS is used does not add a centrality component to peer-to-peer overlays since the aim of DDNS is to accommodate a dynamic relationship between a domain name and the resource record it maps to, as well as incorporating redundancy through multiple bindings to a domain name.

The free DDNS service used in the experiments described in this chapter provides only one mapping per domain name, which means that high network latency may possibly lead to a node assuming that the single DDNS node has failed due to a failure to connect. Even the five second timeout employed may not be able to prevent this. This situation may arise, for example, in the event that a firewall or other security policy has been implemented on the P2P SIP proxy peer, which refuses unsolicited connections altogether, which may happen with or without the human user's knowledge. In this case, the querying node would unnecessarily update the DNS resource record to map to its own address, despite the DDNS node actually being alive. This kind of behaviour does not compromise the operation of the system, since, until the change in the binding is realised globally across all client caches, the overlay will have two DDNS

representatives, until those caches expire, and only the new node is used.

Chapter 9

Conclusion

> *The shrewd guess, the fertile hypothesis, the courageous leap to a tentative conclusion - these are the most valuable coin of the thinker at work.*
>
> - Jerome S. Bruner, 1960

This thesis provides an overview of the effort towards the standardisation of protocols for P2P SIP. Using this discussion as background, it presents the design and implementation of a unique decentralised framework for SIP and details its subsequent incorporation into a conventional SIP UA. The framework, which is called OverCord, addresses the four main objectives that were given in the introductory chapter by creating a distributed service platform that is able to accommodate several structured peer-to-peer protocols, and support interoperation between decentralised overlays and conventional SIP networks. This chapter concludes the thesis by providing an analysis of the work done, summarising the main contributions and giving suggestions for future work.

9.1 Achieved objectives

In this section the set of objectives outlined section 1.4 are revisited. The degree to which the work discussed in this thesis was able to achieve these objectives will be analysed, and other important issues that arose as a result are discussed.

9.1.1 Investigate peer-to-peer protocols for SIP

The investigation into the area of peer-to-peer protocols revealed that there are two major groups: structured and unstructured. By considering the requirements of a P2P SIP system, it is evident that structured protocols such as DHTs are well suited to provide what is needed for the decentralisation of SIP. The proposals by the IETF working group for P2P SIP have been completely dominated by DHTs, and as such, the project would focus largely on their use in a peer-to-peer system. However, there is inconclusive evidence to support the use of DHTs over unstructured protocols for P2P SIP.

9.1.2 Provide a framework for overlay pluggability

The second objective was to provide a framework that would allow implementations of DHTs to be plugged into it as a means to create a DHT-agnostic peer-to-peer layer. This framework would allow an application to interact with the underlying DHT layer through a generic interface. To support this, OverCord was designed to provide a simplified API to a higher level application and supports pluggability of DHTs by defining a space for third party plug-ins which interact directly with the DHT modules. It was demonstrated through two example DHT implementations, Bamboo and OpenChord, that the complexity of the construction of the plug-ins was directly related to the DHTs themselves and the manner in which they provide external access to the plug-ins.

9.1.3 Interoperation between heterogeneous overlays

The third objective was to prove that the framework could be used as an overlay construction tool that would allow interoperation between overlays that are based on heterogeneous DHTs. The ability to create and maintain overlays was demonstrated by means of a sample application. Furthermore, it was demonstrated that an overlay discovery layer could be incorporated into

OverCord, that could be used to assist the DHTs with a discovery function. As an example, a combination of unicast and multicast were used.

9.1.4 Interoperation with client-server SIP systems

The last objective of the research was to study the possibility of interoperating peer-to-peer overlays with conventional client-server based SIP networks. This was important so as to break out of the limitations of a disconnected network, and would make P2P SIP more appealing. The thesis shows that communication between the two types of networks could be provided through the use of a P2P SIP proxy peer residing in the peer-to-peer overlay that had a binding in DNS, and could relay traffic between them. Support for dynamic updates to DNS would also ensure that the churn factor in peer-to-peer systems did not become an obstacle in the provision of interoperability.

9.2 Contributions to P2P SIP research community

The P2P SIP working group, like all IETF working groups, is an open community of developers, academics and researchers and is open to any interested individuals [109]. It was necessary in the initial stages of this investigation to be attentive to debates on the mailing lists in order to better understand the crucial problems that needed to be solved, and to gain a better appreciation for the many different areas of contention. In time, it became possible not only to monitor exchanges, but also to contribute personal opinions on several occasions to the discussions that are currently shaping the P2P SIP protocol. Notably, as far as can be determined, the author was the first to propose a design that interoperates between heterogeneous peer-to-peer overlays in a way that is not reliant on an hierarchical network topology [110].

The successful creation of a framework for peer-to-peer systems made it possible to attempt to embed peer-to-peer functionality into existing source code. When a copy of the JAIN SIP applet phone was obtained and an initial investigation into the software commenced, a number of bugs were uncovered in the original software. In order to complete the integration of OverCord, it was necessary to fix these bugs. At the time of writing, correspondence with the developers of the software has been initiated, and the possibility of committing fixes to the code has been welcomed. Moreover, the developers of the software have recently invited the author to be a co-owner of the JAIN SIP Applet phone project in order to help maintain it, which may also lead to OverCord becoming a possible component of the project as a whole in future.

9.3 Future work

OverCord was designed to be modular and thus support future work and extensions. It has been noted that the use of the multicast discovery technique in the current OverCord implementation is limited since it cannot discover nodes across most IPv4 routers. There are several other peer-to-peer protocols that could be incorporated as future work and used to improve the success of the discovery layer. The investigation would also evaluate the ease of incorporating these protocols into OverCord.

The current implementation of OverCord exclusively uses DHTs but it is hoped that the generic interface and the plug-in theory can also be applied to unstructured protocols as well. Future work could concentrate on the development of plug-ins for some of the unstructured protocols such as those described in section 3.2.3. This work could also investigate the ease of developing these plug-ins in comparison to those for structured protocols.

The experiments that were performed using the simplified interface to OverCord were conducted on a LAN with only a few machines participating in the overlays. It is possible that some aspects of the design may require revision or optimisation when the network size is significantly larger than just a handful of nodes. This could help test the scalability of the system. Virtual machines could be used to simulate large networks and test communications between virtual nodes. Another possible platform for conducting tests is PlanetLab [111], which can be used to test OverCord across hundreds of nodes across the world.

OverCord was designed to support a wide range of services that can be supported by a distributed storage and retrieval system. In this project, this claim was only verified for the purposes of supporting SIP applications. Future work could try to establish the suitability of OverCord for supporting other distributed services besides telephony such as file sharing and peercasting.

The user agent that was modified with peer-to-peer functionality as part of this project was able to support instant messaging, presence and audio communications in a distributed environment. SIP allows for the provision of more complex telephony services such as call holding, call forwarding, 3-way conferencing and call transfer [112]. Future work could evaluate the suitability of OverCord to enable such services.

9.4 Summary

This thesis has detailed an alternative deployment strategy for SIP where multimedia communications can be achieved without relying on static entities in the network. At the time of writing, a protocol for P2P SIP has not yet been standardised. The thesis advocates for the use of DHTs for this purpose and shows how interoperation between peer-to-peer overlays and client-server networks can be achieved.

References

[1] D Bryan, E Shim, and B Lowekamp. Use cases for peer-to-peer session initiation protocol (p2p sip). Available Online, November 2005. Internet Draft: draft-bryan-sipping-p2p-02, work in progress, http://www.p2psip.org/drafts/draft-bryan-sipping-p2p-usecases-00.txt.

[2] Kazaa - Search Download and Share. Available Online. URL: http://www.kazaa.com.

[3] *BitTorrent*. URL: http://www.bittorrent.com/.

[4] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM Press.

[5] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350. ACM, November 2001.

[6] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A Scalable Content-Addressable Network. In *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[7] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical Report UCB/CSD-01-1141, University of California, Berkeley, Berkeley, CA, USA, April 2001.

[8] A jain sip applet phone for the people! Available Online. URL: https://jain-sip-applet-phone.dev.java.net/.

[9] J Rosenberg, H Schulzrinne, G Camarillo, A Johnston, J Peterson, R Sparks, M Handley, and E Schooler. RFC 3261: SIP: Session Initiation Protocol, 2002.

[10] H Schulzrinne, S Casner, R Frederick, and V Jacobson. RFC 1889: A Transport Protocol for Real-Time Applications, January 1996.

[11] H Schulzrinne, R Rao, and R Lanphier. RFC 2326: Real Time Streaming Protocol, April 1998.

[12] F Cuervo, N Greene, A Rayhan, C Huitema, and J Rosenberg. RFC 3015: Megaco Protocol Version 1.0, November 2000.

[13] Gonzalo Camarillo. *SIP Demystified*. McGraw Hill Professional, 2001.

[14] Alan Johnston. *Understanding the Sessions Initiation Protocol*. Artech House, 2004.

[15] Gonzalo Camarillo. The Internet Multimedia Conferencing Architecture. In *SIP Demystified*, pages 56–59. McGraw Hill Professional, 2001.

[16] Alan Johnston. SIP and the Internet. In *Understanding the Sessions Initiation Protocol*, pages 1–4. Artech House, 2004.

[17] Jorg Ott. Location servers. Seminar Series, 2001. URL: http://www.cs.columbia.edu/sip/talks/von2001-sip-wg-status.pdf.

[18] J Rosenberg and H Schulzrinne. RFC 3263: Session Initiation Protocol (SIP)- Locating SIP Servers, June 2002.

[19] M Handley, V Jacobson, and C Perkins. RFC 4566: SDP: Session Description Protocol. July 2006.

[20] AB Roach. RFC 3265: Session Initiation Protocol (SIP)-Specific Event Notification, June 2002.

[21] A Niemi. RFC 3903: Session Initiation Protocol (SIP) Extension for Event State Publication, October 2004.

[22] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.

[23] M Castro, M Costa, and A Rowstron. Peer to Peer Overlays: Structured or Unstructured or Both? Technical Report MSR-TR-2004-73, Microsoft Research, July 2004.

[24] D Karger, E Lehman, T Leighton, R Panigrahy, M Levine, and D Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on the Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM Press.

[25] Josh Cates. Robust and Efficient Data Management for a Distributed Hash Table. Master's thesis, Massachusetts Institute of Technology, United States of America, June 2003.

[26] Frank Dabek. A Cooperative File System. Master's thesis, Massachusetts Institute of Technology, United States of America, September 2001.

[27] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215, New York, NY, USA, 2001. ACM.

[28] Freepastry. Available Online. URL: http://freepastry.rice.edu/FreePastry/.

[29] Simpastry/VisPastry. Available Online. URL: http://research.microsoft.com/users/Cambridge/antr/pastry/download.htm.

[30] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20:1489–1499, oct 2002.

[31] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 188–201, New York, NY, USA, 2001. ACM.

[32] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A decentralized peer-to-peer web cache. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 213–222, New York, NY, USA, 2002. ACM.

[33] A Robust Open Source DHT. Available Online. URL: http://www.bamboo-dht.org/.

[34] OpenDHT: A Publicly Accessible DHT Service. Available Online. URL: http://opendht.org/.

[35] JXTA: Meteor. Available Online. URL: https://jxta-meteor.dev.java.net/.

[36] Sipdht project. Available Online, 2007. URL: http://sipdht.sourceforge.net/.

[37] Swapnil Pundkar. Peer to peer communication over the internet: An open approach. Available Online, July 2007. URL: http://sipdht.sourceforge.net/sipdht2/swapnil-report.pdf.

[38] Infrastructure for Resilient Internet Systems. Available Online. URL:http://project-iris.net.

[39] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Peer to Peer Systems II*, volume 2735/2003, pages 33–44. Springer Berlin / Heidelberg, October 2003.

[40] Overlay Weaver: An Overlay Construction Toolkit. Available Online. URL: http://overlayweaver.sourceforge.net/.

[41] Gnutella. Available Online. Gnutella, http://www.gnutella.com/.

[42] Matei Ripeanu, Adriana Iamnitchi, and Ian Foster. Mapping the gnutella network. *IEEE Internet Computing*, 6(1):50–57, 2002.

[43] Igor Ivkovic. Improving gnutella protocol: Protocol analysis and research proposals. Available Online, 2001. URL: http://www.cs.cornell.edu/people/egs/615/gnutella.pdf.

[44] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-like P2P Systems Scalable. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 407–418, New York, NY, USA, 2003. ACM.

[45] E Lua, J Crowcroft, M Pias, R Sharma, and S Lim. A survey and comparison of peer to peer overlay network schemes. *IEEE Communications Survey and Tutorial*, 7(2):72–93, March 2005.

[46] Shashidhar Merugu, Sridhar Srinivasan, and Ellen Zegura. Adding Structure to Unstructured Peer-to-Peer Networks: The Use of Small-World Graphs. *J. Parallel Distrib. Comput.*, 65(2):142–153, 2005.

[47] Dongyu Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 367–378, New York, NY, USA, 2004. ACM Press.

[48] UDP Host Cache - Gnutella Specification. Available Online, May 2005. URL: http://gnutella-specs.rakjar.de/index.php/UDP_Host_Cache.

[49] Miguel Castro, Manuel Costa, and Antony Rowstron. Debunking some myths about structured and unstructured overlays. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 85–98, Berkeley, CA, USA, 2005. USENIX Association.

[50] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202, New York, NY, USA, 2006. ACM.

[51] Columbia University. The p2p-sip archives. Available Online. URL:https://lists.cs.columbia.edu/pipermail/p2p-sip/.

[52] K Singh and H Schulzrinne. Peer-to-Peer Internet telephony using SIP. In *NOSSDAV '05: Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 63–68, New York, NY, USA, 2005. ACM Press.

[53] K Singh and H Schulzrinne. SIPPeer: A session initiation protocol (SIP)-based peer to peer internet telephony client adaptor. Available Online (White Paper), January 2005. URL: www1.cs.columbia.edu/ kns10/publication/sip-p2p-design.pdf.

[54] D Bryan, B Lowercamp, and C Jennings. Sosimple: A serverless, standards-based, p2p sip communication system. In *AAA-IDEA '05: Proceedings of Advanced Architectures and Algorithms for Internet Delivery and Applications, First International Workshop*, pages 42–49. IEEE, 2005.

[55] Sipeerior technologies. Available Online. URL:http://www.sipeerior.com.

[56] P2psip working group charter. Available Online, 2007. URL:http://www.ietf.org/html.charters/p2psip-charter.html.

[57] D Bryan, P Matthews, E Shim, and D Willis. Concepts and terminology for peer to peer sip. Available Online, June 2007. Internet Draft: draft-ietf-p2psip-concepts-00, work in progress, http://www.ietf.org/internet-drafts/draft-ietf-p2psip-concepts-00.txt.

[58] D Bryan, B Lowekamp, and C Jennings. dsip: A P2P Approach to SIP Registration and Resource Location. Available Online, February 2007. Internet Draft, draft-bryan-sipping-p2p-02, work in progress, URL:http://tools.ietf.org/wg/p2psip/draft-bryan-p2psip-dsip-00.txt.

[59] C Jennings, J Rosenberg, and E Rescorla. Address settlement by peer to peer. Available Online, July 2007. Internet Draft: draft-jennings-p2psip-asp-00, work in progress, http://tools.ietf.org/html/draft-jennings-p2psip-asp-00.txt.

[60] D Bryan, M Zangrilli, and B Lowecamp. Resource location and discovery (reload), July 2007. Internet Draft, draft-bryan-p2psip-reload-01, work in progress, http://tools.ietf.org/html/draft-bryan-p2psip-reload-01.txt.

[61] D Bryan, P Matthews, E Shim, and D Willis. Concepts and terminology for peer to peer sip. pages 23–24. June 2007. Internet Draft: draft-ietf-p2psip-concepts-00, work in progress, http://www.ietf.org/internet-drafts/draft-ietf-p2psip-concepts-00.txt.

[62] J. Hautakorpi and G Camarillo. The peer protocol for p2psip networks. Available Online, February 2007. Internet Draft: draft-hautakorpi-p2psip-peer-protocol-00, work in progress, draft-hautakorpi-p2psip-peer-protocol-00.txt.

[63] S Baset and H Schulzrinne. Peer to Peer Protocol (P2PP). Available Online, July 2007. Internet Draft: draft-baset-p2psip-p2pp-00, work in progress, http://tools.ietf.org/html/draft-baset-p2psip-p2pp-00.txt.

[64] K Singh and H Schulzrinne. Data format and interface to an external peer-to-peer network for SIP. Available Online, May 2006. Internet Draft: draft-singh-p2p-sip-00, work in progress, http://tools.ietf.org/html/draft-singh-p2p-sip-00.txt.

[65] K Singh and H Schulzrinne. Using an external DHT as a SIP location service. Technical Report CUCS-007-06, Columbia University, USA, February 2006.

[66] J Rosenberg. Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. Available Online, June 2007.

Internet Draft: draft-ietf-mmusic-ice-16, work in progress, http://tools.ietf.org/id/draft-ietf-mmusic-ice-16.txt.

[67] J Rosenberg and H Schulzrinne. RFC 3264: An Offer/Answer Model with Session Description Protocol, June 2002.

[68] J Rosenberg, J Weinberger, C Huitema, and R Mahy. RFC 3469: Simple traversal of user datagram protocol (UDP) through network address translators (NATs), March 2003.

[69] D Bryan, B Lowercamp, and C Jennings. A peer to peer approach to sip registration and resource location. March 2006. Internet Draft: draft-bryan-sipping-p2p-02, work in progress, http://tools.ietf.org/html/draft-bryan-sipping-p2p-02.txt.

[70] E Marocco and E Ivov. Extensible peer protocol (xpp), June 2007. Internet Draft: draft-marocco-p2psip-xpp-00, work in progress, http://tools.ietf.org/html/draft-marocco-p2psip-xpp-00.txt.

[71] Mosiuoa Tsietsi, Alfredo Terzoli, and George Wells. A Peer to Peer Layer for SIP Based Realtime Multimedia Communication. In *SATNAC Conference Proceedings*, 2007.

[72] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2006.

[73] Sven Kaffille and Karsten Loesing. *OpenChord version 1.0.3 User's Manual*. Bamberg University, third edition, October 2007.

[74] M Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, University of California, Berkeley, August 2002.

[75] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM Press.

[76] Marcel Dischinger. A Flexible and Scalable Peer to Peer Multicast Application Using Bamboo. Master's thesis, University of Cambridge, June 2004.

[77] The OceanStore Project. Available Online. URL: http://oceanstore.cs.berkeley.edu/.

[78] Oracle Berkeley DB Java Edition. Available Online. URL: http://www.oracle.com/database/berkeley-db/je/index.html.

[79] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 36–44, New York, NY, USA, 1998. ACM.

[80] L Garces-Erice, EW Biersack, PA Felber, KW Ross, and G Urvoy-Keller. Hierarchical Peer to Peer Systems. In *Euro-Par 2003 Parallel Processing*, volume 2790/2004 of *Peer to Peer Computing*, pages 1230–1239. Springer Berlin/Heidelberg, 2004.

[81] Wengophone - call, talk, chat and share for free with anyone. Available Online. URL: http://www.wengophone.com/.

[82] NIST - Project IP Telephony. Available Online. URL: http://www-x.antd.nist.gov/proj/iptel/.

[83] Sofia-sip library. Available Online, 2007. URL: http://sofia-sip.sourceforge.net/.

[84] OSIP - The GNU oSIP Library. Available Online. URL: http://www.gnu.org/software/osip/.

[85] SIPX - Open Source VOIP, The SIP Revolution Continues. Available Online, 2007. URL: http://www.sipfoundry.org/.

[86] SIP Express Router. Available Online. URL:http://www.iptel.org/ser.

[87] Cinema - columbia internet extensible multimedia architecture. Available Online. URL: http://www.cs.columbia.edu/IRT/cinema/.

[88] Vocal - your source for open source communication. Available Online. URL: http://www.vovida.org/.

[89] Java community process program -jcp procedures -jcp2 - process document. Available Online. URL: http://jcp.org/en/procedures/jcp2.

[90] P O'Doherty, A Kristensen, C Bouret, and M O'Doherty. SIP specifications and the Java platforms. Available Online, September 2003. URL: http://www.cs.columbia.edu/sip/Java-SIP-Specifications.pdf.

[91] JAIN SIP API Specification. Available Online, 2006. URL: http://jcp.org/en/jsr/detail?id=32.

[92] SIP Servlet API. Available Online, 2003. URL: http://jcp.org/en/jsr/detail?id=116.

[93] JAIN SIP Lite. Available Online, 2002. URL: http://jcp.org/en/jsr/detail?id=125.

[94] SIP for J2ME Specification. Available Online, 2006. URL: http://jcp.org/en/jsr/detail?id=180.

[95] JAIN SIP. Available Online. URL: https://jain-sip.dev.java.net/.

[96] NIST Labs. Nist sip: Reference implementation for jain sip 1.2. Available Online, 2006. URL: http://snad.ncsl.nist.gov/proj/iptel/jain-sip-1.2/javadoc/.

[97] P O'Doherty. Jain SIP Tutorial - Serving the Developer Community,http://www-x.antd.nist.gov/proj/iptel/tutorial/jain-sip-tutorialv2.pdf, 2003.

[98] Apache Ant - The Apache Ant Project. Available Online. URL: http://ant.apache.org/.

[99] J Rosenberg, D Willis, R Sparks, B Campbell, H Schulzrinne, J Lennox, B Aboba, and C Huitema D Gurle. A data format for presence using XML. Available Online, June 2000. Internet Draft: draft-rosenberg-impp-pidf-00, work in progress, http://www.jdrosen.net/papers/draft-rosenberg-impp-pidf-00.txt.

[100] J Rosenberg and H Schulzrinne. RFC 3840: Indicating user agent capabilities in the session initiation protocol (sip), 2004.

[101] E Shim, S Narayanan, and G Daley. An Architecture for Peer to Peer Session Initiation Protocol (P2P SIP). Available Online, February 2006. Internet Draft: draft-shim-sipping-p2p-arch-00, work in progress, http://www.p2psip.org/drafts/draft-shim-sipping-p2p-arch-00.txt.

[102] E Marocco and D Bryan. Interworking between P2PSIP Overlays and Conventional SIP Networks, March 2007. Internet Draft: draft-marocco-p2psip-interwork-01, work in progress, http://tools.ietf.org/html/draft-marocco-p2psip-interwork-01.txt.

[103] J Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). Available Online, November 2007. Internet Draft: draft-ietf-behave-turn-05, work in progress, http://tools.ietf.org/html/draft-ietf-behave-turn-05.

[104] R Droms. RFC 2131: Dynamic Host Configuration Protocol, 1997.

[105] P Vixie, S Thomson, Y Rekhter, and J Bound. RFC 2136: Dynamic Updates in the Domain Name System (DNS UPDATE). Available Online, 1997.

[106] Sourceforge.net: ddclient. Available Online. URL:http://sourceforge.net/projects/ddclient/.

[107] DynDNS.com. Dyndns – free dynamic dns (ddns) service. Available Online, 2007.

[108] Port numbers. Available Online. URL: http://www.iana.org/assignments/port-numbers.

[109] Overview of the IETF. Available Online. URL: http://www.ietf.org/overview.html.

[110] Interoperation between heterogenous p2p overlay networks. Available Online, August 2006. URL: http://www1.ietf.org/mail-archive/web/p2psip/current/msg01304.html.

[111] An open platform for developing and accessing planetary services. Available Online. URL: http://www.planet-lab.org/.

[112] A Johnston, R Sparks, C Cunningham, S Donovan, and K Summers. Session Initiation Protocol service examples. Available Online, January 2007. Internet Draft: draft-ietf-sipping-service-examples-12, work in progress, http://tools.ietf.org/html/draft-ietf-sipping-service-examples-12.

Appendix A

OverCord terminal outputs

A.1 Single node

This section provides the output of the experiment described in section 6.5 where a preliminary version of the plug-in management module was designed.

1. Plug-in listing - the application searches in the plug-in repository and tries to discover the plug-ins that are embedded in the system. In this case, it discovers two, lists their names, and allows the user to select which plug-in they would like to start.

```
--- Initialising ---
--- Looking for installed plugins ...
2 plugin(s) loaded. Names are:
chordplugin.ChordPlugin
bambooplugin.BambooPlugin
--- You can start an overlay with any of the following :
--- >> 1. chordplugin.ChordPlugin
--- >> 2. bambooplugin.BambooPlugin
--- Your choice:
```

3. Plug-in creation and control - In the experiment, the ChordPlugin plug-in which is based on the OpenChord DHT implementation is chosen. As the first node, it creates the overlay and begins running on the default port 3730. After the plug-in has successfully started, it provides the user with options for some operations that can be performed.

```
### Finished overlay discovery and bootstrapping module ... ###
### Node instantiation ... ###
Using plugin: chordplugin.ChordPlugin
Starting: 146.231.123.55 as bootstrap peer...
Creating overlay: chord.ru.ac.za
Booting parameters: 146.231.123.55 0
Running node on 146.231.123.55 on port 3730
[main] log4 configured with 'log4j.properties
***** Node options *****
1. Insert record
2. Retrieve record
3. Remove record
4. Leave overlay
Your value: [0]
```

4. Resource insertion - To insert a key, option 1 is selected. The plug-in manager prints out the syntax for inserting a new record, which is key followed by value, separated by a space. Only records that contain the FQDN of the overlay can be inserted. When the user has pressed the enter key, the plug-in manager passes these values to the plug-in, which uses the native syntax to request the insertion into the DHT.

```
Syntax: <key> <value> sip:mtsietsi@chord.ru.ac.za /
    sip:mtsietsi@146.231.123.55:5060
Key sip:mtsietsi@chord.ru.ac.za inserted
```

5. Resource retrieval - The record that was inserted in the previous step can be retrieved by choosing option 2 on the menu. The plug-in manager requests the user to enter the key, after which it returns the value associated with the key.

```
Syntax: <key> sip:mtsietsi@chord.ru.ac.za
Retrieved sip:mtsietsi@146.231.123.55:5060
```

A.2 Multiple Nodes

In the next stage of experimentation with the plug-in manager, the discovery module is introduced. The first node is kept alive and a new node on a different host machine tries to join the overlay by discovering any existing nodes.

1. Node detection - When the new node performs the discovery, it finds the first node, which returns details that the joining node needs to know in order to join the overlay through it. By choosing the option 0, the user tells the plug-in manager to repeat the discovery procedure. If option 1 is chosen, the node joins the overlay through the discovered node. If the option -1 is chosen, the discovery procedure exits, and the user can try to create an overlay that was not discovered (for which it will be the first node in that overlay).

```
### Starting overlay discovery and bootstrapping module ... ###
--- Listing overlays:
   --- Found overlay: (1)chord.ru.ac.za
        --- Host: 146.231.123.55
        --- Port: 3730
Your choice (0 to do rediscovery, -1 to quit discovery):
```

2. Bootstrapping existing overlay - Option 2 is chosen which prompts the plug-in manager to create the plug-in that is needed. The native join methods are used to join the overlay through the discovered node. The new node runs on port 3730. Options are presented to the user for operations that can be performed.

```
### Node instantiation ... ###
Using plug-in: chordplugin.ChordPlugin
Joining overlay: chord.ru.ac.za
Starting: 146.231.121.180 as non-bootstrap peer...
Booting parameters: 146.231.123.55 3730
Running node on 146.231.121.180 on port 3730
[main] log4j configured with 'log4j.properties'.
***** Node options *****
1. Insert record
2. Retrieve record
3. Remove record
4. Leave overlay
Your value: [0]
```

A.3 Heterogeneous overlays

The last stage of experimentation with the plug-in manager addresses interoperating between heterogeneous overlays.

8. Resource insertion (Node A - OpenChord) - A node in the chord.ru.ac.za overlay inserts a record into the location service.

```
Syntax: <key> <value> sip:mtsietsi@chord.ru.ac.za /
    sip:mtsietsi@146.231.123.55:5060
```

9. Resource retrieval (Node B - Bamboo) - A node in the bamboo.ru.ac.za overlay needs the contact record of the node above. The plug-in manager on the Bamboo node realises that this resource record cannot be found in the local location service since the RHS points to a foreign overlay. It performs a discovery and tries to find responses that are specifically from the chord.ru.ac.za overlay. It gets such a response and tries to start an appropriate plug-in for that overlay. The plug-in is started and a get request is issued. The record is retrieved and the contact address is passed the plug-in manager of the Bamboo node and the new plug-in is terminated.

```
Syntax: <key> sip:mtsietsi@chord.ru.ac.za
Parameters: chord.ru.ac.za chordplugin.ChordPlugin /
    146.231.121.180 3730
--- Compatible plugin found
Starting: chordplugin.ChordPlugin as non-bootstrap peer...
Booting parameters: localhost 3730
Running node on 146.231.123.55 on port 3731 via bootstrap 3730
[main] log4j configured with 'log4j.properties'.
Retrieved sip:mtsietsi@146.231.123.55:5060
```

Appendix B

OverCord Classes

B.1 Plugin Interface

```
/**
 * Plugin: An interface class that all DHT plugins implement
 * All implementors have to have a description method as well
 * as methods for inserting, retrieving and removing keys
 * from the DHT
*/
package gov.nist.applet.phone.locationcomponent.pluginmanagement;
/**
 * @author mtsietsi
 *
 */
public interface Plugin {
  //Descriptors of plugin
  void description();
  String getOverlays();
  int getPort();
  String getBootstrapHost();
  int getBootstrapHostPort();
  //DHT membership interface
  void joinOverlay();
```

```
  void leaveOverlay();
  //DHT management interface
  void put(String key, String value);
  String get(String key);
  void remove(String key, String value);
  //Convenience methods
  void setDomain(String newDomain);
  void updateStatus(String type, String value);
}
```

B.2 Multicast Discovery

```
/**
 * McastLayer: Class for multicast discovery of overlays.
 * This class is used in PluginManager.
 * It multicasts a bootstrap requests to peers on the
 * sip.mcast.net address and awaits to receive bootstrap
 * information which is in the form
 * UNICAST_OVERLAYID_PLUGINCLASS_HOST_PORT of the
 * contacted peer
 */
package gov.nist.applet.phone.locationcomponent.
                    overlaydiscovery.multicastdiscovery;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;
import java.net.SocketException;
import java.util.StringTokenizer;
import java.util.Vector;
/**
 * @author mtsietsi
 *
 */
```

```
public class McastLayer {
  private int m_listenPort;
  private boolean m_found;
  private String m_message,m_strAddress;
  private MulticastSocket m_socket;
  private InetAddress m_castGroup;
  private DatagramPacket m_packet;
  private Vector m_bootstrapList,m_bootDisplayList;
  private int m_numResponses,m_numHits,m_numDisplay,m_maxPort;
  public McastLayer(){
    m_listenPort = 0;
    m_strAddress = "sip.mcast.net";
    m_listenPort = PortScanner.getPort();
    m_found = false;
    m_bootstrapList = new Vector(5,2);
    m_bootDisplayList = new Vector(5,2);
    m_numResponses = m_numHits = m_maxPort = 0;
  }
  public void clearList(){
    m_found = false;
    m_bootstrapList = new Vector(5,2);
    m_bootDisplayList = new Vector(5,2);
    m_numResponses = m_numHits = m_numDisplay = 0;
  }
  public void discover() {
    try{
      //join mcast group
      m_socket = new MulticastSocket(m_listenPort);
      m_castGroup = InetAddress.getByName(m_strAddress);
      m_socket.setSoTimeout(1000);
      //m_socket.joinGroup(m_castGroup);
      //set the message data
      byte[] inbuf = new byte[512];
      byte[] outbuf = new byte[512];
```

```
        m_message = new String("MCAST_BOOTSTRAP");
        outbuf = m_message.getBytes();
        //set up sockets and datagrams
        m_packet = new DatagramPacket(m_message.getBytes(),
           m_message.length(), m_castGroup, m_listenPort);
        //send multicast message
        m_socket.send(m_packet);
        //set up datagram for receiving
        m_packet = new DatagramPacket(inbuf, inbuf.length);
        //receive responses and store data
        //loop to get multiple overlays
        String unformattedDetails;
        while(true){
          try{
            m_socket.receive(m_packet);
            inbuf = m_packet.getData();
            unformattedDetails = new String(inbuf);
            unformattedDetails = unformattedDetails.trim();
            if(unformattedDetails.startsWith("UNICAST")){
              setDetails(unformattedDetails);
              unformattedDetails = "";
              setFoundOverlays(true);
              m_packet = new DatagramPacket(inbuf, inbuf.length);
            }
          }catch(SocketException so_ex){
            //socket timed-out
            setFoundOverlays(false);
            break;
          }
        }
      }catch (IOException io_ex) {
      }
  }
    //sets the relevant details for bootstrapping obtained
```

```
//from the bootstrap node
public void setDetails(String details){
  String temp;
  temp = "";
  String strDomain,strPluginClass,strHost,strPort;
  StringTokenizer tokenizer;
  String[] tokens = new String[5];
  Vector generalVec = new Vector();
  Vector displayVec = new Vector();
  int counter = 0;
  strDomain = strPluginClass = strHost = strPort = "";
  // we are expecting to tokenize for five tokens
  tokenizer = new StringTokenizer(details,"_");
  while(tokenizer.hasMoreTokens()){
    tokens[counter] = tokenizer.nextToken();
    counter++;
  }
  strDomain = tokens[1];
  strPluginClass = tokens[2];
  strHost = tokens[3];
  strPort = tokens[4];
  if(strPort.length() > 4){
    strPort = strPort.substring(0,4);
  }
  strDomain = strDomain.trim();
  strPluginClass = strPluginClass.trim();
  strHost = strHost.trim();
  strPort = strPort.trim();
  if(!domainExists(strDomain)){
    //new domain
    displayVec.insertElementAt(strDomain,0);
    displayVec.insertElementAt(strPluginClass,1);
    displayVec.insertElementAt(strHost,2);
    displayVec.insertElementAt(strPort,3);
```

```
        m_bootDisplayList.add(m_numDisplay,displayVec);
        m_numDisplay++;
      }
      else{
        //have a binding already, but is it highest port number?
        if(isHigherPort(strPort)){
          updatePortBinding(strDomain,strPort);
        }
      }
      //insert into general list
      generalVec.insertElementAt(strDomain,0);
      generalVec.insertElementAt(strPluginClass,1);
      generalVec.insertElementAt(strHost,2);
      generalVec.insertElementAt(strPort,3);
      m_bootstrapList.add(m_numResponses,generalVec);
      m_numResponses++;
    }
    //updates the bootstrap peer port for this domain
    public void updatePortBinding(String domain, String port){
      Vector vec = new Vector();
      String tempDomain;
      for(int i = 0; i < m_bootDisplayList.size(); i++){
        vec = (Vector)m_bootDisplayList.elementAt(i);
        tempDomain = (String)vec.elementAt(0);
        tempDomain = tempDomain.trim();
        if(tempDomain.compareTo(domain) == 0){
          vec.setElementAt(port,3);
          m_bootDisplayList.setElementAt(vec,i);
          break;
        }
      }
    }
    //lists all the overlays that were detected
    public void listOverlays(){
```

```
    Vector tempVec = new Vector();
    String buildList = "";
    String domain;
    int counter = 0, peerCounter = 0;
    for(int i = 0; i < m_bootDisplayList.size(); i++){
      try{
        tempVec = (Vector) m_bootDisplayList.elementAt(i);
        domain = (String)tempVec.elementAt(0);
        System.out.println("---Found overlay:
           ("+Integer.valueOf(counter+1).toString()+")"+
                                          tempVec.elementAt(0));
        System.out.println("\t--- Host: "+tempVec.elementAt(2));
        System.out.println("\t--- Port: "+tempVec.elementAt(3));
        System.out.println("\t--- PluginClass: "+
                                          tempVec.elementAt(1));
        counter++;
      }catch(ClassCastException cl_ex){
        break;
      }
    }
  }
  //sets the success of the bootstrap operation
  public void setFoundOverlays(boolean flag){
    m_found = flag;
  }
  //gets the success of the bootstrap operation
  public boolean getFoundOverlays(){
    return m_found;
  }
  //gets the number of detected overlays
  public int getNumOverlays(){
    //return m_numResponses;
    return m_bootDisplayList.size();
  }
```

```
//have we already got bootstrap details for this domain?
public boolean domainExists(String domain) {
  String tempDomain;
  Vector vec = new Vector();
  for(int i = 0; i < m_bootDisplayList.size(); i++) {
    vec = (Vector)m_bootDisplayList.elementAt(i);
    tempDomain = (String)vec.elementAt(0);
    tempDomain = tempDomain.trim();
    if(tempDomain.compareTo(domain) == 0)
      return true;
  }
  return false;
}
// we want the highest port number possible
public boolean isHigherPort(String port){
  Vector vec = new Vector();
  int portNum,tempPort;
  String strPort;
  m_bootDisplayList.size());
  try{
    portNum = Integer.parseInt(port);
    for(int i = 0; i < m_bootDisplayList.size(); i++){
      vec = (Vector)m_bootDisplayList.elementAt(i);
      strPort = (String) vec.elementAt(3);
      strPort = strPort.trim();
      tempPort = Integer.parseInt(strPort);
      if(portNum  > tempPort){
        return true;
      }
    }
    return false;
  }catch(NumberFormatException num_ex){
    return false;
  }
```

```
    }
    //gets the indicated overlay by selection index in terminal
    //gets highest order port number
    public Vector getOverlayProfile(int index){
      Vector vec = new Vector();
      String domain,temp,port;
      int max = 0,value = 0,desiredIndex;
      try{
        vec = (Vector)m_bootDisplayList.elementAt(index);
        return vec;
      }catch(ClassCastException cl_ex){
        return null;
      }
    }
    //gets the index of the sought after domain
    //gets highest order port number
    public int findOverlay(String key){
      Vector vec = new Vector();
      String domain,temp,port;
      int max = 0,value = 0,desiredIndex;
      for(int i = 0; i < m_bootDisplayList.size(); i++){
        vec = (Vector)m_bootDisplayList.elementAt(i);
        domain = (String) vec.elementAt(0);
        domain = domain.trim();
        if(key.contains(domain)){
          return i;
        }
      }
      return -1;
    }
  }
```

B.3 Plugin Manager

```
/**
 * PluginManagement: main class for managing plugins
 */
package gov.nist.applet.phone.locationcomponent.pluginmanagement;
import java.util.NoSuchElementException;
import java.util.Vector;
import java.util.Enumeration;
import java.net.InetAddress;
import java.net.NetworkInterface;
import java.net.SocketException;
import java.io.File;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.w3c.dom.*;
import gov.nist.applet.phone.locationcomponent.
                          overlaydiscovery.multicastdiscovery.*;
/**
 * @author mtsietsi
 */
public class PluginManager extends PluginCreator {
  public Enumeration m_enumPlugins;
  public McastLayer m_mcastLayer;
  public Plugin m_plugin;
  public Plugin m_tempPlugin;
  public InetAddress m_inetAddr;
  public String m_strAddress;
  public Vector m_domains;
  public Vector m_usernames;
  public boolean m_discover;
```

```
public boolean m_bootstrapPeer;
public String m_bootstrapHost;
public int m_bootstrapPort;
public Vector m_bootstrapList;
public DocumentBuilderFactory m_docBuilderFactory;
public DocumentBuilder m_docBuilder;
public Document m_document;

@SuppressWarnings("unchecked")
public PluginManager()        {
 // Variable declarations
 String domain, pluginClass;
 int countFoundOverlays,numPlugins,numOverlays;
 boolean foundaddr;
 // Initializations
 m_mcastLayer = new McastLayer();
 m_docBuilderFactory = DocumentBuilderFactory.newInstance();
 m_domains = new Vector();
 m_usernames = new Vector();
 m_bootstrapList = new Vector();
 m_discover = false;
 m_bootstrapPeer = false;
 m_plugin = null;
 m_tempPlugin = null;
 // Other variables
 pluginClass = "";
 m_strAddress = "";
 countFoundOverlays = numPlugins = numOverlays = 0;
 foundaddr = false;
 m_inetAddr = null;
 try{
   foundaddr = false;
   Enumeration enumface = NetworkInterface.getNetworkInterfaces();
   while(enumface.hasMoreElements()) {
```

```
            NetworkInterface netface = (NetworkInterface)
                                                  enumface.nextElement();
            if(netface.getName() != "lo") {
               Enumeration enumaddr = netface.getInetAddresses();
               while (enumaddr.hasMoreElements()){
                 //enumaddr.nextElement();
                 m_inetAddr = (InetAddress) enumaddr.nextElement();
                 m_strAddress = m_inetAddr.getHostAddress();
                 if(m_strAddress.indexOf("%") == -1 ) {
                   foundaddr = true;
                   break;
                 }
               }
            }
            if(foundaddr)
               break;
          }
        }
        catch(NoSuchElementException un_ex) {
          System.out.println("Could not get machine IP address");
          System.out.println("failed to start node, exiting");
          System.exit(0);
         }catch(SocketException sock_ex){
           System.out.println("Could not get machine IP address");
           System.out.println("Failed to start node, exiting");
           System.exit(0);
         }

         System.out.println("");
         System.out.println("### Starting overlay discovery and
                                        bootstrapping module ... ###");
         m_mcastLayer.clearList();
         m_mcastLayer.discover();
         m_discover = m_mcastLayer.getFoundOverlays();
```

```
//if we were able to discover overlays we list them
if(m_discover) {
   System.out.println("--- Listing discovered overlays:
                       ("+m_mcastLayer.getNumOverlays()+")");
   m_mcastLayer.listOverlays();
   numOverlays = m_mcastLayer.getNumOverlays();
   try{
     for(countFoundOverlays = 0; countFoundOverlays <
                         numOverlays; countFoundOverlays++) {
       try {
        Vector foundOverlay = m_mcastLayer.
                       getOverlayProfile(countFoundOverlays);
        domain = (String)foundOverlay.elementAt(0);
        pluginClass = (String)foundOverlay.elementAt(1);
        m_bootstrapHost = (String)foundOverlay.elementAt(2);
        String temp = (String)foundOverlay.elementAt(3);
        temp = temp.trim();
        m_bootstrapPort = Integer.parseInt(temp,10);
        try {
          if(verifyPlugin(pluginClass)) {
            System.out.println("---Compatible plugin found:
                                              "+pluginClass);
            m_domains.add(countFoundOverlays,
                                    new String(domain));
            Vector bootlist = new Vector();
            bootlist.add(0,new String(pluginClass));
            bootlist.add(1,new String(m_bootstrapHost));
            bootlist.add(2,new Integer(m_bootstrapPort));
            m_bootstrapList.add(countFoundOverlays,bootlist);
            //parse the xml file with user account details
            m_docBuilder = m_docBuilderFactory.
                                    newDocumentBuilder();
            m_document = m_docBuilder.parse
```

```
                                (new File("networks.xml"));
            m_document.getDocumentElement().normalize();
            NodeList listOfOverlays = m_document.
                            getElementsByTagName("network");
            int numOfRecords = listOfOverlays.getLength();
            for(int i = 0; i < numOfRecords; i++) {
              Node firstOverlayNode = listOfOverlays.item(i);
              if(firstOverlayNode.getNodeType()
                                 == Node.ELEMENT_NODE) {
                Element firstOverlayElement =
                                 (Element)firstOverlayNode;
                NodeList overlayList = firstOverlayElement.
                                getElementsByTagName("overlay");
                Element firstOverlayNameElement =
                                  (Element)overlayList.item(0);
                NodeList textFOList = firstOverlayNameElement.
                                                 getChildNodes();
                String strFOList = textFOList.item(0).
                                              getNodeValue().trim();
                NodeList usernameList = firstOverlayElement.
                                getElementsByTagName("username");
                Element userNameElement =
                                  (Element)usernameList.item(0);
                NodeList textUNList = userNameElement.
                                                 getChildNodes();
                String strUNList = textUNList.item(0).
                                              getNodeValue().trim();
                if(strFOList.compareToIgnoreCase
                                     (new String(domain)) == 0){
                  m_usernames.add(new String(strUNList));
                }
              }
            }
          }
```

```
                else {
                   System.out.println("--- Incompatible plugin class");
                }
               }catch(PluginNotCreatedException pn_ex) {
                   System.out.println(pn_ex.getMessage());
               }catch(SAXParseException saxpe) {
                   System.out.println("--- SaxParse Exception!!");
                }catch (SAXException e) {
                   Exception x = e.getException();
                   ((x == null) ? e : x).printStackTrace();
                }catch(ParserConfigurationException pce) {
                    System.out.println("--- Parser Exception!!");
                }catch(IOException ioex) {
                    System.out.println("--- IOException on File!!");
                }
              }catch(NumberFormatException num_ex) {
              }
            }
           }catch (NoSuchElementException e) {
               System.out.println("Invalid option!");
           }
        }
            // we get here after discovered domains were enumerated
            m_enumPlugins = PluginDetector.detect();
            if(m_enumPlugins == null) {
                   System.err.println("Error while detecting plugins");
                   return;
            }
            else {
              System.out.println("--- You can start an overlay with any
                                                  of the following :");
              pluginClass = "";
              for ( ; m_enumPlugins.hasMoreElements() ; ) {
                 pluginClass = (String)m_enumPlugins.nextElement();
```

```
if(isUniquePluginClass(pluginClass)) {
  try{
    m_plugin = createPlugin(pluginClass,m_strAddress,0);
    String strPluginDomain = m_plugin.getDomain();
    m_domains.add(countFoundOverlays,
                          new String(strPluginDomain));
    System.out.println(pluginClass);
    Vector bootlist = new Vector();
    bootlist.add(0,new String(pluginClass));
    bootlist.add(1,new String(m_strAddress));
    bootlist.add(2,new Integer(0));
    m_bootstrapList.add(countFoundOverlays,
                                      (Vector)bootlist);
    //parse the xml file with user account details
    m_docBuilder = m_docBuilderFactory.
                                    newDocumentBuilder();
    m_document = m_docBuilder.parse(
                             new File("networks.xml"));
    m_document.getDocumentElement().normalize();
    NodeList listOfOverlays = m_document.
                       getElementsByTagName("network");
    int numOfRecords = listOfOverlays.getLength();
    boolean foundName = false;
    for(int i = 0; i < numOfRecords; i++) {
     Node firstOverlayNode = listOfOverlays.item(i);
     if(firstOverlayNode.getNodeType() ==
                                Node.ELEMENT_NODE) {
      Element firstOverlayElement =
                           (Element)firstOverlayNode;
      NodeList overlayList = firstOverlayElement.
                    getElementsByTagName("overlay");
      Element firstOverlayNameElement =
                       (Element)overlayList.item(0);
      NodeList textFOList = firstOverlayNameElement.
```

```
                                        getChildNodes();
        String strFOList = textFOList.item(0).
                                  getNodeValue().trim();
        NodeList usernameList = firstOverlayElement.
                     getElementsByTagName("username");
        Element userNameElement =
                          (Element)usernameList.item(0);
        NodeList textUNList =
                        userNameElement.getChildNodes();
        String strUNList = textUNList.item(0).
                                   getNodeValue().trim();
        if(strFOList.compareToIgnoreCase(
                           m_plugin.getDomain()) == 0){
         m_usernames.add(countFoundOverlays,
                                  new String(strUNList));
         foundName = true;
        }
       }
      }
      if(foundName == false){
       m_usernames.add(countFoundOverlays,
                               new String("unassigned"));
      }
   }catch(PluginNotCreatedException e) {
      System.out.println("--- Plugin not created);
   }catch(ClassCastException e) {
      System.out.println("--- Tried to load a non-plugin
                                        as a plugin!!");
   }catch(SAXParseException saxpe){
         System.out.println("---SaxParse Exception!!");
   }catch (SAXException e){
     Exception x = e.getException();
     ((x == null) ? e : x).printStackTrace();
   }catch(ParserConfigurationException pce){
```

```
                System.out.println("--- Parser Exception!!");
              }catch(IOException ioex){
                System.out.println("--- IOException on File!!");
              }
            }
            numPlugins++;
          }
        }
        System.out.print("");
      }
       public boolean isUniquePluginClass(String pluginClass) {
         int counter;
         boolean flag = true;
         counter = m_bootstrapList.size();
         if(counter == 0) return flag;
         for (int i = 0; i < counter; i++) {
           Vector vector = (Vector) m_bootstrapList.elementAt(i);
           String className = (String) vector.elementAt(0);
           if(className.compareToIgnoreCase(pluginClass) == 0) {
              flag = false;
              break;
           }
         }
         return flag;
       }
       public void printBootlist() {
           int counter;
           counter = m_bootstrapList.size();
           for (int i = 0; i < counter; i++) {
             Vector vector = (Vector) m_bootstrapList.elementAt(i);
             String className = (String) vector.elementAt(0);
             String host = (String) vector.elementAt(1);
             Integer port = (Integer) vector.elementAt(2);
           }
```

```
}
public void printDomains() {
    int counter;
    counter = m_domains.size();
    for (int i = 0; i < counter; i++) {
      String domain = (String) m_domains.elementAt(i);
      System.out.println(domain);
    }
}
public Plugin getPlugin() {
   return m_plugin;
}
public Vector getDomains() {
   return m_domains;
}
public Vector getUsernames() {
   return m_usernames;
}
public Vector getBootList() {
   return m_bootstrapList;
}
public int getDHTPort() {
   return m_plugin.getPort();
}
public void register(String pluginToUse, String host,
                                                int port) {
   try{
     m_plugin = createPlugin(pluginToUse,host,port);
     m_plugin.joinOverlay();
   }
   catch(PluginNotCreatedException e) {
     System.out.println(e.getMessage());
   }
   catch(ClassCastException e) {
```

```
        System.out.println("--- Tried to load a non-plugin
                                              as a plugin!!");
      }
    }
    public void deregister(String key, String value) {
      m_plugin.remove(key, value);
      m_plugin.leaveOverlay();
      m_plugin = null;
    }
    public void createBinding(String key, String value) {
      m_plugin.put(key,value);
    }
    public String get(String key) {
      String strValue = "";
      String domain = key.substring(key.indexOf("@")+1);
      String user = key.substring(0,key.indexOf("@"));
       if(isInDomain(domain)) {
           strValue = m_plugin.get(key);
           return strValue;
       }
       else {
          m_mcastLayer.clearList();
          m_mcastLayer.discover();
          boolean discover = m_mcastLayer.getFoundOverlays();
          if(discover) {
             int overlayIndex = m_mcastLayer.findOverlay(domain);
             if(overlayIndex != -1) {
               Vector overlayChoice = m_mcastLayer.
                                  getOverlayProfile(overlayIndex);
               String pluginClass =
                              (String)overlayChoice.elementAt(1);
               String bootstrapHost =
                              (String)overlayChoice.elementAt(2);
               String temp = (String)overlayChoice.elementAt(3);
```

```
                temp = temp.trim();
                int bootstrapPort = Integer.parseInt(temp,10);
                try {
                  if(verifyPlugin(pluginClass)) {
                     m_tempPlugin = createPlugin(pluginClass,
                                    bootstrapHost,bootstrapPort);
                     for(int i = 0; i < 10; i++) {
                        strValue = m_tempPlugin.getClientRequest(key);
                        if(strValue != "GET_FAILED")
                          break;
                     }
                     System.out.println("Found "+strValue);
                     m_tempPlugin = null;
                  }
                  else {
                      System.out.println("Incompatible plugin");
                  }
                }catch(PluginNotCreatedException pn_ex) {
                     System.out.println(pn_ex.getMessage());
                }
                return strValue;
              }
               else{
                 System.out.println("Could not find overlay");
                 strValue = user+"@"+domain+":5060";
                 return strValue;
               }
            }
            else {
              System.out.println("Could not find overlay");
              strValue = user+"@"+domain+":5060";
              return strValue;
            }
          }
```

```
        }
        public boolean isInDomain(String domain){
          boolean flag = true;
          String pluginDomain = m_plugin.getDomain();
          if(pluginDomain.compareTo(domain) == 0){
              flag = true;
          }
          else flag = false;
          return flag;
        }
    }
```

Appendix C

Accompanying CD-ROM

The accompanying CD-ROM contains the following:

MosiuoaTsietsi.pdf This thesis in pdf format.

/References Electronic copies of most of the reference material cited in this thesis.

/SourceCode The source code of the standalone OverCord peer-to-peer layer and the modified JAIN SIP Applet Phone that embeds OverCord.