Implementation and Evaluation of the Service Peer Discovery Protocol

DIEGO URDIALES DELGADO

KTH Microelectronics
and Information Technology

Master of Science Thesis
Stockholm, Sweden 2004

IMIT/LCN 2004-06

Implementation and Evaluation of the Service Peer Discovery Protocol

Diego Urdiales Delgado

Master's thesis report
Dept. of Microelectronics and Information Technlogy (IMIT)
Kungliga Tekniska Högskolan, Stockholm
17 May 2004

Abstract

This document is the final report of the master's thesis "Implementation and Evaluation of the Service Peer Discovery Protocol", carried out at the Center for Wireless Systems, KTH, Stockholm. This thesis addresses the problem of service discovery in peer-to-peer mobile networks by implementing and evaluating a previously designed protocol (the Service Peer Discovery Protocol).

The main feature of peer-to-peer networks is that users connected to them can communicate directly with each other, without the necessity of interaction via a central point. However, in order for two network users (or peers) to communicate, they must have a means to locate and address each other, which is in general called a discovery protocol.

There are many different solutions for discovery protocols that work efficiently in fixed or slow-moving networks, but full mobility introduces a set of new difficulties for the discovery of peers and their services. The potential changes in location, which can occur very often, the changes in IP address that these changes may cause, and roaming between networks of different kinds are good examples of these difficulties.

To solve these problems, a new Service Peer Discovery Protocol was designed and a test application built. The next step towards the introduction of this protocol was creating a working implementation, setting up a suitable test environment, performing tests, and evaluating its performance. This evaluation could lead to improvements in the protocol. The aim of this thesis is to implement and document the Service Peer Discovery Protocol, to carry out measurements of it, to evaluate the efficiency of the protocol, and to suggest ways in which it could be improved.

The Service Peer Discovery Protocol was found to be well targeted to wireless, peer-to-peer networks, although improvements in the protocol could make it more time and traffic-efficient while maintaining the same level of performance.

Sammanfattning

Detta är den slutliga rapporten för examensarbetet "Implementation och utvärdering av Service Peer Discovery Protocol", utfört på Center for Wireless Systems, KTH i Stockholm. Uppsatsen behandlar problemet med sökning efter tjänster i icke-hierarkiska (peer-to-peer) mobila nätverk genom att implementera och utvärdera ett redan konstruerat protokoll (Service Peer Discovery Protocol).

Den huvudsakliga fördelen med icke-hierarkiska nätverk är att anslutna användare (parter), kan kommunicera direkt med varandra, utan att behöva interagera med en central punkt. Dock måste metoder för att lokalisera och adressera andra parter vara tillgängliga för att parterna skall kunna kommunicera, metoder som kallas sökprotokoll (discovery protocol).

Det finns många olika sökprotokollösningar som fungerar effektivt i fasta eller långsamma mobila nätverk, men med full mobilitet introduceras ett antal nya svårigheter vid sökande efter parter och tjänster. Den potentiella förändringen av position (vilken kan inträffa ofta), byte av IP-address som dessa förändringar medför, och förflyttning mellan olika typer av nätverk, är exempel på sådana svårigheter.

För att lösa dessa problem, konstruerades protokollet Service Peer Discovery Protocol och en testapplikation byggdes. Nästa steg mot en introducering av detta protokoll var en fungerande implementation, en lämplig testmiljö, utförandet av tester och en utvärdering av prestandan. Utvärderingen syftade till att förbättra protokollet. Syftet med detta examensarbete är att implementera och dokumentera protokollet Service Peer Discovery Protocol, att göra mätningar, att utvärdera effektiviteten samt att föreslå förbättringar av protokollet.

Service Peer Discovery Protocol fanns vara väl anpassat till icke-hierarkiska trådlösa nätverk. Dock torde förbättringar av protokollet innebära tidseffektivare och trafikeffektivare beteende utan att kompromissa prestandanivån.

Acknowledgements

I would like to thank my examiner, Prof. Gerald Maguire, and my supervisor, Andreas Wennlund, for their priceless dedication, help and patience with me. Thanks, Professor, for prioritizing your students even over your sleep…

Some fellow students or researchers have also been of great help: Roberto Cascella, who helped me understand SPDP from the beginning; Kostas Avegeropoulos, to whom I owe so much for his assistance, fruitful comments, and company; Asim Jarrar, who provided me with the *key* to *unlock* the protocol.

Without my friends, who support and listen and amuse and comfort, I would not be able to do *anything* in life, let alone this master's thesis. Thank you Miguel, Amer, Agustín, Antonio, Joserra, Sara, María José..... and, fortunately, many others.

Por ultimo, quiero dar las gracias a mi familia: a mi padre, a mi hermano, y a mi querida mamá, por quererme tanto, tanto, tanto. A vosotros os dedico especialmente este pequeño fruto de mi esfuerzo.

Table of contents

List of figures

List of tables

1 Introduction

1.1 Motivation

Wireless networks are becoming more and more important in our life. The major advantage that mobility and the absence of cables provide for network users has gradually become more and more evident as the quality gap in terms of bandwidth, delay, and other capabilities between wired networks and wireless technologies has decreased.

The near ubiquity of such networks, which have various underlying technologies such as General Packet Radio Service (GPRS) [1] or Wireless Local Area Network (WLAN) [2], has made it possible and **necessary** to study a scenario of wireless and mobile users who request connectivity from their operators, but are otherwise independent peers in the network. In this scenario of overlay peer-to-peer networks [3], as long as a user can get connectivity, he/she can and will become a peer and thus will be able to **request** or **provide** services, such as file sharing, voice/text/video conversations, etc.

One of the first issues that arise when designing wireless peer-to-peer networks is the way in which the different peers and services in the network are discovered, considering that they can change their location and availability and that bandwidth and battery power are often precious resources for mobile devices. In his master's thesis "Reconfigurable application networks through peer discovery and handovers" [4], Roberto Cascella proposes a discovery protocol suitable for these kinds of networks: the Service Peer Discovery Protocol (SPDP).

SPDP is an extension to the Session Initiation Protocol (SIP) [5]. It provides network peers with a means to discover other peers and request services from them, in a way that is independent of the underlying communication technology. Furthermore, it defines a common scheme to describe services and it supports unique identification of all peers by giving each of them a name (the SIP Uniform Resource Identifier, i.e. SIP URI) which does not change as the peer changes location or as a user changes the device with which he/she connects to the network.

Once the specification of the protocol was ready, a working implementation needed to be written to test the protocol. This master's thesis aims to design and document *an* implementation of the Service Peer Discovery Protocol, to set up a test environment to analyze and measure the efficiency of the proposed protocol, as well as to make suggestions for improvements.

This thesis was carried out in close relation to the Adaptive and Context-Aware Services project [6] at the Center for Wireless Systems, Royal Institute of Technology (KTH).

1.2 Goals

This thesis has tried to accomplish the following goals:

- To analyze previous work in the fields of service discovery protocols, SIP and the Service Peer Discovery Protocol.
- To create a working implementation of the Service Peer Discovery Protocol (SPDP) based on the protocol specification [4] and the sample application built by Roberto Cascella. The focus was on creating an illustrative and re-usable working implementation.
- To evaluate the performance of the Service Peer Discovery Protocol in comparison with other available discovery protocols.
- To suggest ways in which the Service Peer Discovery Protocol could be improved or extended.

The first goal is realized in the background of the thesis report (chapter 2). The implementation created is described in chapter 3. The results and conclusions of the evaluation are contained in chapter 4. Finally, the suggested changes and improvements are detailed in chapter 5.

1.3 Outline of the document

After this **introduction**, where the motivation and the goals of this thesis are explained, comes the **background** section, which addresses a number of topics that are closely related with the subject of the thesis. First, the scenario for which the SPDP protocol was designed is described and analyzed. Since SPDP is to be used in a wireless, peer-to-peer networking environment, the basics of such an environment are described. Then, some strategies for service discovery that are already in place in fixed networks are described, as an introduction to the proposed SPDP protocol. The functionality and operation of SIP, the session establishment protocol upon which SPDP is built, is examined in the third section. Finally, the Service Peer Discovery Protocol itself is explained in the last section of the background.

In the third chapter, the proposed **implementation** of SPDP is thoroughly described and documented. Both the external interface to the implementation – the requirements of the program, its interactions with the Session Initiation Protocol, and the practical information to actually use the software, and its internal operation are described in detail. Internally, the focus is on the data structures used by the protocol and the thread model that governs its operation.

Chapter 4 contains the **evaluation** of the protocol. In order to evaluate the protocol, two approaches were taken. Firstly, some simple test cases were designed in order to perform some comparative measurements of SPDP and

another discovery protocol, the Service Location Protocol (SLP) [7]. Then, the suitability of SPDP for more realistic, complex scenarios was examined from a theoretical point of view. The outcome of the evaluation is summarized in some conclusions included in this chapter.

Both evaluation stages led to several proposals for **improvements** to the protocol that are suggested as future work. Those are detailed in the fifth and last chapter.

2 Background

2.1 Peer-to-peer wireless scenario

Peer-to-peer is a communications model in which all parties are equal and can initiate communication with each other [8]. This is what distinguishes the peer-to-peer model from the client/server model, where it is up to clients to begin communication sessions whereas servers simply wait for requests. In a peer-to-peer network, each node (called a peer) can act as a client and a server at the same time. Furthermore, in the peer-to-peer model, peers interact directly with each other, as opposed to a centralized model where all communication goes through a central point.

Surely the most successful and widespread peer-to-peer application so far has been file sharing. Napster [9], Gnutella [10], and Kazaa [11] are some of the most famous peer-to-peer networking applications, which allow users to access each other's files. Such has been their success that, although peer-to-peer refers to a model where many different kinds of applications could fit, its name is nowadays synonymous with what should be more properly called peer-to-peer file sharing.

The peer-to-peer scenario that this thesis refers to is **not** restricted to file sharing or any other particular application. The protocol under study, SPDP, provides generally applicable peer and service discovery mechanisms. Although one of the discovered services could be access to files, this need not be the case.

The main features of this scenario are that it is peer-to-peer and wireless. The implications of a **wireless** scenario are many. Firstly, wireless access implies that user mobility can be *fast* compared to the session duration and *frequent* compared to the service inter-request time; here, "fast" means that a peer may move (roam) several times during a single communication session, and "frequent" means that there is a high probability that a peer is not found in the same location as it was when the previous request was initiated. Most of the ordinary peer-to-peer discovery protocols support *infrequent* (as opposed to frequent) mobility by providing a set of **core peer addresses** to connect to in case all the previously learned local addresses are unavailable or unreachable. However, this can become very inefficient if a peer has to resort to that core very often.

Secondly, in a wireless network there are typically also fixed nodes (such as base stations or access points) that serve a particular area or cell. This is to say that the scenario under study consists both of highly mobile peers which potentially change their position rapidly and completely fixed peers that are very likely to always be found in the same location (or otherwise not be found at all in case of temporary or permanent unavailability).

Apart from being mobile or fixed, peers can be characterized by the **services** they offer. Every peer can offer many different kinds of services;

examples are file sharing, voice calls, printing, connectivity, context information,…

From a networking point of view, all peers will be considered to get network **connectivity** from their network providers. This means that the scenario under study consists of peers whose operators allow them to connect to the network, and once they are connected they can interact with the rest of the peers. This can fit for instance in an overlay peer-to-peer network scenario such as the one described in [3]. However, SPDP [4] is also designed to work in networks without such infrastructure; here we assume that the peers act as their own network operators.

As far as the rest of the protocol stack is concerned, all peers are considered to utilize Mobile IP [12]. This provides the peers with mobility at the IP level while they keep the same home address. Peers can make use of different underlying technologies to connect to the network, specifically GPRS and WLAN in this scenario. It will be assumed that WLAN availability is restricted to a number of "hot spots" whereas GPRS is expected to provide total coverage. A device can have interfaces connected to both kinds of networks (possibly at the same time, since these networks can overlap) and thus the device can have multiple IP addresses.

Finally, devices can be behind Network Address Translation (NAT) boxes, which implies that they will not have a globally routable IP address, i.e. their addresses will not be reachable from outside this NAT domain. As proposed in [4], this difficulty can be solved by using a particular proxy configuration depicted in Figure 1, which will be assumed in the network under study.

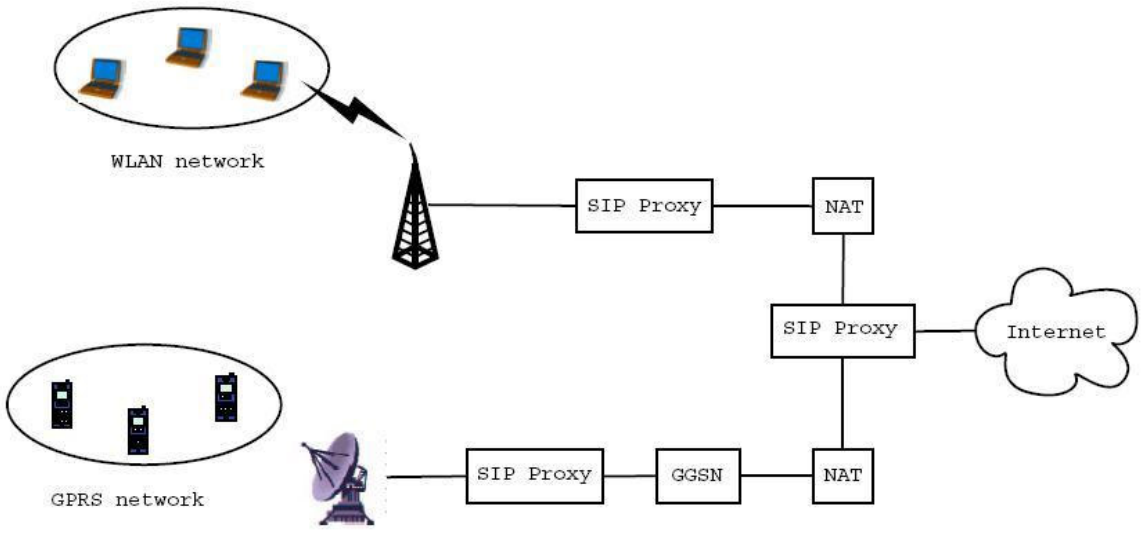

Figure 1: GPRS and WLAN networks for SIP communication

2.2 Service discovery in peer-to-peer networks

2.2.1 Introduction

Roberto Cascella presented in his thesis [4] an overview of the different strategies for service discovery in peer-to-peer networks. In a peer-to-peer network, users interact *directly* with each other to perform a service or exchange data. In order for a peer to know about other peers in the network, several possibilities exist:

- **Static configuration**: Every peer needs to know the other peers in advance. Presence information (information about the present availability of a given peer) may also be needed.

  The advantage of this strategy is security: if all peers are known, there is no risk of communicating with unknown, untrusted peers (although a peer can be impersonated and other security threats remain).

  The disadvantage is scalability, because the introduction of a new peer requires that all peers update their information (manually), and thus the peer location databases grow linearly with the total number of *potential* peers. Another disadvantage with this strategy is that there is a tradeoff between accurate presence information and signalling network traffic.

- **Centralized model**: A central server, possibly replicated for robustness, knows about all existing peers. Peers register with it and request services from it. This server then redirects them to a suitable destination peer.

  This model is more scalable, since a peer only needs to register to be able to *reach* and *be reached by* the rest of the peers. However, this scalability is limited by the capacity of this central server and the network links to it.

  The main drawbacks are efficiency and flexibility, since it is easy for the central server or its replicas to become congested. It also has low reliability, since the central server takes part in every service request.

- **Decentralized model**: In this model, there is no central control point and no single node knows the whole network topology. This model can be realized as a virtual network built on top of the IP network, where each user has an id which is temporarily mapped to an IP-address.

  The **Gnutella** protocol is a version of such a model (see section 2.2.2.2), where a peer connects to the network and notifies to possible neighbours, which are the neighbours known from manual configuration or from previous sessions. When a peer needs a

service, it asks its neighbours for it, and if they cannot provide the service themselves, they will in turn forward the request to their neighbours. When the peer finally communicates with the service providing peer, the two peers become logical neighbours of each other. This model has the advantage that there is no need to rely on central servers, nor is there dependence on a single node's load. Additionally, a single point of failure is avoided. However, a disadvantage is that logic neighbours need not be physical or topological neighbours, and hence communication cost may be higher than necessary.

Another version is the **multicast model**, where requests for services and presence notifications are sent to a well-known multicast address. This eliminates the need to know how many listeners are currently present, and most importantly avoids other peers having to route or forward requests that are not intended for them. Additionally, it identifies *topological* neighbours. However, multicast is often only supported in a shared link and has little widespread support which makes this model difficult to implement. In the scenario studied, GPRS has no multicast capability.

There are several protocols and architectures that use one of the above models to provide service and peer discovery in peer-to-peer networks. Some of these will be described next.

2.2.2 Service discovery in peer-to-peer file sharing applications

In file sharing peer-to-peer applications, service discovery could be considered as assimilated into the discovery of a particular file, and every shared file can be considered an offered service. Three of the most widely known peer-to-peer file sharing applications are Kazaa, Gnutella, and Napster.

2.2.2.1 The Kazaa protocol

Kazaa [11] is one of the most famous peer-to-peer file sharing applications that exist nowadays. A user who connects to the Kazaa network today is likely to join more than 3 million other users, sharing files that total more than 4.5 million gigabytes. This means that the Kazaa protocol has successfully solved the problem of scalability.

The Kazaa protocol is secret and very little is thus known about its details. However, its architecture is better known. The Kazaa protocol contemplates three levels in the peer hierarchy [13]:

- Normal users
- *Supernodes*, which are users that have all the service information of a particular zone, that is, they know all the available files of the users subscribed to them.

- Central servers, whose addresses are built into the software and are static global supernodes to which a user can always resort.

When a user connects to the Kazaa network, it uploads the list of the files it wants to share to its assigned supernode (which it can know for example from previous sessions) [14]. In this way, the supernode knows about all the available files in a particular zone (i.e. for a subset of users).

Whenever a user wishes to search for a file, it requests its supernode to search for this file. The supernode first performs the search locally within the subset of users that it controls and returns the results to the user. The user can then ask the supernode to seek more sources for the file, which results in a new search performed within another supernode's users in a rather transparent way for the user, since it is the local supernode that initiates the search, either iteratively or recursively.

This scheme clearly favours **locally available** content, which often yields the highest downloading speeds. Besides, since the number of supernodes is large, because many normal users act as supernodes more or less unawares, their **load** can be kept very low and hence the risk of overload is small. If the local hit ratio can be kept high (a condition that is favoured by users sharing a very large number of files and by the fact that most of the searches aim at a small subset of "popular" files), the model is very **scalable** because of the very local scope of the searches.

There are however some tradeoffs in this model: first, if the file that is being searched for is not very common, **many consecutive searches** will be needed in order to find a source for it. Although there is an extremely high number of users and file instances, most users ask for the same files. Secondly, the assumption that local connections will provide the **highest downloading speeds** is not always true, and this causes some high-speed users to be trapped in slow-speed environments. In order to solve this, tools are available in the Internet (such as KaZuperNodes [15]) that allow a user to specify **which supernodes to connect to**. A high-speed user can then connect via his/her high-speed environment to other high-speed environments, but of course the local-favouring scheme is lost (this has implications because of the geographical distribution of the files: a high-speed user can elect to download "generally popular" files from users in the high-speed environment, but may have to perform many consecutive searches or eventually even resort to the local low-speed environment to find "locally popular" files).

2.2.2.2 The Gnutella protocol

Gnutella [10] is another very popular protocol for peer-to-peer search. Unlike Kazaa, the Gnutella protocol [16] is open and its specification is available via the Internet.

Peer discovery in Gnutella is performed through so-called Ping and Pong messages. A peer connected to the network will periodically send Ping messages to probe the network for newly connected peers. In response to a Ping, a peer will send a Pong message with its address and possibly some additional cached information about the addresses of other known peers in the network. However, the protocol does not specify how a peer should learn another peer's address in order to connect to the network.

As far as **service discovery** is concerned, the availability of a source for a certain file can be probed by sending a Query message for that file. This message contains some file search criteria as well as a minimum speed for the peer. A Query message will only generate a response (with a QueryHit message) if a peer offers a source matching the criteria and has a speed greater than the specified minimum.

As can be seen, the Gnutella architecture is purely peer-to-peer since there is no hierarchy among the peers. Every peer acts on its own behalf in the network. This gives increased **robustness** and **fault-tolerance** to the network. **Local content** is preferred to remote content, since TimeToLive (TTL) fields are included in both Ping and Query messages to limit their scope.

Gnutella expects every peer to be able to **route** messages appropriately. The routing scheme is an important part of the protocol, since every peer needs to forward messages in the right way in order for peer and service discovery to work.

The Gnutella protocol works on top of TCP and in cooperation with HTTP, which is the protocol used for file downloading once a file instance has been found and selected.

2.2.2.3 The Napster protocol

Napster [9] was the pioneering application in peer-to-peer file sharing. It first appeared as an MP3 music file sharing application in 1999 and was banned by law a few months later.

A Napster protocol specification can be found in [17], and although it is unclear, it does give important information about the protocol architecture. The Napster protocol relies on servers to set up communication between clients, perform searches, and keep track of client status.

A client connects to a server and asks the server for the title and author of the requested song. The server then performs a search and presents the results to the client. When the client selects the source he/she wants to download from, he/she notifies the server, which will connect him to the other client to start their direct communication. For each connected client, the server keeps track of the current number of uploads and downloads. This status is kept coherent as clients also notify the server when an upload/download is finished.

Although this architecture seems suited for a single, omniscient server to control the whole network (since nothing is said about inter-server communication or server hierarchy), this approach has poor scalability and has the possibility of a single point of failure. As can be seen, the Napster protocol is markedly centralized in terms of service discovery, and hence needs no peer discovery mechanism since every peer has to register with a (well-known) server. This centralized model also made it easy to shut down Napster since the servers were well known.

2.2.3 SLP – The Service Location Protocol

The Service Location Protocol [7], developed by the IETF [18] and specified in RFC 2608, provides a scalable framework for service discovery and selection in a network.

The SLP architecture [7, 19] specifies three different types of entities:

- *User Agents* (UA), which perform service discovery on behalf of the client (user or application).
- *Service Agents* (SA), which advertise the location and attributes of services.
- *Directory Agents* (DA), which collect service advertisements and store service information.

According to the protocol specification [7], DAs are optional, and their function is to provide scalability to the protocol in larger networks. If no DA is present, UAs wanting to discover a particular service multicast a Service Request to a well-known SA multicast address. A SA advertising a service which matches the requirements in the request will then unicast a Service Reply to the UA. The message exchange is depicted in Figure 2.

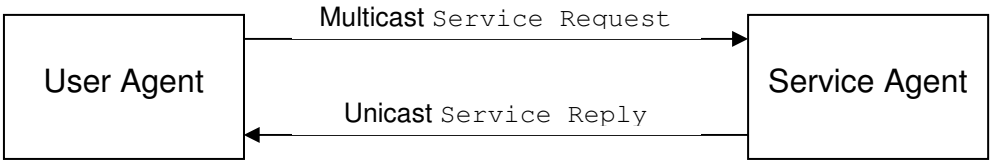

Figure 2: Messages exchanged between an SLP User Agent and Service Agent

If a DA is present, it functions as a cache. SAs send Service Registration messages containing all the services they advertise to the DA, and UAs send unicast Service Requests to the DA. The message exchange is depicted in Figure 3.

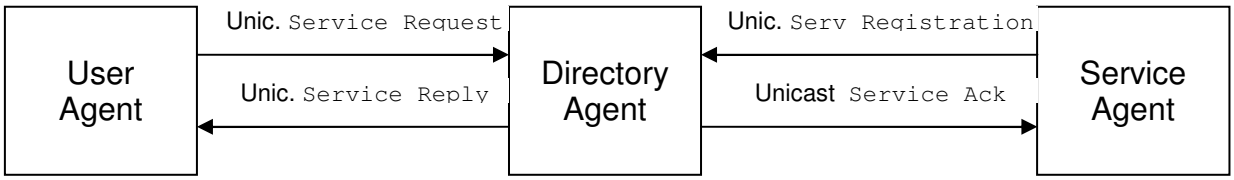

Figure 3: SLP Directory Agent messages

The discovery of the DA can be done in three different ways, static, active or passive [19]:

- With *static* discovery, User Agents and Service Agents obtain the address of the Directory Agent through DHCP (Dynamic Host Configuration Protocol [20]). The DHCP options for SLP are defined in [21].
- With *active* discovery, UAs and SAs issue a multicast Service Request for the "Directory Agent" service, to which the DA replies with a DA Advertisement.
- With *passive* discovery, the DA sends an unsolicited DA Advertisement infrequently (the protocol specification suggests one advertisement every 3 hours), which SAs and UAs listen for. The message exchange for active and passive discovery is depicted in Figure 4.

Figure 4: SLP message exchange for active (top) and passive (bottom) discovery

Services are advertised in a standard manner using a Service URL and some attributes. For instance, a service associated with a printer may look like this:

```
service:printer://ld123.burgos.com:1020/queue1
scopes = x, y, z
printer-name = ld123
```

```
printer-model = Epson 700
printer-location = Room A21
color-supported = true
pages-per-minute = 18
sides-supported = one-sided
```

An important feature of the Service Location Protocol is that it is intended to work within networks under cooperative administrative control, which permits a policy to be implemented regarding security, multicast routing and grouping of services and clients [7]. This makes it unfeasible for SLP to be applied to the Internet as a whole. Furthermore, due to its architecture, SLP may not necessarily scale for wide-area service discovery or for networks with hundreds of thousands of clients or tens of thousands of services. In the scenario studied, this means that either the DA must be inside the NAT subnet or the UA must know the address via static configuration (i.e. DHCP) or do active discovery.

2.2.4 Jini

Jini [22] is the extension to the Java [23] programming language which deals with how devices connect with each other in order to form a so-called *Jini community*, and how services are provided inside this community [4, 19].

As far as the architecture is concerned, Jini utilizes the same basic principles as SLP. Jini uses a so-called *Lookup Service* as a directory service that maintains the information about the available services in a community. At least one copy of the Lookup Service needs to be present in the network [24]. A client can discover a community using the *Discovery* protocol, and then join the community using the *Join* protocol. After a client has joined a community, it can access the services contained in the community's Lookup Service. Communication between entities is accomplished using Java RMI (Remote Method Invocation).

The main feature that distinguishes Jini from SLP is that the Lookup Service can contain both pointers to services and Java-based program code for those services, which can be drivers, an interface, or other programs that help the user access the service. This eliminates the need to pre-install drivers for a service in the client.

However, Jini places the requirement that every client should have memory and computational power to run a JVM (Java Virtual Machine), in order to be part of a community and access or advertise services.

2.2.5 UPnP – Universal Plug and Play

Universal Plug and Play [25] is the architecture for peer-to-peer network connectivity developed by Microsoft [26]. The UPnP architecture defines three different entities [4, 24]:

- *Devices*: a device must have an IP address to be located and addressed. A device may consist of one or more services plus other embedded devices. Those services and embedded devices are specified in the device's XML description.

- *Services*: a service is the smallest unit of control. It is specified in an XML description. Changes in service state and invocations of a service may trigger events.

- *Control points*: they are responsible for discovering new devices in a local network, learning about device and service changes, and controlling devices.

UPnP introduces several protocols that are used to accomplish the different functions in the architecture:

- The *Simple Service Discovery Protocol* (SSDP) [27] deals with device and service discovery. It allows for both announcement messages (i.e. a device indicates its presence in the network to the control points by sending a NOTIFY message to a well-known local multicast address) and discovery messages (i.e. a control point tries to discover new devices or services by multicasting an M_SEARCH message to a well-known local multicast address).

- The *General Event Notification Architecture* (GENA) [28] provides a standard architecture for event subscription and notification. It uses HTTP over TCP and multicast UDP.

- The *Simple Object Access Protocol* (SOAP) [29] provides a standard architecture for control message exchange.

A device's unique identification in an UPnP network is its IP address. UPnP requires all devices to be capable of self-configuring their IP addresses using AutoIP [30] in case no DHCP server is present in the device's local network. In case of dynamic networks, where devices may change their IP addresses, a host name can also be associated with each device. The current IP address of the device can be obtained from the host name using DNS. For small networks without a DNS server, UPnP defines Multicast DNS, which uses link-local multicast messages for the queries.

2.2.6 The JXTA Peer Discovery Protocol (PDP)

2.2.6.1 JXTA

JXTA [31] is an acronym for "juxtaposed". It is a set of protocols whose aim is to offer standardized blocks that can simplify the creation of overlay peer-to-peer networks [3]. JXTA provides a generic overlay network that any application can use to provide peer-to-peer networking.

A JXTA network is composed of three types of entities:

- **Peers** are the endpoints of the network; they can discover and communicate with each other

- **Relays** can represent peers in the network and receive messages on their behalf. Relays help peers that are behind NAT/NAPT boxes connect to the JXTA network. The relay receives and buffers the messages intended for the peer; the peer periodically polls the relay and retrieves the messages that were intended for itself. The peer uses HTTP to poll the relay.

- **Rendezvous** aggregate information about peers and services in an area.

The JXTA architecture is thus semi-distributed: there is no central point (unlike a centralized network like Napster) but not every peer has routing and naming responsibilities (such as in a fully distributed architecture like Gnutella). This approach is intended to benefit from avoiding a single point of failure while keeping the load on the peers low (by relieving the peers of routing and naming).

JXTA is group-oriented, meaning that services are offered and accessed in the context of a group. Upon initiation, a JXTA platform tries to join a super-group called the *world peer group*. Inside a group, a peer can communicate in both unicast or multicast fashion with other peers in the group. The Peer Membership Protocol (PMP) allows a user to create secure groups by adding a digital signature or certificate to the group advertisement. The rendezvous responsible for group membership verifies that a peer wishing to join the secure group presents a matching signature or certificate.

2.2.6.2 The Peer Discovery Protocol (PDP)

One of the seven protocols defined in the JXTA platform is the Peer Discovery Protocol (PDP) [32], which is used to discover any published peer resource. As JXTA is group-oriented, PDP allows a peer to discover resources within its group. Notably, PDP is the discovery protocol of the *world peer group*.

PDP uses Discovery Query messages to search for resources and Discovery Response messages to advertise resources. The information units exchanged in PDP are JXTA advertisements, which are descriptions of resources, whether these are peers, peer groups, pipes, modules, or other kinds of resources. Both Queries and Responses are formatted in XML.

A Discovery Query contains Attribute-Value pairs that specify the resource that is being searched for, as well as a Threshold parameter which sets the maximum number of advertisements that each of the responding peers should provide in the Response. It can also contain the advertisement of the requestor (in order to favour peer information spreading in the network).

A Discovery Response contains zero to Threshold advertisements for resources matching the conditions of the Query, as well as possibly the advertisement for the respondent. Every advertisement has an Expiration attribute.

A key feature of PDP is that it does not provide a reliable service: a peer that receives a Query is not required to respond; furthermore, a reliable transport protocol is optional in PDP, resulting in zero, one, several or redundant Responses arriving in response to a single Query. Finally, a peer may receive a Response without having issued a Query for it, and should interpret this as a publication of the advertisements contained in it.

A JXTA peer is configured to attempt to join the *world peer group* upon start-up. Initially, the peer will issue a multicast Query in the local network in order to find other world group peers or rendezvous. If there is no response, then the peer will then try to contact known rendezvous following a predefined list.

2.3 SIP – The Session Initiation Protocol

2.3.1 Introduction

The Session Initiation Protocol SIP [5] is a widely used protocol for session establishment. SIP is an application-layer control (signalling) protocol for creating, modifying, and terminating sessions with one or more participants. Such sessions can be used for any communication or data exchange, such as voice conversations, conferencing, streaming, file sharing,…

2.3.2 Functionality of SIP

SIP provides five main functions related to session management:

- User location: finding the user that is to be contacted.
- User availability: finding out the willingness of the called party to start communication.
- User capabilities: finding out the communication capabilities of the called user.
- Session setup: "ringing" and establishment of the session parameters.
- Session management: transfer, termination, and modification of the session, as well as invoking services.

SIP places no constraints on what the established session will be used for. It works with other protocols such as RTP (Real-Time Protocol), RTSP (Real-

Time Streaming Protocol), MEGACO (Media Gateway Control Protocol, for interaction with the public switched telephone network), or SDP (Session Description Protocol). It controls the session but **not** the media that is exchanged during the session.

SIP does not directly provide services, but rather provides primitives that can be used to implement a variety of services. SIP provides end-to-end security and is compatible with both IPv4 and IPv6.

2.3.3 Operation of SIP

To study the basic operation of SIP, an example of a session establishment between Sara and Diego will be examined. Sara wants to have a voice conversation with Diego through the Internet.

Each user is identified by a so-called SIP URI (Uniform Resource Identifier). Sara's SIP URI is *sip:sara@burgos.com* and Diego's SIP URI is *sip:diego@stockholm.com*. There is also the possibility of using SIPS URIs, which provide secure communications of the SIP messages using TLS.

2.3.3.1 SIP trapezoid

The basic operation of SIP is represented in the so-called SIP trapezoid and the message flow depicted in Figure 5:

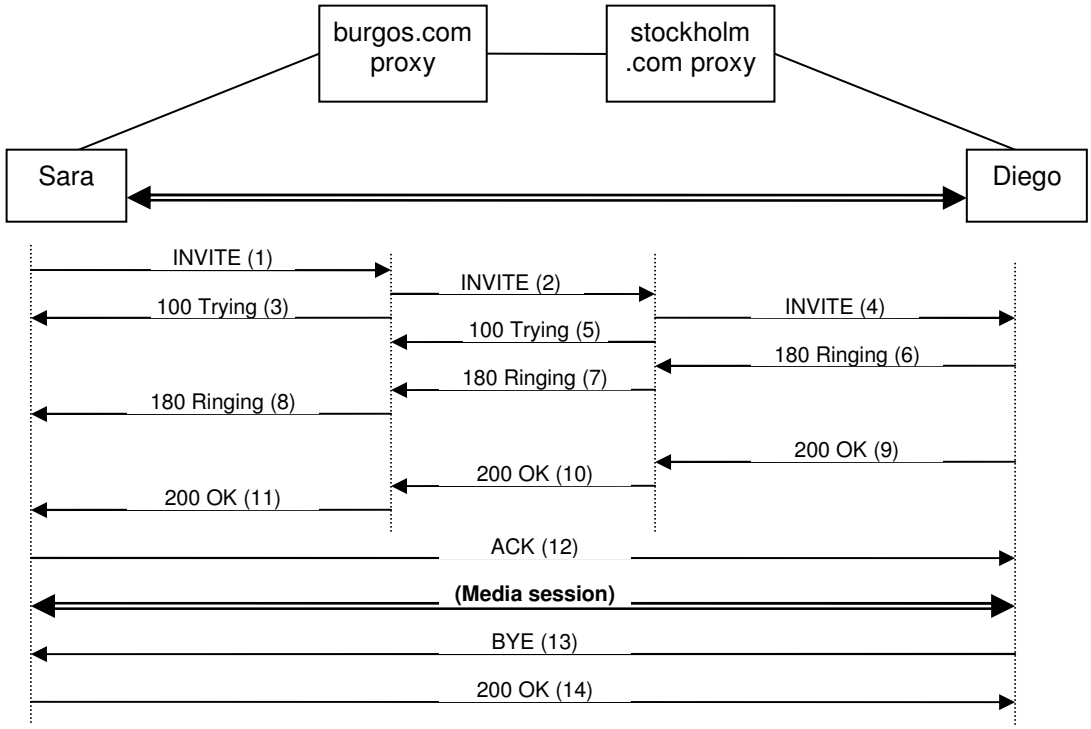

Figure 5: SIP trapezoid and message flow

Requests are usually specified by their request method (INVITE, ACK, BYE), and responses by a three-digit code plus a string (such as "180 Ringing").

2.3.3.2 SIP message flow

The message flow is as follows:

1. If Sara's terminal does not yet know the machine at which Diego is sitting right now, she issues an INVITE for Diego's address-of-record (public address) towards her domain's proxy, whose address she knows, for example, through manual configuration.

2. The proxy at Sara's domain (burgos.com) performs a DNS search in order to find the proxy at Diego's domain (stockholm.com), it adds its address in a **Via** header of the request, and then forwards the INVITE message.

3. The burgos.com proxy then sends Sara the temporary response "100 Trying".

4. The proxy at stockholm.com looks for Diego's currently registered location in its location service and forwards the INVITE to him, after adding its address in another **Via** header.

5. Then, the proxy at stockholm.com sends the corresponding temporary "100 Trying" response to the burgos.com proxy.

6. Upon receiving the INVITE, Diego's machine sends the temporary response "180 Ringing" back to its proxy; this indicates that Diego's voice terminal is ringing.

7. The "180 Ringing" response is propagated back to the burgos.com proxy.

8. And finally Sara's terminal receives the temporary 180 response thanks to the fact that both proxies had added their addresses to the request.

9. When Diego decides to answer using his voice terminal, a final "200 OK" response is generated.

10. It too is propagated back via both proxies.

11. And finally delivered to Sara's terminal.

12. When the "200 OK" response reaches Sara, her terminal now knows Diego's actual location and sends an ACK message directly to him. The proxies no longer take part in the subsequent communication, unless otherwise specified. Now the media session can start. Usually, both parties would have used the message bodies of their INVITE

(respectively "200 OK") messages to exchange the session parameters using, for instance, the Session Description Protocol (SDP). Furthermore, the session parameters can be changed during the media session by any of the parties by issuing another INVITE message with the new parameters. Should the other party agree, the changes would be made. Should he/she not agree, the session continues with the original set of parameters.

13. Diego decides to hang up and ends his communication. This results in a BYE message being sent to Sara.

14. Sara replies with a "200 OK" final response.

2.3.3.3 SIP message header fields

As an example of a SIP message, the first INVITE that Sara's terminal issues towards the proxy at burgos.com will be analyzed. It contains the minimum set of required header fields:

```
INVITE sip:diego@stockholm.com SIP/2.0
Via: sip/2.0/UDP pc33@burgos.com;branch=z9hG4bk776asdhds
Max-Forwards: 70
To: Diego <sip:diego@stockholm.com>
From: Sara <sip:sara@burgos.com>;tag=1928301774
Call-ID: a84b4c76@pc33.burgos.com
CSeq: 314159 INVITE
Contact: <sip:sara@pc33.burgos.com>
Content-Type: application/sdp
Content-Length: 142

(Content not shown)
```

The header mainly specifies the request method (INVITE). The header fields can be explained as follows:

Via: The address at which Sara expects responses to THIS request. The branch parameter identifies the transaction and is used by proxies to avoid loops. Each proxy adds a further **Via** header field before the first to ensure that they are informed of the whole session establishment process.

To: The name and SIP URI to which the request is directed.

From: The name and SIP URI of the originator of the request. The tag parameter is for identification purposes.

Call-ID: A globally unique identifier for this call, consisting of a random string plus the originating terminal's host name (or IP address).

CSeq: It is a sequence number, plus the method it refers to.

Contact: A SIP URI where the calling user can be contacted directly, intended for future requests, not for this one.

Max-Forwards: The maximum hop count to destination.

Content-Type: The description of the message body, in this case an SDP session description.

Content-Length: The byte count of the message body.

2.3.3.4 Proxy functionality

In SIP, proxies perform some special functions, which are:

- Location service look-ups to find the current mapping of URIs in its own domain.
- DNS look-ups to find the proxy of a specific domain.
- Flexible routing decisions, such as routing a call to a voicemail server if the voice terminal replied with a "486 Busy here" response.
- Forking invitations, that is, trying to find a user by sending invitations to several terminals at the same time.
- Being part of the whole SIP session, which a proxy does by inserting its address in a **Record-Route** header field. This can be done for instance when the proxy needs to perform some kind of third-party intervention in the session.

2.3.3.5 SIP methods

The example above features three of the methods in SIP (requests). There are six methods in SIP, but there is a possibility to create more through extensions:

INVITE: used to start a SIP session or to change parameters in the middle of a session (it is then called a re-INVITE).

CANCEL: used to cancel a pending invitation.

ACK: used to acknowledge, for instance, a session establishment.

BYE: used to terminate a SIP session.

OPTIONS: used to query the capabilities of a SIP entity.

REGISTER: a user agent sends a registration message periodically to a machine in its domain (called the registrar) in order to inform it of its current location. The registrar then updates the location service. This is one of the ways to populate the location service

database. Usually, the domain proxy and the registrar are located on the same machine.

2.3.4 SIP mobility support

SIP has some features that support mobility of devices, users, and sessions. However, there are important considerations regarding the different implications of supporting mobility at the application layer (as SIP does) or relying on the network layer for mobility support (using Mobile IP). Please refer to [4] for those considerations.

2.3.4.1 Device mobility

SIP supports both so-called pre-call mobility (i.e., a user moves between two service sessions) and mid-call mobility (i.e., a user moves during a service session).

If a device moves between two service sessions and has the time to register its new location with the home registrar, new sessions will be initiated correctly and the change in address will be transparent to the calling device, thanks to the fact that the roaming device always keeps its address-of-record. However, if the registration had not been completed, the request will be directed to the device's most recent location. In order to solve this, Schulzrinne and Wedlund [33, 34] propose a solution where the proxy of the most recent registration sends a scoped multicast message to try to find the device. If the device can still not be found, the proxy reports a failure to the caller. This solution works well if the device moves to a nearby network, and in case of failure provides the device with some time to complete the registration while keeping the caller informed.

In case of mid-call mobility, the moving device should send the corresponding party a reINVITE request specifying the new location (in a Contact header). In order to accelerate this process and avoid the need to send a reINVITE all the way back to the corresponding party, the session could be directed from the beginning through a proxy in the moving device's domain, and when the movement occurs, the device will only need to notify that proxy. The proxy will then deliver the session media to the appropriate location. This assumes that the new location is close to the old location.

Another solution takes advantage of the conferencing feature of SIP. A session between two (or more) parties could be established via a Multipoint Control Unit (MCU). This way, when one of the parties moves to a different location, it will join the conference from the new location. No re-routing is needed in the corresponding node, but resources are wasted in the MCU.

2.3.4.2 Personal mobility

SIP allows a user to connect to the network through several different devices while keeping a single identifier. A request for the user will be directed to all the devices thanks to the forking feature of SIP.

2.3.4.3 Session mobility

SIP allows a user to change terminal while maintaining an ongoing session. A user can negotiate new session parameters, including a new terminal, by sending a reINVITE message. This way, the session will be completely redirected to the new terminal.

Furthermore, if the session is initiated and negotiated by a third party, the third party control allows a user to redirect the session to different terminals or even to split the session into different terminals according to the media types. However, this approach makes it necessary to contact the third party for changes or termination of the session.

2.3.5 SIP extension support

SIP was designed to be extended. SIP extensions define additional request methods apart from INVITE, CANCEL, ACK, OPTIONS, BYE, and REGISTER. The introduction of extensions to SIP is facilitated by the existence of the header fields **Require**, **Proxy-Require**, and **Supported**, among others. All these header fields refer to extensions defined in standards-track RFCs, never to vendor-specific extensions. In this way, the protocol is kept vendor-independent.

Not every intermediate node in a SIP session needs to support an extension for the message exchange to be successful; no element may refuse to proxy a request because it contains a method or header field it does not know.

The **Require**, **Proxy-Require**, and **Supported** header fields are used as follows:

- If a user agent client (UAC) requires that the user agent server (UAS) understand an extension that it will use in its request, it will include this extension in a **Require** header field. If intermediate proxies also need to understand the extension, the UAC will include it in a **Proxy-Require** header as well.

- If a user agent client supports extensions that can be used by the server in a response, the UAC should include a **Supported** header for those extensions in the request.

- A UAS cannot apply an extension that was not in the **Supported** header field of the request. In the rare case that this extension is essential, it should reply with a "421 Extension required", but this is not recommended.

- All extensions used in a UAS response have to be in the **Require** header field.

2.3.6 The SIP Specific Event Notification extension

One of the extensions to SIP defined in RFCs is the **Specific Event Notification extension** [35], whose functionality and operation will be summarized here due to its close relationship with the subject of this thesis. In fact, the Service Peer Discovery Protocol functionality is based on this SIP extension.

This extension allows the SIP nodes which implement it to request and give notification that an (asynchronous) event has occurred. The node which is interested in the event must **subscribe** at the corresponding node, which will in turn **notify** the subscribed node(s) whenever one of the specified events happens.

The Specific Event Notification extension does not specify the actual events that should trigger notifications and subscriptions. Those events are specified in so-called **Event Packages**.

2.3.6.1 Operation of the extension

When a node is interested in subscribing to a particular event or change in the state of another node, it sends a SIP request with **SUBSCRIBE** as a method to that node.

This subscription message must contain an **Expires** header which defines the duration of the subscription. The duration can be shortened by the notifying node, but never lengthened. However, if the duration is too small and will presumably lead to congestion in the notifying node because of re-subscriptions or for other reasons, the notifier can reply with a "423 Interval too small" error. In any case, it is important to note that every subscription is limited in time and will expire unless periodically renewed.

There are special semantics associated with a SUBSCRIBE request with an **Expires** value of 0. This causes the subscription to be automatically finished, i.e. it is an **un**subscribe message.

The SUBSCRIBE message must contain an **Event** header to specify the event or set of events that the subscriber wants to be notified about.

The SUBSCRIBE message initiates a SIP dialog, unless it starts inside a previously initiated dialog. Not until all subscriptions inside a dialog have expired can the dialog be closed; this also applies if the dialog was started with INVITE and a BYE message is sent.

When the notifying node receives a SUBSCRIBE message, it issues a reply. If there is no error and the subscription can be admitted straight away, it will issue a "200 OK" response with the actual expiration time for the subscription. If there is no error in the request but the subscription cannot be started straight away because authorization has still not been granted, a "202 Accepted" response will be issued. Whichever of the two 2xx-class

responses is sent, a dialog will be created and the notifying node will immediately send a NOTIFY message.

Whenever a certain event happens and also every time a SUBSCRIBE message is sent inside a dialog to renew a subscription, the notifier will respond (with a 2xx response) and then issue a NOTIFY message to inform the subscriber about the state of the resource or the event that has occurred. Notably, this includes the unsubscribe messages (i.e. SUBSCRIBE messages with an **Expires** header value of 0).

A subscription can be **forked**, and this results in several SUBSCRIBE messages arriving at different nodes. The subscriber must therefore be prepared to receive NOTIFY messages from nodes other than the one he/she sent the SUBSCRIBE to.

SIP **proxies** need no additional behaviour to handle the Specific Event Notification extension, although they may decide to record-route the SUBSCRIBE and NOTIFY messages in a dialog.

The Specific Event Notification extension does not need the headers **Require**, **Proxy-Require**, or **Supported**. Instead, a node may query another node about its OPTIONS to see if the extension is supported. The presence of an **Allow-Events** header in a message is also enough to indicate that the extension is supported.

2.4 SPDP – The Service Peer Discovery Protocol

2.4.1 Introduction

Designing a service and service peer discovery protocol suitable for mobile, heterogeneous networks is a difficult task because of the many requirements that are imposed by the environment. In his master's thesis "Reconfigurable application networks through peer discovery and handovers" [4], Roberto Cascella proposed a protocol, the Service Peer Discovery Protocol SPDP, which builds upon SIP and the SIP Specific Event Notification extension to provide service and peer discovery functionality in such networks. In this section, this protocol will be summarized and explained.

2.4.2 Special requirements on SPDP

The network in which SPDP is supposed to work places some special requirements on the discovery protocol:

- It is a **heterogeneous** network. Specifically, the network under study combines GPRS and WLAN access, the latter in some specific places. This raises several issues:

- The protocol should work properly regardless of the current access network the device is using.

- However, the access technology may influence the device capabilities at the application level (for example, WLAN offers higher bandwidth than GPRS, GPRS does not support multicast, GPRS nodes use private IP addresses,…). The protocol should therefore be aware of and able to adapt to each of the access networks.

- A device can be connected simultaneously to both kinds of networks, and thus have two or more different IP addresses. The protocol should not only be able to uniquely identify the device, but should also be able to take advantage of the situation by enforcing intelligent roaming policies between both interfaces [36].

- It is a **mobile** network. The availability and location of peers and services can potentially change rapidly in comparison with service session duration, which means that:

  - The protocol needs to support both device mobility (i.e. a device may move and roam between different networks both in the middle of a service transaction or between service transactions), and personal mobility (i.e. a user can use several different devices to connect to the network).

  - The information about peers and service locations and availability must be kept consistent despite the mobility of the nodes.

  - The protocol should take into account the battery power consumption of the mobile devices when transmitting and receiving messages. Thus, the protocol must try to minimize the number of messages exchanged.

SPDP was designed with all these requirements in mind. It is structured as an Event Package which itself extends the SIP Specific Event Notification extension. The protocol deals with the requirements mentioned above as follows:

- It uses SIP to carry its messages:

  - SIP is an application level protocol and is independent of the transport level protocol used as well as of the underlying network infrastructure.

  - SIP provides an addressing scheme of its own, according to which every node or user has a unique identifier or SIP URI. This makes it possible for users to register with different

devices but with the **same** URI (this enables personal mobility) and to move from network to network while keeping their identity (enabling device mobility). SIP also provides forking of requests, which implies that a user who has registered with more than one device can get a call at every one of the devices he/she has registered. Furthermore, the addressing scheme also allows for a device which is connected to different networks and has more than one IP address to still be uniquely identified.

- A key feature of SIP's addressing scheme is that it allows every user to specify an address (called an address-of-record) where he/she can always be contacted. The user then relies on proxies to deliver the requests to its current location. In this way, a user can change address between two service sessions, but can still be contacted via the same address-of-record. Inside a communication session, a user can move and change address without the session being interrupted by using a reINVITE message informing the other party of the new address.

- SIP provides a registration mechanism that allows nodes to communicate with each other even if they do not know the other party's exact location, if they know the user address-of-record. This is done through proxies and registrars.

- SIP provides built-in security. Among other features, SIP provides authentication of the originators of requests, authentication of the servers to which requests are sent, end-to-end security of message bodies and/or particular message header fields, etc.

- It defines a naming convention and a common description for services and peers through the Extensible Mark-up Language (XML) [37], which is a widely supported technology.

- It contemplates mechanisms to minimize mobile terminals' battery power consumption or cost, such as using an intermediary such as a file server. File servers are useful when a user is roaming between networks with different speeds/prices (such as GPRS and WLAN). A user asking for a service from a slower/more expensive network may choose to forward the content to a file server, which will act as a temporary repository (i.e., a cache) until the user can connect to the faster/cheaper network in order to download the content.

2.4.3 Operation of the protocol

2.4.3.1 Compatibility with the SIP Specific Event Notification extension

As said before, SPDP is an event package which extends the SIP Specific Event Notification extension. More precisely, SPDP is an event package with these attributes:

Event package name: **Service Discovery**
Corresponding **Event** header field value: `sdpEvent`
Corresponding **Content-Type** header field value: `application/sxdp-xml`

Service discovery requests are embedded in SUBSCRIBE messages with an **Expires** value of 0. In this way, a NOTIFY message containing the current state of the peer and its services is sent, but the subscription is immediately closed. Although this functionality could have been obtained through a SIP instant message (i.e. a MESSAGE method), the subscription and notification extension assures that the request has been processed by the notifier before it responds, because the notifier sends a 2XX response upon reception of a request, but will not send a NOTIFY message until it has processed it. The SUBSCRIBE request body contains rules for the server formatted in XML. The NOTIFY message body contains the information requested in the subscription, also in XML.

Note that future subscriptions to the same provider can have an **Expires** value different from 0, if the requestor wishes to be notified of changes in the state of the peer or its services. In this case, SPDP does not allow notification frequency to exceed one notification every 5 seconds to avoid network congestion.

2.4.3.2 Protocol architecture

The protocol defines two types of entities:

- User agents, which are the peers that can join or stay in the network and that interact with each other. Every peer can simultaneously act as client or server:
    - User agent clients query other peers for services.
    - User agent servers accept service discovery queries, deliver services, and can also act as proxies on behalf of a client.

  User agents are uniquely identified by their SIP URIs as they move in the network. They have to support the Specific Event Notification extension.

- Context servers, which keep context information in the network. Each part of the network is served by zero, one, or more context servers. Context servers know about peer presence and capabilities, since peers register with them. A peer can locate a context server using

SIP localization for servers, which is based on DNS (see [38] for details).

The information in context servers for centralized networks, such as those described in [39], is collected and accessed by the operator. User privacy could be threatened if users are allowed to access this information without control. In a peer-to-peer network, in order to avoid privacy problems, users should be able to decide which information they want to give to the context server, for instance which services they want to advertise.

2.4.3.3 Protocol messages

SPDP messages formatted in XML are included in the body of SIP messages. They have a similar structure to SIP messages, with their own method, header fields and body.

The methods can be DISCOVERY, ACCEPT, or DENY. A DISCOVERY message is embedded in SUBSCRIBE requests, and queries a peer for its services or capabilities. It includes the conditions of the service in a conditionList field. If the queried peer has a service that matches the conditions, it replies with an ACCEPT message embedded in a NOTIFY request, which contains a list of services or peers in its body. If no matching service is found, it replies with a DENY message, also embedded in a notification.

The DISCOVERY queries can either ask for a peer or for a service type, as defined in a taxonomy tree. The service and entity taxonomy trees constitute a standard naming convention to identify each type of service or entity. Some taxonomy trees are proposed in the specification [4]. According to them, for instance, a web camera service is identified as `Root/Real Time/Audio-Video/Web Camera`, whereas a GPRS entity is identified as `Root/Physical Device/Access Point/GPRS`.

3 A Service Peer Discovery Protocol implementation

After the specification of the Service Peer Discovery Protocol was written, a message interpreter application [4] was created to illustrate the feasibility and the operation of the protocol. This application was capable of interpreting the SPDP messages that the user input, and perform some operations accordingly. The first goal of this thesis was to adapt that SPDP message interpreter application in order to create a sample implementation of the protocol. A key objective was also to generate appropriate documentation and information about the implementation to facilitate further development and re-use. In this section, the implementation that was created will be described in detail.

3.1 The implementation from the outside

The Service Peer Discovery Protocol implementation that has been developed was designed as an extension that provides a SIP User Agent with SPDP functionality. The implementation was therefore not designed to stand alone, but to be integrated into a SIP implementation and run as part of an SPDP-enabled SIP User Agent.

In this subsection, an overall external view of the implementation will be given. Firstly, the requirements of the implementation in terms of technology will be summarized. Then, the interaction with the SIP implementation will be explained. The actual file and directory structure will be described next. Finally, the different ways to run and/or test the implementation will be detailed.

3.1.1 Technology requirements

The implementation created requires some technologies to be available in the device it will run on, as detailed next.

3.1.1.1 Java Virtual Machine

The implementation was written in Java [23], and therefore a Java Virtual Machine is needed to run the applications. The Java technology used to develop the implementation was J2SDK 1.4.2_03.

3.1.1.2 JAXB

The Java Architecture for XML Binding (JAXB) [40] provides a convenient way to bind an XML schema to a representation in Java code. This technology performs the translation between the fields and attribute values in an XML document and the corresponding Java objects. The SPDP implementation uses JAXB to build and manipulate the protocol messages, the peer file and the service file (see section 3.2).

The implementation is compatible with the latest currently available version of JAXB, which is JAXB 1.0.2 as included in the Java Web Services Developer Pack (JWSDP) 1.3. Some of the libraries present in JWSDP 1.3 are required for the implementation to work, in particular the JAXB libraries and the JWSDP shared libraries.

3.1.1.3 Ericsson SIP implementation

The SPDP implementation was developed to interoperate with the SIP implementation by Ericsson [41]. The exact interface between both implementations is detailed in the next section. In order to use the current SPDP implementation, Ericsson's SIP implementation must be available in the device.

3.1.2 Interaction with the SIP implementation

As said before, the SPDP implementation was not designed to work alone, but rather as an extension to a SIP implementation. In particular, for compatibility with Roberto Cascella's SPDP message interpreter, the SIP implementation developed by Ericsson [41] was used.

In order to facilitate the re-use of this SPDP implementation with other SIP implementations, the interface between both was made as small as possible. The points of connection between both implementations are:

- In the SIP User Agent (class **UserAgent**), an attribute *spdpEngine* was included, to contain a reference to the SPDP engine associated with this SIP UA.

- In the SIP User Agent (class **UserAgent**), a new method *setSpdpEngine* was included. This method is used to attach an SPDP engine to the User Agent, thus enabling it to discover services through SPDP.

- In the SIP User Agent (class **UserAgent**), a new method *findService* was included. This method is the gateway to use SPDP to perform service discoveries.

- In the User Agent thread (method **UserAgent**.*run*), a few lines were included in order to check for SPDP messages in the body of SIP requests and if so place them in the SPDP message queue.

- The SPDP implementation uses two specific objects from the SIP implementation: a reference to the User Agent (instance of class **UserAgent**), and a reference to a SIP transaction (instance of **CallLeg**). The reference to the User Agent is needed in order for SPDP to make use of the SIP infrastructure, since SPDP messages are encapsulated in SIP SUBSCRIBE requests. The reference to the transaction is needed in order for SPDP to be able to send a response embedded in a NOTIFY SIP request belonging to the same transaction

as the SUBSCRIBE SIP request in which the SPDP request was embedded.

Note that an important implementation decision was taken by only providing the SIP UA with an entry point to trigger a *service* search, never a *peer* search. The reason for this decision is that it was considered that a user, whether human or not, will always want to access and utilize a *service*, since a peer has no use to the user except for the services it provides. Internally, of course, both service *and* peer *discoveries* are performed in the protocol, as the specification dictates. The internal division between services and peers is very appropriate since it provides a means of aggregating information – a peer need not have all the services provided by another peer in its service file, it can be enough to have an entry for the peer in the peer file. For more details about this, see section 3.2 and section 3.3.2.

The SPDP implementation uses two objects taken from the SIP implementation: **UserAgent** (which represents a SIP UA) and **CallLeg** (which represents a SIP transaction). These objects are central to the operation of SIP and are very likely to be found as such in other implementations of SIP. Therefore, the changes that are needed in this SPDP implementation in order to make it compatible with another SIP implementation written in Java are likely to be minimal, and are estimated to take no longer than one or two days of work.

3.1.3 File structure

The SPDP implementation is contained in the directory *newspdp*. The structure of its subdirectories and files is as follows:

- **Directory *spdp***: This directory contains the specific classes of the implementation (package *newspdp.spdp*), such as the SPDP engine. It also contains the executable class to test the implementation (class **SpdpUserAgentGui**).

- **Directories *peer*, *service,* and *spdpmessage***: These directories contain the source files generated by the JAXB compiler from the XSD schemas (packages *newspdp.peer*, *newspdp.service*, and *newspdp.spdpmessage*). These classes represent the different objects described in the schemas: a peer list, a service list, or an SPDP message.

- **Directory *_spdplibs***: It contains the necessary libraries for JAXB functionality.

- **Directory *_spdpfiles***: It contains some files that are relevant to the protocol implementation and tests, such as the XSD schemas for the peer list, the service list, and the protocol messages, and backup files for the initial service and peer information used for the tests.

- **Directory *_javadoc***: It contains the Javadoc documentation of the implementation. The reader is encouraged to browse it for a complete description of all the packages, classes, methods, and fields present in the SPDP implementation.

- **Files *services.xml* and *peers.xml***: They are the service respectively peer information repositories, intended to be accessible to every application wanting to be informed about known peers and services, and accessed and modified by the SPDP implementation when performing discoveries and/or processing external requests.

3.1.4 Running the implementation

As stated above, the SPDP implementation is not a standalone application, since it needs to be attached to a SIP User Agent. A simple SPDP-enabled SIP User Agent based on this SPDP implementation and on Ericsson's SIP implementation can be started by running class **SpdpUserAgentGui** with the appropriate parameters described below. This class provides a simplified graphical user interface to the User Agent, allowing the user to access only the service discovery functionality. In a terminal window, type:

```
java newspdp.spdp.SpdpUserAgentGui <username> <host> <port>
```

The parameter *username* specifies the user name of the User Agent; it defaults to the user's name for the system (i.e., the value of the system property "user.name"). The parameter *host* specifies the name of the host in which the User Agent runs; by default, the IP address of the device is used (specifically, the address returned by the call to **InetAddress**.*getLocalHost().getHostAddress()*). Finally, the *port* parameter specifies the port on which the User Agent will run; if not specified, the SIP default port number (5060) is used.

3.2 Data model

There are three distinct data structures involved in the Service Peer Discovery Protocol. Firstly, the protocol messages that SPDP peers exchange in order to perform discoveries; secondly, the structure that contains the peer information that a peer has; and finally, the equivalent structure containing the service information known to a peer.

While protocol messages were dynamically generated, the peer and service information structures were kept in files residing in each peer. This was done to provide the easiest possible access to the information for the different applications running in the peer, which can browse the files at any time. The price to pay is the increase in processing power and time required by having to update the files for every discovery.

The Service Peer Discovery protocol uses the Extensible Mark-up Language (XML) [37] to represent all protocol messages, as well as the service and peer information stored in local files. The next subsections explain the formats of those XML documents.

3.2.1 The SPDP message format

The format of every SPDP message is specified in an XSD schema called *spdpMessage.xsd*. Below is an example of an SPDP message[1]:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes">
<spdpMessage>
      <requestId>24537094108090027082l</requestId>
      <sender>
            <entityId expire="3600">
                  sip:pojken@130.237.15.247
            </entityId>
            <entityAddress entityType="IPv4">
                  130.237.15.247
            </entityAddress>
      </sender>
      <method>
            <name>ACCEPT</name>
      </method>
      <serviceRequest>
            <path>printer</path>
            <value>EasyPrint</value>
      </serviceRequest>
      <expireTime>4000</expireTime>
      <replyTo priority="1">
            <entityId expire="3600">
                  sip:pojken@130.237.15.227
            </entityId>
            <entityAddress entityType="IPv4">
                  130.237.15.247
            </entityAddress>
      </replyTo>
      <content>
            <serviceList>
                  <service>
                        <serviceId>printer</serviceId>
                        <name>EasyPrint</name>
                        <source>
                              <entityId expire="3000">
                                    sip:gonzalo@130.237.15.248
                              </entityId>
                              <entityAddress entityType="IPv4">
                                    130.237.15.248
                              </entityAddress>
                        </source>
                        <protocol>RTP</protocol>
                  </service>
            </serviceList>
      </content>
</spdpMessage>
```

[1] By default, all XML documents generated with JAXB are encoded using UTF-8. The encoding to be used can be specified as a property of the JAXB marshaller.

An SPDP message contains the following fields:

requestId: This is a compulsory field that contains a string that uniquely identifies the SPDP transaction that this message belongs to. An SPDP transaction consists of one request and one response to that request. In this implementation, it is generated by appending the current time in ms to the hash code of the SPDP engine that initiated the request.

sender: The identity of the sender of this message. In an SPDP message, identities are represented as an **entityId** (with an optional expire attribute), which is the SIP URI of the entity, and an **entityAddress** (with the compulsory attribute entityType) that will usually contain the IP address of the entity.

method: The method of the message [4]. It consists of a name which can be DISCOVERY, ACCEPT, or DENY, and an optional responseCode attribute.

peerRequest or **serviceRequest**: Only one of these fields will be present in a given message. They contain the peer or service that is searched for, specified by a **path** (in this case, a printer) and a **value** (in this case, EasyPrint, the type of printer).

expireTime: The protocol specification regards this value as an overall expiration time for all the fields in the message. This implementation, however, only takes into account each of the individual expire times for the specific fields (such as those in the various **entityId** fields), since the meaning of an overall expiration time can be ambiguous; the implementation will therefore ignore this field.

replyTo: The identity of the peer to which the reply to this message should be sent.

conditionList: This is an optional sequence of **condition** fields to refine the search. Each condition consists of a path (called the **nodeId**) with a taxonomy attribute, and a **value** for the condition. The taxonomy attribute is present in the specification, potentially to more easily match the conditions to some pre-defined conditions structured in taxonomy trees. As these taxonomy trees have not been defined, the taxonomy attribute is ignored in this implementation, and the verification is done directly on the **nodeId**.

content: The actual peer and/or service information contained in the message. Usually, content will only be present in the ACCEPT messages, and it will consist of either a **serviceList** or a

peerList, according to whether it is a service search or a peer search. The format in which peers are represented in the **peerList** object is the same as in the peer file (see section 3.2.3) except for the expire time format, as explained below. Analogously, the format in which services are represented in the **serviceList** object is the same as in the service file, described in section 3.2.2, except for the format of the expire time (see below).

As explained before, all SPDP messages are embedded in SIP messages. DISCOVERY messages are embedded in SIP subscriptions (method SUBSCRIBE), while ACCEPT or DENY messages are embedded in SIP notifications (method NOTIFY).

It should be noted that all expiration timers in the SPDP messages are integers that represent the number of seconds for which the information they refer to will be valid, from the moment the message is issued. Here, the assumption is that the delay in receiving messages is less than one second, and hence the time is not exact, it is ±2 seconds.

3.2.2 The service file format

The service file is the cache that contains all the information about the services that a peer knows. This file:

- is written to by applications running in the peer that want to advertise their services via SPDP.
- is read by the SPDP engine as the first step of a service search.
- is written to by the SPDP engine when it receives a message containing new service information.
- is read by applications that wish to access to the information about the known services by themselves.

The service file is named *services.xml*. Below is an example of a service file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<serviceList xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="serviceList.xsd">
      <service>
            <serviceId>filemp3</serviceId>
            <name>The Autumn.mp3</name>
            <source>
                  <entityId expire="3000">
                        sip:pojken@130.237.15.247
                  </entityId>
                  <entityAddress entityType="IPv4">
                        130.237.15.247
                  </entityAddress>
            </source>
```

```
            <protocol>RTP</protocol>
            <expires>2004-05-23T17:58:17+01:00</expires>
      </service>
      <service>
            <serviceId>sipphone</serviceId>
            <name>Pojken</name>
            <source>
                  <entityId expire="3000">
                        sip:pojken@130.237.15.247
                  </entityId>
                  <entityAddress entityType="IPv4">
                        130.237.15.247
                  </entityAddress>
            </source>
            <protocol>RTP</protocol>
            <expires>2004-06-07T01:00:00+01:00</expires>
      </service>
      <service>
            <serviceId>printer</serviceId>
            <name>EasyPrint</name>
            <source>
                  <entityId expire="3000">
                        sip:gonzalo@130.237.15.248
                  </entityId>
                  <entityAddress entityType="IPv4">
                        130.237.15.248
                  </entityAddress>
            </source>
            <protocol>TCP</protocol>
            <expires>2005-03-02T12:03:37+01:00</expires>
      </service>
</serviceList>
```

The service file is a **serviceList** object. It contains a sequence of fields of type **service**. Each **service** entry consists of the following fields:

serviceId: The path to the service. Examples of **serviceId** are printer, filemp3, sipphone…

name: The specific service value. Examples of **name** can be EasyPrint for the printer **serviceId**.

source: The identity of the peer that provides the service. This field follows the same format as the **sender** field in the SPDP message.

protocol: The protocol used to access the service: RTP, FTP,…

expires: The date and time until which this service is supposed to be found in the specified peer with the specified parameters. It is stored with a precision of a second.

Note that a peer needs to convert expiration timers from integer values in seconds to dates and vice versa when generating or interpreting SPDP messages.

3.2.3 The peer file format

The peer file is the cache that contains all the peer information known by this peer. On start-up, it should contain at least one record, corresponding to the information of the local peer (that is, every peer should have information about itself). Although the peer file can be accessed by other applications, it will usually be the SPDP engine which will read or write to it. The peer file is named *peers.xml*. An example of a peer file is shown below:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<peerList>
      <peer>
            <identity>
                  <entityId expire="3780">
                        sip:diego@130.237.15.211
                  </entityId>
                  <entityAddress entityType="IPv4">
                        130.237.15.211
                  </entityAddress>
            </identity>
            <network>WLAN</network>
            <cellId>3454FG</cellId>
            <expires>2004-05-26T12:03:37+01:00</expires>
            <services>
                  <service>filemp3</service>
            </services>
      </peer>
      <peer>
            <identity>
                  <entityId expire="3780">
                        sip:pojken@130.237.15.247
                  </entityId>
                  <entityAddress entityType="IPv4">
                        130.237.15.247
                  </entityAddress>
            </identity>
            <network>WLAN</network>
            <cellId>3454FG</cellId>
            <expires>2004-05-27T09:00:37+02:00</expires>
            <services>
                  <service>filemp3</service>
                  <service>sipphone</service>
            </services>
      </peer>
</peerList>
```

A peer file contains a **peerList** object. The **peerList** consists of a sequence of **peer** objects, each of them with the following fields:

identity: The identity of the peer, represented in the same format as the **sender** field in an SPDP message

network: The type of network that the peer is attached to, for instance WLAN, GPRS,…

cellId: The identifier of the cell in which the peer is, if applicable.

expires: The date and time up to which the information about the peer can be considered valid.

services: A list of elements of type **service**, representing the service ids that the peer provides. Note that no service values are stored in the peer file, but the information is aggregated by only considering the service ids.

3.3 Operation of the SPDP implementation

This subsection describes how the SPDP implementation is structured internally and how it discovers peers and services. It will be structured according to the different threads that run concurrently and provide the SPDP functionality. First, an overview of the different threads will be given, then each of them will be described separately: the main thread, the SPDP engine thread, and the request handler threads.

3.3.1 Threads in the implementation

The main threads that interact in the SPDP implementation (that is, when an SPDP-enabled SIP User Agent is running) are depicted in Figure 6. The User Agent is started by the main thread, which in this case is also the thread that launches the graphical user interface (class **SpdpUserAgentGui**). The main thread also spawns the SPDP engine thread, and passes a reference to the SPDP engine object to the SIP User Agent to enable SPDP. Then, the main thread waits for the user to interact with the GUI.

The SIP User Agent thread spawns a listener thread (class **ericsson.sip.protocol.ListenerThread**) on start-up. The listener thread listens for datagrams that arrive at the specified SIP port (by default, it is port 5060). Upon arrival of a datagram, the listener thread parses it and, if it is a valid SIP message, places it in the SIP message queue. After that, the listener thread continues listening to the SIP port.

The SIP User Agent thread (class **ericsson.sip.ced.UserAgent**) periodically checks the queue to process SIP messages that arrived, according to whether they are requests or responses and to the request methods. As a specific feature of the SPDP-enabled User Agent, the UA checks the body of the SUBSCRIBE and NOTIFY requests arrived to look for SPDP message

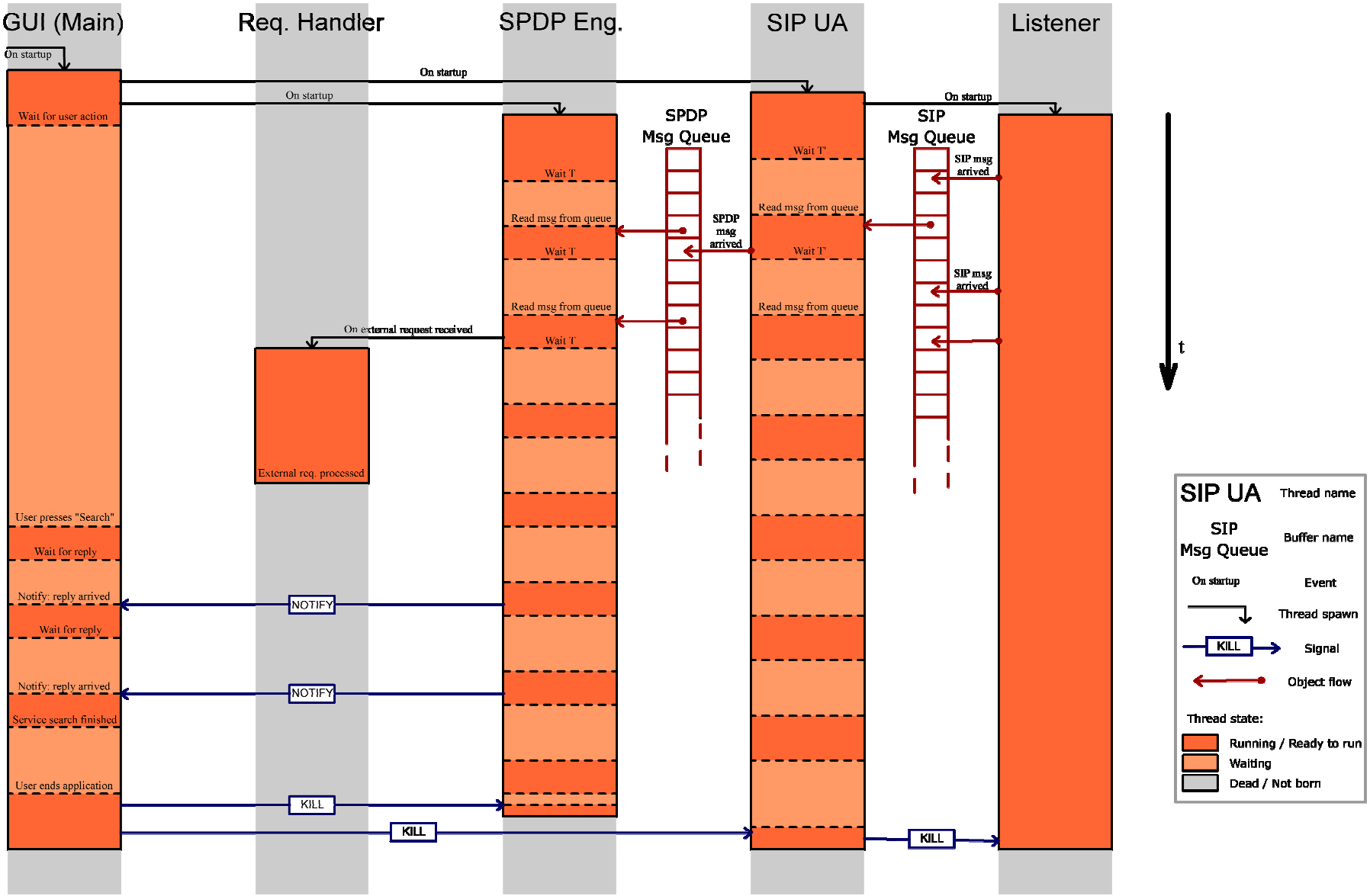

Figure 6: Interaction of the threads in the SPDP implementation

If so, it places the SPDP message (the body of the SIP request) in the SPDP message queue.

The SPDP engine thread (class **SpdpEngine**) periodically checks the SPDP message queue to process the SPDP messages that arrive. The messages can either be replies to a locally initiated search or requests coming from another peer. If the SPDP engine receives a request from another peer (that is, an external request), the engine will pass the request on to a request handler thread.

The request handler threads (class **RequestHandler**) process the external requests. They perform a search in the service or peer file, depending on the type of request, and issue the corresponding SPDP response: ACCEPT (if matching services or peers were found) or DENY (otherwise). After a request handler finishes its task, it will wait until it is assigned another.

When the user requests a service by interacting with the GUI (this is done for instance by typing in a service name in the corresponding text field, and then pressing the "Search service" button), the main thread invokes a method in the **SpdpEngine** object in order to search for the service. If some SPDP messages have to be sent, the main thread waits for the reply. When the service search is completed, the main thread shows the results in the GUI.

Note that the external interface to the SPDP implementation does not allow for peer searches (there is **no** method *findPeer*). The user can only search for **services** provided by peers, but not for peers themselves (a peer is assumed to be just a service providing entity, as it is services that a user will ultimately require). However, the implementation of course performs peer discoveries as a means to search for services. A service search can consist of several peer and service discoveries.

In the SPDP engine thread, if the message taken from the SPDP message queue belongs to an internal search (i.e., a search initiated by the user), the SPDP engine will notify the main thread, which was waiting for the reply.

When the user decides to terminate the User Agent (for instance by closing the window), the threads are destroyed and the application ends.

In the next sections, the operations performed in each of the SPDP-specific threads will be explained in more detail.

3.3.2 Main thread flow

The main thread is invoked by an application that launches the SPDP-enabled SIP User Agent. The SPDP implementation provides one such application (class **SpdpUserAgentGui**), which creates a graphical user interface to the User Agent with limited access to the agent's functionality: it only allows the user to search for services using SPDP.

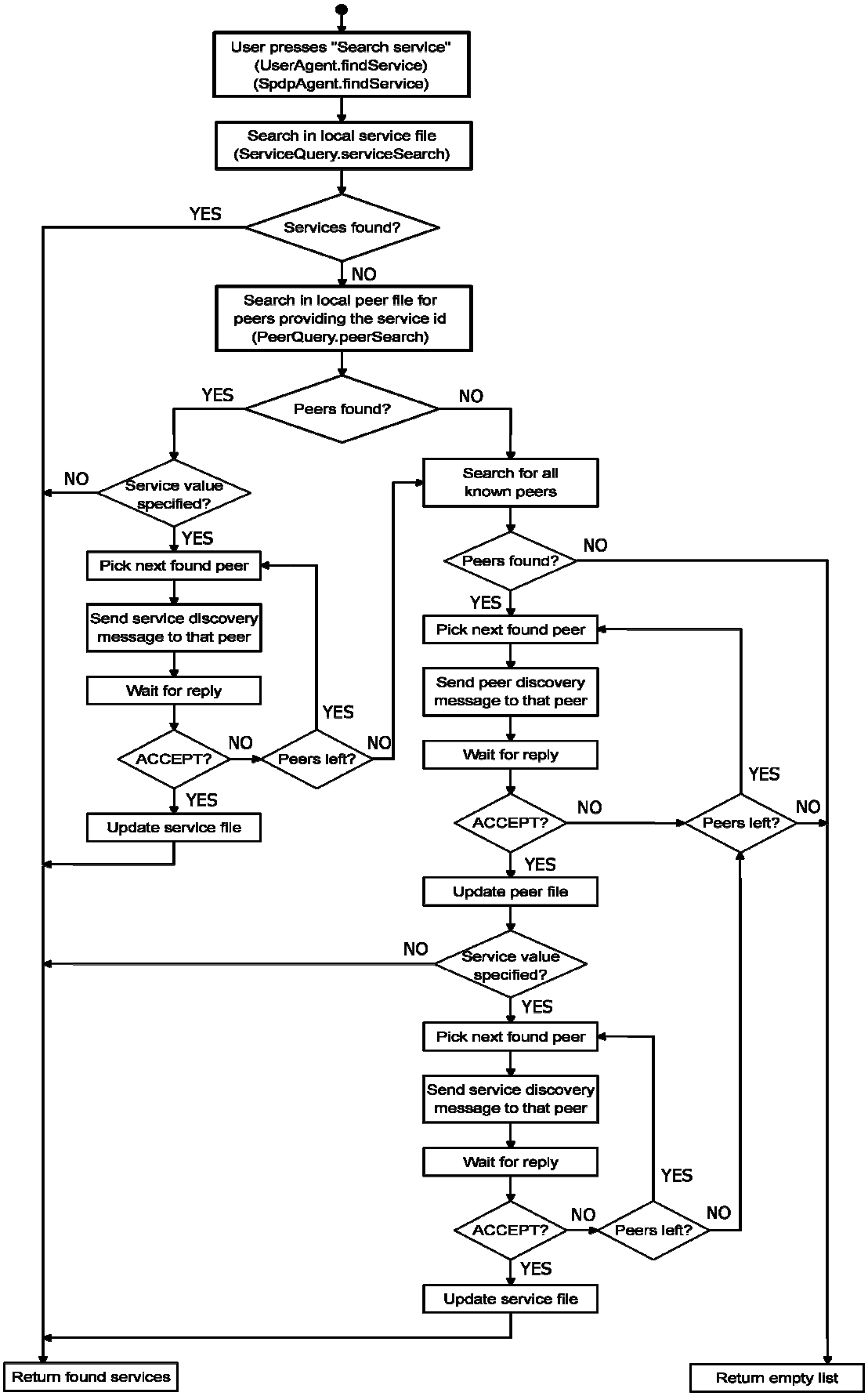

Figure 7: Flow diagram of the main thread

The GUI is depicted in Figure 8. It consists of a simple frame, with text fields to input the path and (optionally) the value of the service to search for, and the optional conditions of the search (input as a comma-separated sequence of strings of the form *conditionPath=conditionValue*, or selected from a list). It also includes a "Search service" button to trigger the search, and a console text area where the information is output to the user. Examples of service names can be: *printer*, *sipphone*, *filemp3*,… For the *filemp3* service name, examples of values can be *The Spring.mp3*, *The Best Castilian Jotas*, *Happy Birthday.mp3*,…

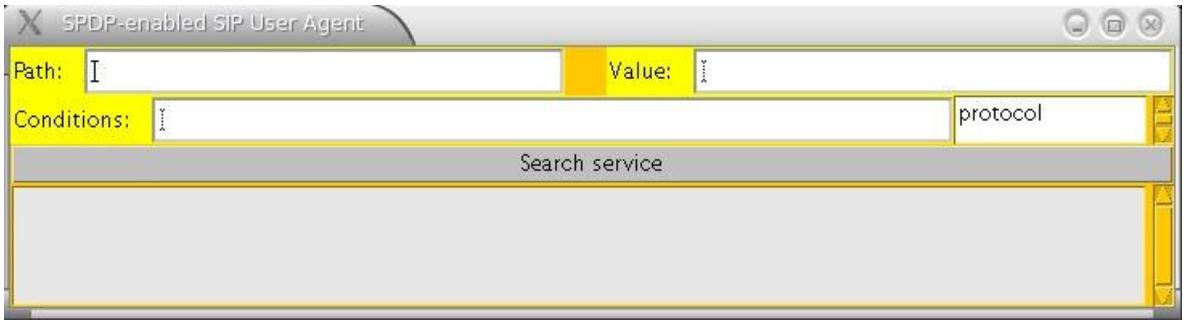

Figure 8: Graphical user interface to the SPDP-enabled SIP UA created in class SPDPUserAgentGui

When the user starts a search, the method *findService* in class **UserAgent** is invoked by the main thread. This method directly calls the corresponding *findService* method in the **SpdpEngine** object. As explained before, this method is the single entry point to make use of the SPDP implementation. Method *findService* implements the service discovery algorithm depicted in Figure 7.

Services are first searched for in the service file (named *services.xml*). If a matching service is found in the file, then the search ends and the results are output to the user.

Otherwise, a search is performed in the peer file (*peers.xml*). Every peer in the peer file is described by a record that includes the service names that the peer offers (see section 3.2.3). If there are some peers in the peer file that offer the requested service id, a DISCOVERY message will be sent to them one by one, until one of them replies with a successful response (an ACCEPT response), and the search will end successfully. The DISCOVERY message will be a service request for the specified service id *and* value. Peers are queried sequentially in this implementation since it was estimated that traffic is usually a more important constraint in the scenarios where SPDP runs than time. Note that the search will end as soon as **one** ACCEPT response (i.e., a response containing at least **one** matching service) is received, but the service description to match can be defined as accurately as needed. In this implementation, peers are queried in the (sequential)

order in which they are in the peer file. A random querying order may be more efficient in order to find a match with fewer queries; it is left as future work to test and implement this if appropriate.

The service value is optional. If it was not specified, then the lookup in the peer file may directly yield valid results without having to send any DISCOVERY messages.

If none of the peers in the peer file provide the requested service id, or the peers that provide the service id do not provide the service value, then new peers need to be discovered.

A peer request will be sent to each of the known peers, asking them for peers that offer the service id. If an ACCEPT response is received for this peer request, containing information about new peers, those peers will be queried with a service request, asking them for the requested service value. If a matching service is found, the search will end.

The search is given up when all the known peers have been asked for their known peers that provide the service id, and all these newly-acquired peers have been asked for the requested service value, and no matching service has been found.

The mechanism used to issue and identify requests and responses works as follows:

- First, the identity of the destination peer is checked, since no requests should be issued to oneself, nor to a peer that has previously been queried for the same request (in the same search) and responded with a DENY.

- Then, a request id is generated randomly. This id will differentiate this request from any other, and will be included in the response to this request to identify it.

- The request (a DISCOVERY message) is sent to the destination peer.

- The thread waits for a reply for the request. The maximum waiting time is specified as a constant in the **SpdpEngine** class.

- The transaction is completed after a response that matches the stored id arrives, or the timeout value is reached.

- The request id of the newly-completed request is stored in a buffer. This allows delayed responses to requests to be discarded and not be mistaken with new external requests (whose id is not present in the buffer). The length of the buffer is also specified as a constant in **SpdpEngine**.

Whenever new information is acquired about new peers or services, namely because an ACCEPT has been received to a request, the peer or service file is updated.

When the search is complete, the information about the services found is output to the user in the GUI (see Figure 9). An informational message is also displayed if no services have been found. The main thread then waits for the next search.

Figure 9: The SPDP-enabled SIP User Agent GUI showing the results of a successful search.

Note that this algorithm implies that a service will only be found if it is provided by the same peer which searches for it (at "distance 0"), by a known peer (at "distance 1") or by a peer known to at least one of the known peers (at "distance 2"). This way of proceeding, although not specified in the protocol specification, was already present in the message interpreter application that this implementation is based on. Choosing a maximum depth of 2 "hops" in the "knowledge graph" was considered to be a sensible decision, since a higher value would imply longer, more traffic-consuming searches, usually without considerably increasing the probability of finding a service, whereas a lower value of 1 cannot ensure that the information is properly propagated in the network. This crucial implementation decision has some implications in the testing and evaluation of the protocol (see chapter 4).

3.3.3 SPDP engine thread flow

The SPDP engine thread continuously checks the SPDP message queue for new SPDP messages that arrived, and processes the messages accordingly. Its flow is depicted in Figure 10.

If there is a message in the queue, it will be unmarshalled (converted from a string to an object of type **SpdpMessage**, which represents a protocol message according to the *spdpMessage.xsd* schema) and its request id will be checked to detect if it is a message for the internal request, if it is an

external request or if it was a delayed response to a previously issued request.

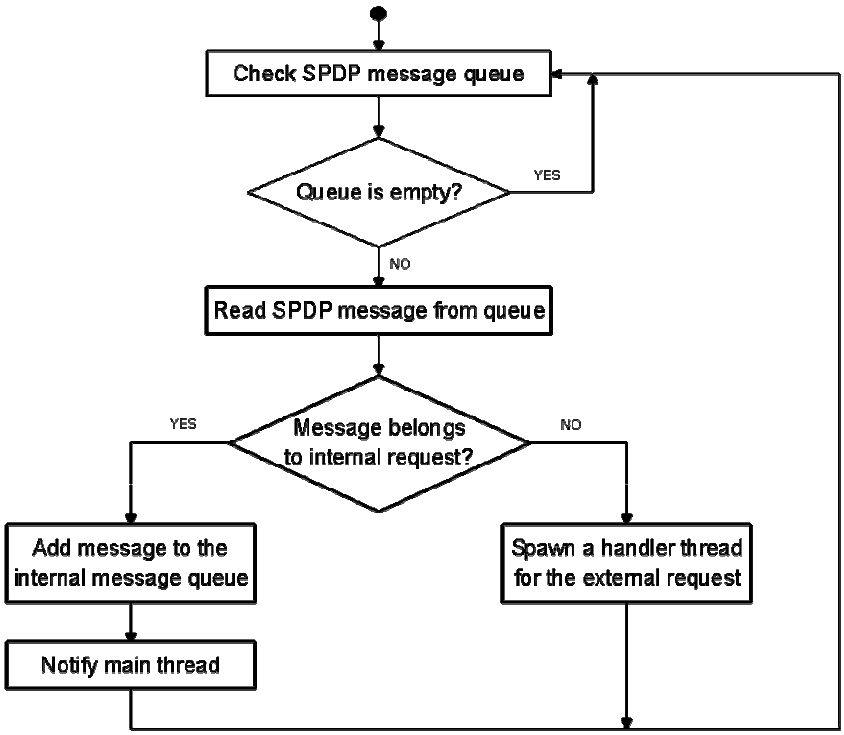

Figure 10: Flow diagram of the SPDP engine thread

If an external request is received, one request handler thread is randomly picked from the thread pool and, if it is not already busy, the SPDP message and the SIP transaction that it belonged to are passed to the handler. The handler is then awakened.

If the received message was a response for the internal request, then the message is added to the internal message queue, and the main thread is notified.

3.3.4 Request handler thread flow

The flow of the request handler threads is shown in Figure 11. A request handler thread will wait until an external request arrives and is assigned to it by the SPDP engine.

The handler will then check that the method in the message equals DISCOVERY. In order to take advantage of as much network information as possible, the peer file will be updated with the information that the request provides about its sender in the *replyTo* field.

The request will then be processed. If it is a service request, the service file will be searched in order to find a service that matches the conditions. If it is a peer request, the peer file will be searched. In either case, depending on whether the search was successful, an appropriate response will be issued (an ACCEPT or DENY message).

After the message is processed, the request handler returns to its wait state.

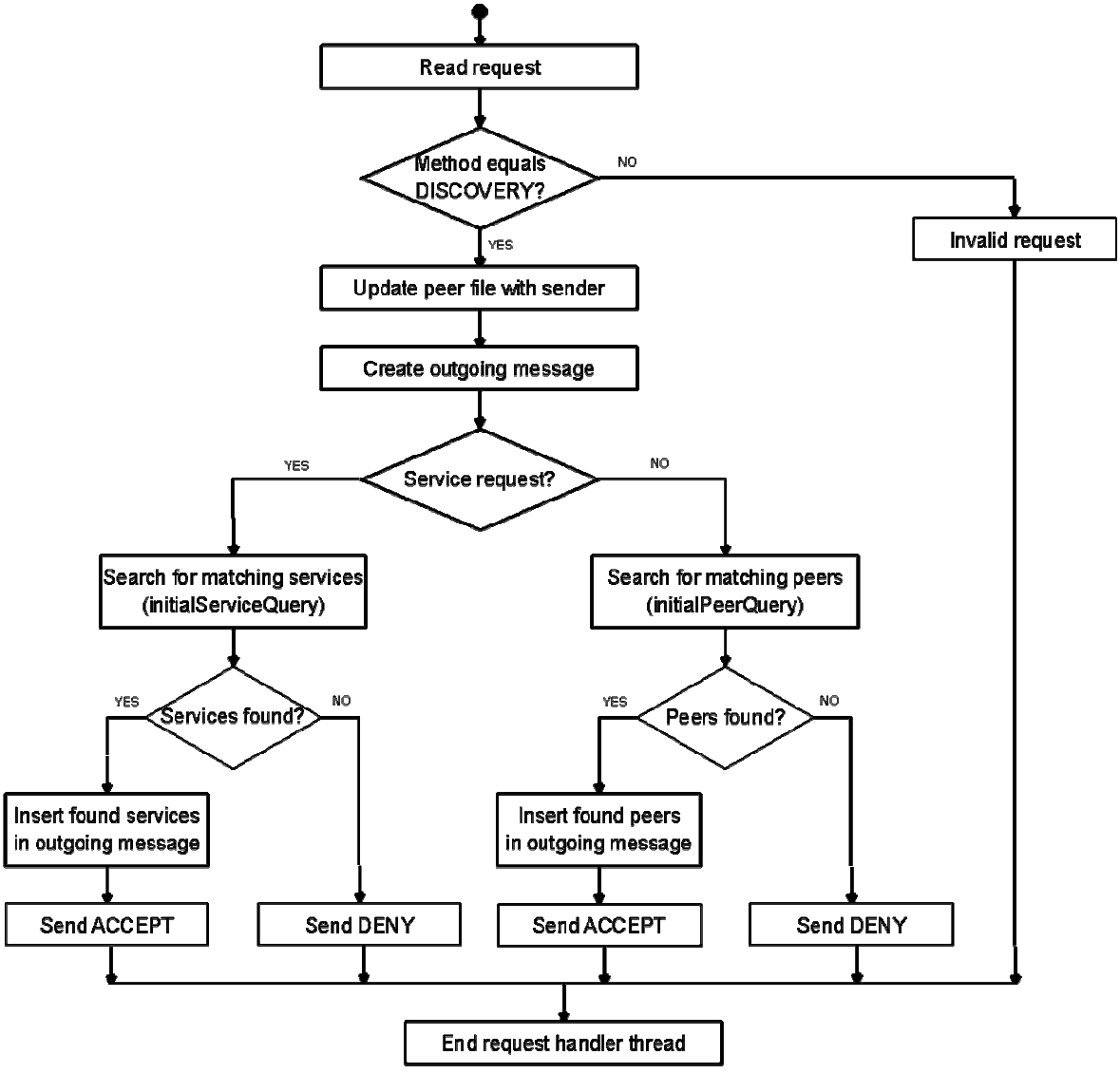

Figure 11: Flow diagram of the request handler thread

4 Evaluation of the Service Peer Discovery Protocol

After the implementation of the Service Peer Discovery Protocol has been described in the previous chapter, this chapter is devoted to the evaluation of the protocol.

The evaluation process is structured in three parts. First of all, a theoretical comparison was established between SPDP and several of the discovery protocols described in section 2.2.

After that, the implementation was used to compare the performance of SPDP and the Service Location Protocol (SLP) [7] in some controlled test scenarios. This comparison shows some of the differences that are inherent to both protocols and allows some conclusions to be drawn concerning the way both protocols behave in their simplest forms.

To evaluate the performance of SPDP in more realistic settings, a discussion in the second part of this chapter considers some SPDP scenarios, which are analysed and conclusions are drawn.

4.1 SPDP in comparison with other discovery protocols

The Service Peer Discovery protocol has some characteristics that distinguish it from other service discovery protocols: it interacts very closely with SIP, which it uses as a carrier for the protocol messages; it uses XML to represent the protocol information; and it relies on unicast to perform discoveries. In this section, some of the main service discovery protocols are compared to SPDP. Refer to section 2.2 for descriptions of the protocols discussed in this section.

4.1.1 SPDP and the Service Location Protocol (SLP)

SLP is a widespread service discovery protocol, standardized in RFC 2608 [7], and the subject of intense research. Its architecture, consisting of UAs, SAs, and DAs, is simple and robust, and has inspired some of the other discovery protocols [24]. Thanks to the use of multicasting, to the direct use of the transport layer as carrier of the protocol messages, and to a simple text-string representation of the services, SLP is a traffic and time-efficient protocol.

SLP was not specifically designed for rapidly-changing networks, but its performance in such scenarios can be enhanced by the presence of a DA colocated with the access point (or another fixed node). In fact, SLP takes full advantage of the presence of a DA in the network, and it provides an easy way in which peers can turn to DAs dynamically, by changing the configuration of the SLP daemon.

In comparison to SPDP, however, SLP provides a less flexible representation of services through service URLs. Furthermore, its extensive use of multicast brings about some problems in the context of the peer-to-peer wireless scenario. SLP was selected to be tested against SPDP for some simple scenarios, and the results and conclusions of those tests can be found in section 4.2.

4.1.2 SPDP and JXTA

Unlike SPDP or SLP, JXTA defines a framework for peer-to-peer networking rather than a single protocol. Among the different protocols specified in JXTA, a discovery protocol is included, the Peer Discovery Protocol (PDP). A demo application that allows a user to connect to the JXTA network [42] is provided at the JXTA homepage [31]. A disadvantage found when using this demo application is that the address of at least one rendezvous agent must be provided by the user at start-up in order to connect to the World Group.

There is a great flexibility in JXTA. The specification of PDP [32], for instance, encourages other discovery protocols to be used in local communities if they are considered more convenient than PDP, and provides easy integration of those protocols with the rest of the architecture. Another example of flexibility is that the carrier protocol for the JXTA messages is also unspecified: TCP, UDP, and even HTTP are suggested.

An important feature of JXTA [32] is that it provides mechanisms to guarantee connectivity for every peer even if it is behind a NAT or a firewall, which is often the case in the scenario studied.

But all in all, it is difficult to compare JXTA and SPDP at the same level. Indeed, SPDP could be used as the internal discovery protocol for a JXTA group in a wireless network.

4.1.3 SPDP and Universal Plug and Play (UPnP)

UPnP shares the same architecture principles as SLP or Jini. It also relies on central information servers to provide the service information, even if the protocol could work without a centralised point.

An interesting feature of UPnP is that it represents services through service URLs as SLP (although the formats are incompatible), but the URLs point to the complete service descriptions formatted in XML (as with SPDP) [24]. In this way, the amount of traffic exchanged is kept low, while taking advantage of the features of XML for the representation of services. The price is of course a detachment of the full service description from the discovery process, since it is the URLs that are exchanged, and the introduction of an intermediate step in service discovery: "simple discoveries" only examine service URLs, while "full discoveries" take into account the XML descriptions [24]. UPnP requires HTTP to perform "full discoveries".

4.1.4 SPDP and Jini

As stated in section 2.2.4, Jini shares the same architectural principles as SLP, with Lookup Services playing the role of Directory Agents. The most characteristic feature of Jini is that it is very tightly bound to Java: services are represented as Java objects and the communication is performed via Java's Remote Method Invocation (RMI).

Jini requires that every peer has a Java Virtual Machine (JVM) running in order to be part of the community. This can be a disadvantage in mobile devices with limited memory, processing power, and battery, but there is intense on-going research on lightweight, power-efficient JVMs that could alleviate this problem.

However, unlike SPDP, Jini requires both a centralized point of information (an instance of the Lookup Service) and the possibility to multicast messages in order to discover peers (at least to discover the Lookup Service). This can be disadvantageous in the scenario studied, where networks will often be too small to support the cost of including a central point of information, and where multicasting can be a problem in certain network technologies (such as GPRS).

4.2 SPDP and SLP: comparison for simple test cases

This section presents the results of measurements of SPDP and a comparison with the Service Location Protocol (SLP) in terms of traffic and discovery time for several specific cases. First, the tests that have been performed are described: which statistics have been measured and how, as well as the test environment. After that, the results of the measurements for SPDP and SLP are presented. Finally, the main conclusions are drawn.

Note that the conclusions have been drawn bearing in mind that the objective was to judge and compare both protocols, **not** their implementations. Notably, the implementation of SPDP used can be subject to many improvements, as detailed in chapter 5, and a more in-depth analysis by a skilled programmer is bound to yield many more improvements in the implementation. The goal of this project was not so much to produce a competitive, efficient implementation as to illustrate and show the capabilities of the protocol. This is why the measured values, although accurate for the implementation, have to been taken as orders of magnitude estimations for the protocol. This is one of the reasons why the discovery time graphs in the results subsection have been plotted using a logarithmic scale.

4.2.1 SPDP test cases

In order to illustrate the operation of SPDP, its message flow and the service discovery algorithm, several test cases were designed which

consisted of one specific service search. The test conditions and the tests themselves are described next.

4.2.1.1 Test conditions

For these tests, three computers connected to the laboratory's LAN were used. In each of them, a dual SPDP+SLP peer was started (these peers were able to receive, process, and reply to both SPDP and SLP messages). The SPDP implementation used was the one described in this document (see chapter 3); the SLP implementation used was OpenSLP provided by SourceForge [43]. The connectivity of the peers is depicted in Figure 12.

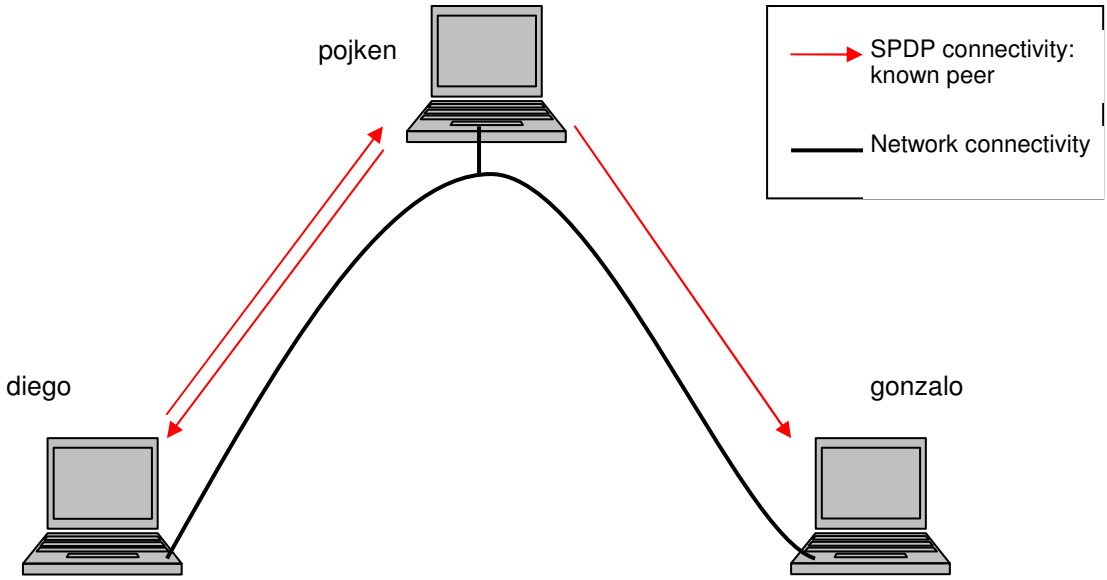

Figure 12: Network and logical connectivity of the peers in the test scenario

For the **SPDP** protocol:

- Each peer had initial *partial* knowledge of the network at start-up. Apart from the locally provided services, which SPDP requires every peer to know, peer *diego* knew of the existence of peer *pojken*, which in turn knew about the existence of both *diego* and *gonzalo*. *gonzalo* did not know about the existence of any of the other peers. As for the services, each peer had limited knowledge of all the available services as well. Each peer's exact start-up knowledge of peers and services is contained in files `peers-diego.xml`, `peers-pojken.xml`, `peers-gonzalo.xml`, `services-diego.xml`, `services-pojken.xml` and `services-gonzalo.xml`. These files are included as Appendix B.

- SPDP does not specify the strategy of the peers at start-up or how they initially learn of peers connected to the network, although the protocol suggests several ways, such as from a Context Server,

broadcasting peer discovery messages, or static network knowledge. In this case, the simplest strategy was to provide every peer with some static knowledge at start-up. However, the behaviour of this network is equivalent to that of another network with a broadcasting strategy **and** without a context server. The only difference is the time that it takes the peers in the first network to learn what the peers in the second network would know from the start. This means that the measurements performed are independent of the start-up strategy, for any network without a context server.

In order to make the comparison as fair as possible, the **SLP** protocol was used as follows:

- Each SLP peer provided the same services as the SPDP peer co-located with it. The translation between the XML-formatted SPDP service descriptions to the SLP service URLs was performed taking into account only the *serviceId* and *value* of the SPDP services, while ignoring the rest of the fields. Then, the SLP service URL was built as:

  `serviceId:value://host:port`

  with no attributes. Since every SLP service running on a host requires a different port number, random port numbers were assigned.

- Since the network runs without a Context Server, no SLP Directory Agent (DA) was present. In this way, neither of the two protocols had a centralised source of information.

4.2.1.2 Tests

Ten different test cases were studied in order to illustrate the operation of SPDP and its service discovery algorithm. All of them consist of a service discovery for a particular service. Note that no peer discovery was performed *per se*, since this SPDP implementation does not support it. However, this does not limit the validity of the tests since many of the service discoveries involve one or more peer discoveries. Peer discovery was used as a tool for discovering services.

The tests are related to the way SPDP discovers services. The control flow for each test is depicted in Figure 13.

Table 1 gives the details of the services that were searched for in each test and the service and peer files that were used as start-up knowledge. All searches were initiated by the peer *diego*.

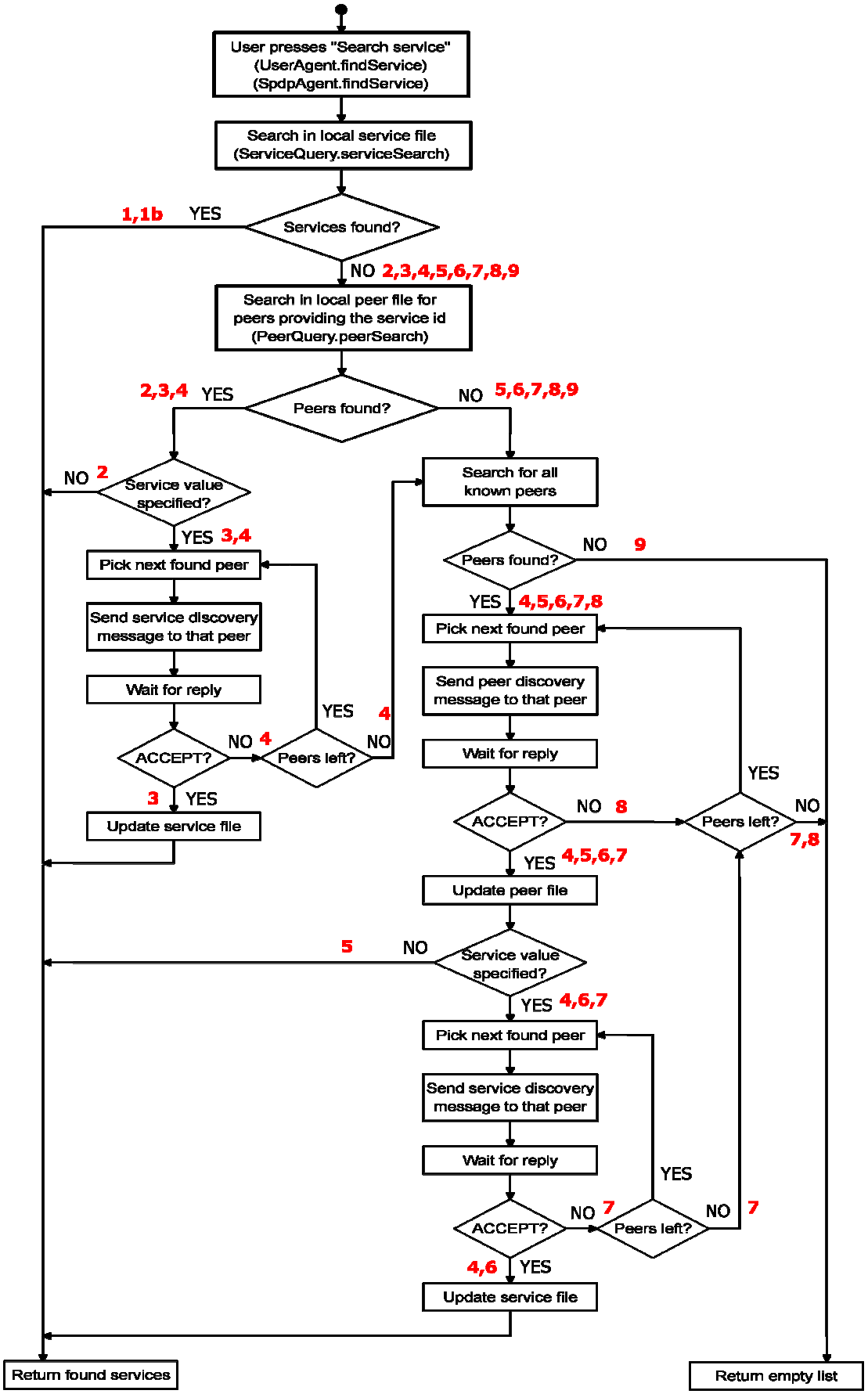

Figure 13: Control flow for the main thread for the ten test cases. The labels in the arrows correspond to test numbers.

Test	diego's service file	diego's peer file	Service id	Service value
1	=	=	filemp3	The Spring.mp3
1b	=	=	filemp3	The Autumn.mp3
2	=	=	sipphone	*(No value)*
3	*	=	sipphone	Pojken
4	*	*	sipphone	Gonzalo
5	=	*	printer	*(No value)*
6	*	*	printer	EasyPrint
7	=	*	printer	FrankInkStain
8	=	=	teletransport	Stockholm-Burgos
9	=	empty.xml	teletransport	Stockholm-Burgos

=: use default file (peers-diego.xml or services-diego.xml, see Apendix B)
*: use default file, but it will be changed after the discovery (network knowledge acquired)

Table 1: Service and peer files and service ids and values used for the tests

4.2.1.2.1 Test 1

Test 1 is the simplest query that can be made to SPDP. It is a query for a service that is provided locally (in the same host where SPDP is running). The first service search in the service file finds a hit, and it returns the found service.

4.2.1.2.2 Test 1b

In Test 1b, a locally known service (that is, it is present in the service file) that is not locally provided is searched for. For SPDP, this case is equivalent to Test 1.

4.2.1.2.3 Test 2

In Test 2, a service id without service value is searched for. No matching services are found in the service file. However, since there are known peers that provide this service id, and since the service value is not specified, those peers are returned.

4.2.1.2.4 Test 3

In Test 3, an unknown service is searched for. As no matching services were found in the local service file, a peer search is conducted for peers providing this service id. A service discovery message is sent to each of these peers until one is found which provides the requested service value.

4.2.1.2.5 Test 4

Test 4 illustrates the most complex action flow that can occur after a service search is started. In this case, as in Test 3, no matching service is found in the service file, but some peers are known to provide the required service id. A service discovery message is sent to each of those peers asking them

for the specific service value. However, none of them provides the desired value. Therefore, a peer search must be performed to locate new peers that provide the service id. A peer discovery message is issued and sent to each of the known peers, until one of them replies with information about peers that match the condition. In that case, the newly-acquired peers are queried directly for the service value. One of them is found to provide the requested service value, and the search ends successfully.

Note that this iterative search is done with only one step depth (as explained in section 3.3.2), i.e., only peers that are known to a peer's already known peers are searched (i.e., only "friends of our friends", never "friends of our friends' friends"). However, after a service discovery is performed, the list of known peers grows with (some of) the peers that were two steps away (i.e., "some friends of our friends become friends of our own"), thus becoming one step closer.

4.2.1.2.6 Test 5

In Test 5, an unknown service id is searched for. No known peers provide the service id, so a peer discovery message is issued to each of the known peers looking for peers that provide that service id. Here, one of the peers knows a peer that provides the service id, and so the search ends successfully.

4.2.1.2.7 Test 6

Test 6 is an extension to Test 5. After some peers have been found that provide the previously unknown service id, they are queried for the service value. One of them is found to provide the requested service, and the search ends successfully.

4.2.1.2.8 Test 7

Test 7 is a variant of Test 6, with the sole difference that none of the new peers provides the service value that has been requested, and so the search returns without result. Again, note that even if no matching services have been found, more knowledge about the network has been acquired, which can result in a better chance of finding services and peers in subsequent searches or iterations of the same search.

4.2.1.2.9 Test 8

Test 8 is a search for a service id which is unknown to this peer as well as to all the known peers, and therefore returns without any result.

4.2.1.2.10 Test 9

Finally, Test 9 illustrates the extreme case in which an unknown service is searched for, but the local peer does not know any peers. The search ends without any result. Note that even in the case of Tests 8 and 9, where no new network knowledge is acquired with the search is performed, a second iteration of the search after some time may yield services, since the network knowledge of this peer's known peers, or of this peer itself, can grow (for instance, due to receiving discovery requests from other peers).

4.2.2 Discovery time and traffic measurements

In order to compare SPDP and SLP, seven out of the ten tests described above were performed for both protocols, and the discovery time and traffic generated have been compared. Tests 2 and 5 were not performed with SLP since the translation from SPDP service descriptions to SLP service URLs requires the presence of a service value. Furthermore, Test 9 was not performed because it was difficult to find a fair counterpart with SLP: since SLP uses multicast messages, it will automatically perform as if it knew about the existence of all peers listening to the SLP multicast group. The results obtained in the tests for both protocols are detailed next.

4.2.2.1 Test 1

Figure 14 shows the histogram of the discovery times for Test 1 for both protocols. As can be seen, both protocols perform comparably in terms of **discovery time**, although SLP has a slightly smaller average time (4.5 ms instead of 6.5 ms for SPDP).

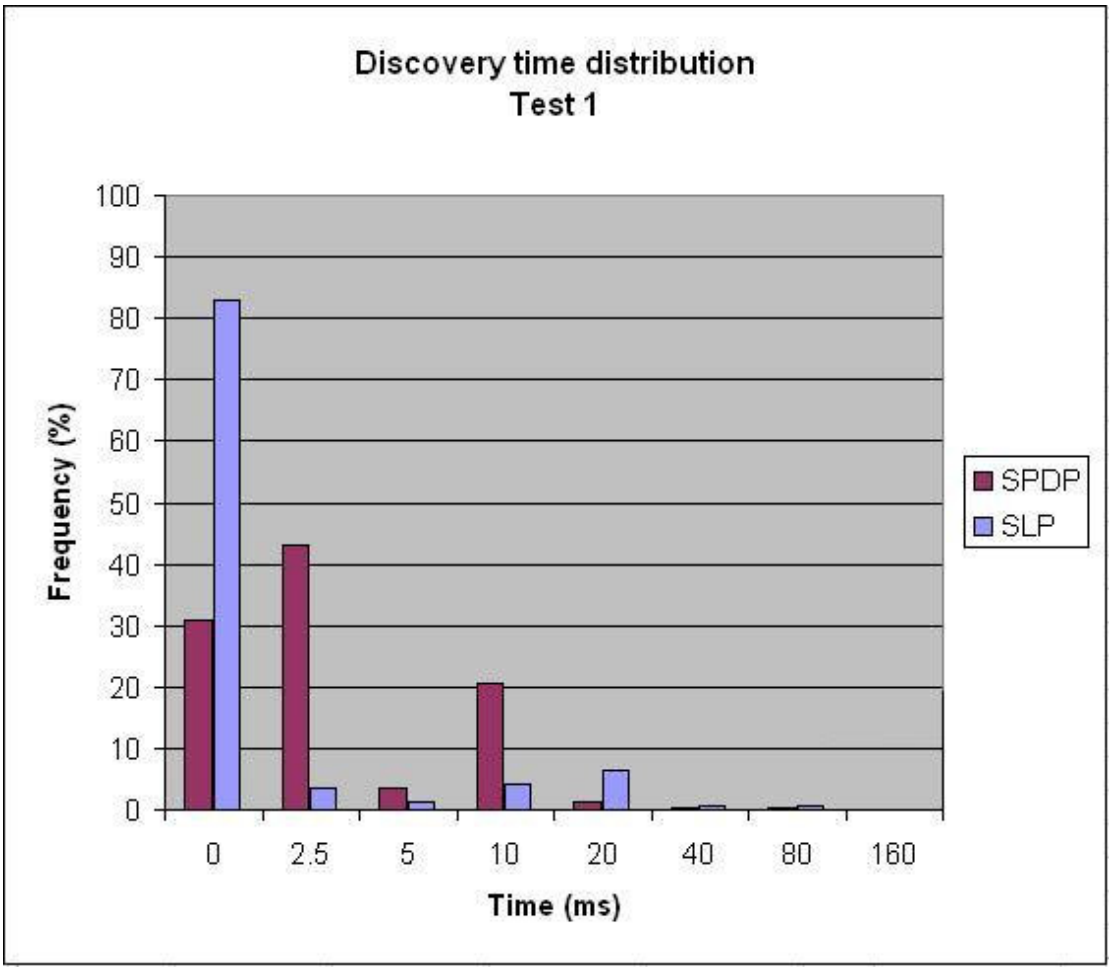

Figure 14: Discovery time distribution for Test 1 for SPDP and SLP

However, the operations that each of the protocols perform to accomplish the search are very different; while SPDP looks for the requested service in the service file and finds it, SLP issues a multicast message for it, which is received on the loopback interface and triggers a successful ServiceReply by the SLP daemon.

This difference becomes apparent when looking at the **traffic** measurements, which are summarized in Table 2 for a normal[2] Test 1 discovery. While SPDP sends no discovery message for known services, SLP issues a multicast ServiceRequest that reaches all the nodes in the network. As the service is locally provided, the ServiceReply message is generated internally and therefore does not reach the network (SLP nodes that do not provide a matching service do not respond to ServiceRequests). This is why the total traffic generated by SPDP is 0 bytes, while for SLP it is 100 multicast bytes in 1 message.

Note that this multicast message must be processed by all hosts listening to this multicast group despite the fact that the service was available locally.

Protocol	Bytes	Messages
SLP	100m	1
SPDP	0	0

Table 2: Traffic generated for the normal discovery for Test 1 by SPDP and SLP. "m" indicates multicast.

4.2.2.2 Test 1b

As stated before, Test 1b is equivalent to Test 1 for the SPDP protocol, since no distinction is made among services that are present in the service file. The **discovery time** distribution is therefore very similar to the one obtained for Test 1, and so is the average discovery time of 6.3 ms.

SLP has to send a ServiceRequest and wait for a ServiceReply in order to complete the search, but since the network introduces a negligible[3] delay, its performance in terms of discovery time is comparable to that of Test 1. The average discovery time measured was 3.9 ms. The service discovery time distribution for both protocols is depicted in Figure 15.

The results in terms of **traffic** are shown in Table 3. The difference between both protocols is more obvious for Test 1b than for Test 1, since SLP generates 2 messages (ServiceRequest and ServiceReply) for each discovery, amounting to 215 bytes (100 of which are multicast). SPDP does

[2] Throughout the measurements, the traffic comparison will be established for the normal discovery exchange, that is, a discovery where no messages have been lost or retransmitted.
[3] The transmission of 215 bytes over a 10 Mbps network takes 0.172 ms plus inter-frame spacing.

not generate any messages, since the service is contained in the service file and hence locally known.

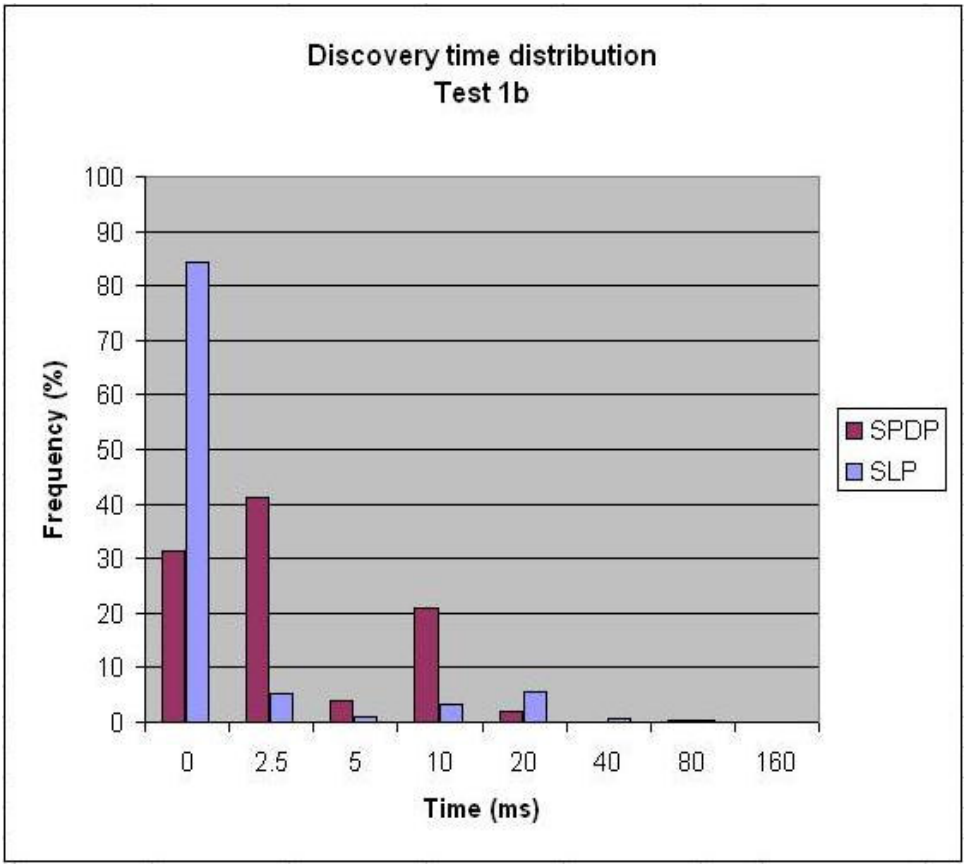

Figure 15: Discovery time distribution for Test 1b for SPDP and SLP

Protocol	Bytes	Messages
SLP	100m+115	2
SPDP	0	0

Table 3: Traffic generated for the normal discovery for Test 1b by SPDP and SLP

4.2.2.3 Test 3

Figure 16 shows the **discovery time** distribution for SPDP and SLP for Test 3. It can be seen that SLP outperforms SPDP in terms of discovery time. SLP follows the same strategy as in previous tests, sending a ServiceRequest message and waiting for a ServiceReply. SPDP needs additional message exchanges to perform a service discovery and thus takes longer. On average, SLP takes 4.1 ms, approximately the same time as in the previous tests, while SPDP takes 220 ms. The little peak in the 5120-10240 ms range for SPDP is due to a lost message during the

exchange; the maximum waiting time for messages was set to 9 s in the SPDP implementation.

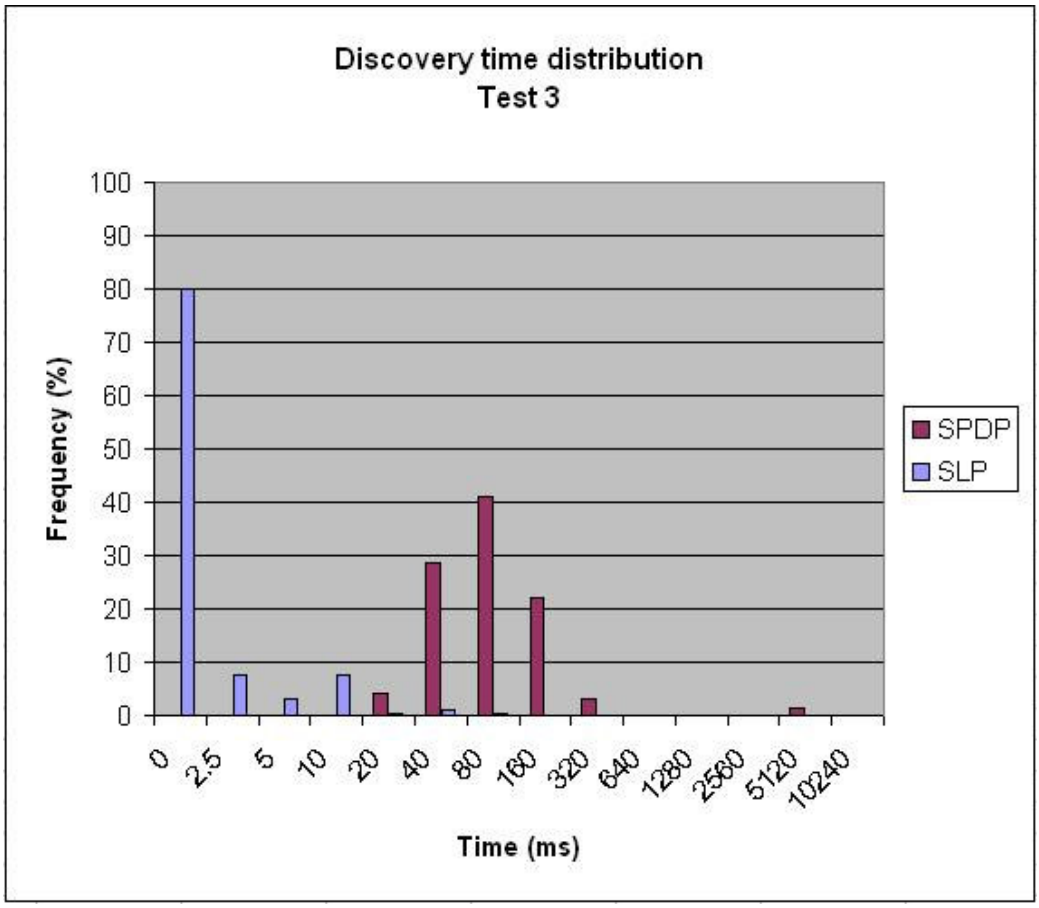

Figure 16: Discovery time distribution for Test 3 for SPDP and SLP

The **traffic** statistics for the normal discovery are shown in Table 4. Whereas a normal Test 3 discovery for SLP needs as much traffic as for Test 1b, SPDP must perform a service discovery, which normally consists of four messages:

- The SPDP DISCOVERY message for the requested service, embedded in a SIP SUBSCRIBE message.
- The SIP "200 OK" response to the subscription.
- The SPDP ACCEPT reply to the discovery, embedded in a SIP NOTIFY message.
- The SIP "200 OK" response to the notification.

In total, this amounts to 3378 bytes for the specific service request studied.

Protocol	Bytes	Messages
SPDP	3378	4
SLP	93m+108	2

Table 4: Traffic generated for the normal discovery for Test 3 by SPDP and SLP

On the other hand, SLP traffic is similar to the case of Test 1b, with one 93 byte long multicast ServiceRequest and one 108 byte long unicast ServiceReply. Again, it should be noted that the fact that SLP multicasts its messages does not change the number of messages exchanged because only one peer provides the requested service; however, it does have an influence in terms of processing power and battery consumption, since every peer in the network must receive and process the multicast messages

4.2.2.4 Test 4

As stated before, Test 4 illustrates the most complex message exchange that SPDP can perform in a single search iteration.

As Figure 17 shows, the average **discovery time** for SPDP is 920 ms, a time that is influenced by the lost messages that occur in some of the iterations of the experiment. The average for discoveries without any lost message is 405 ms.

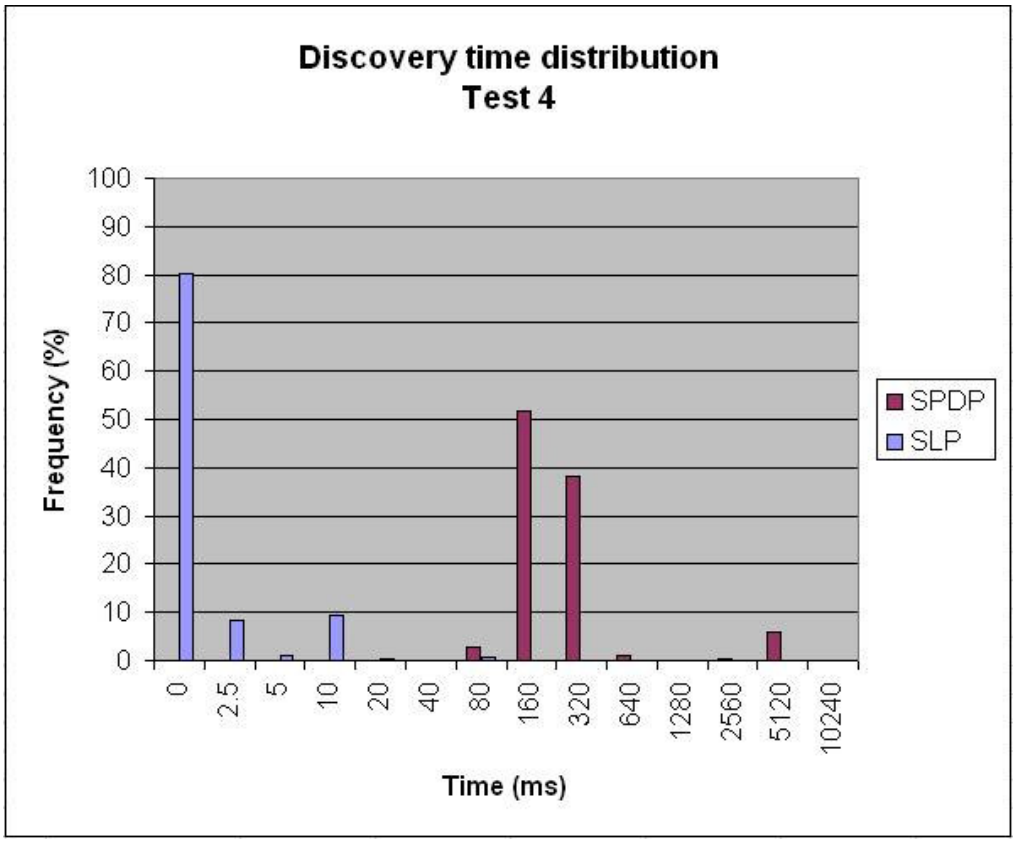

Figure 17: Discovery time distribution for Test 4 for SPDP and SLP

These values are high compared to SLP, which has an average discovery time of 4.0 ms, very similar to the previous cases.

With respect to **traffic**, SLP issues the same type of messages as in previous tests, whereas SPDP needs to perform a 12 message long exchange to accomplish a normal discovery. In detail, this message exchange consists of:

- The SPDP service discovery message for the requested service, embedded in a SIP subscription.
- The SPDP DENY response to the service discovery request, embedded in a SIP notification.
- The SPDP peer discovery message for peers providing the service name (SIP SUBSCRIBE).
- The SPDP ACCEPT response to the peer discovery request (SIP NOTIFY).
- The SPDP service discovery message asking the newly-found peer for the requested service (SIP SUBSCRIBE).
- The SPDP ACCEPT response for the final service request.
- Six SIP "200 OK" responses for each of the previous SIP messages.

This rather complex exchange results in 9868 bytes of unicast traffic exchanged over the network among the different peers.

In SLP the situation is the same as in the previous cases: 2 messages are sent, amounting to 203 bytes, 94 bytes of which are multicast. These results are summarized in Table 5.

Protocol	**Bytes**	**Messages**
SPDP	9868	12
SLP	94m+101	2

Table 5: Traffic generated for the normal discovery for Test 4 by SPDP and SLP

4.2.2.5 Test 6

The discovery time distribution for Test 6 for both protocols is depicted in Figure 18. The histogram shows that, as expected, SLP performs as for the previous cases (the average discovery time for SLP is 4.4 ms), since its strategy is the same as in all the previous tests.

SPDP requires an average of 635 ms to perform a discovery, which is reduced to around 265 ms for discoveries without lost messages.

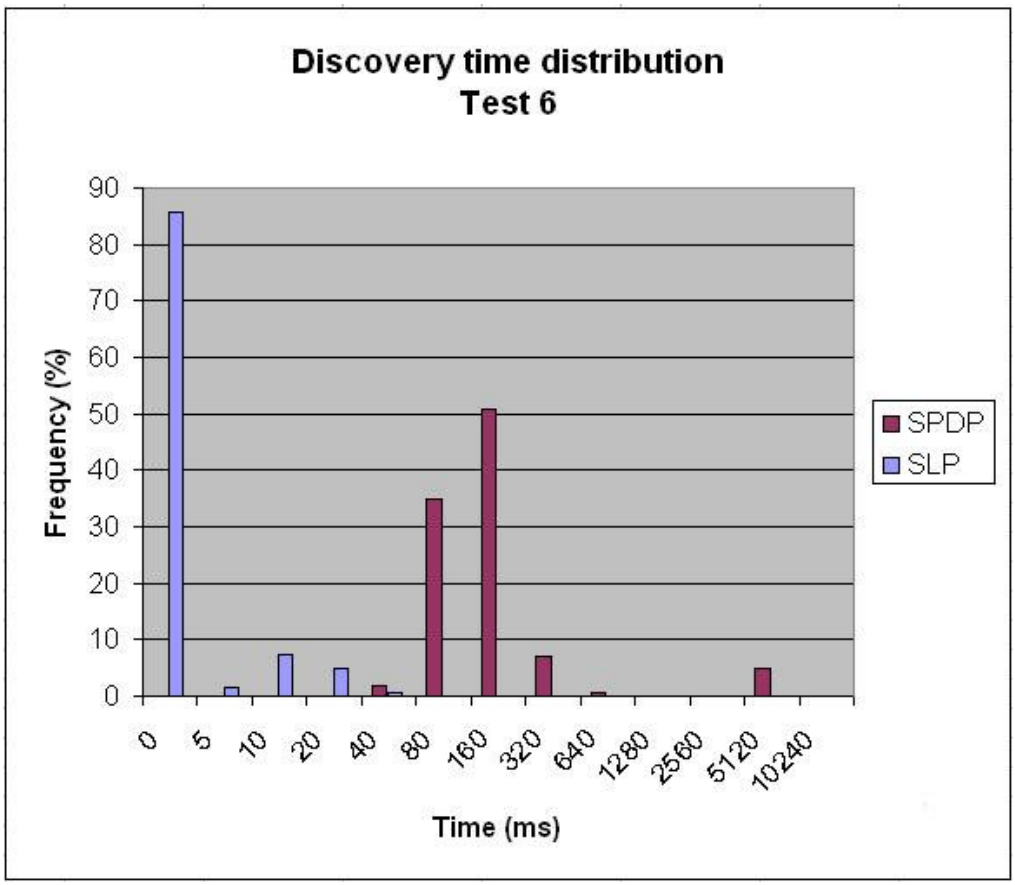

Figure 18: Discovery time distribution for Test 6 for SPDP and SLP

The **traffic** exchange for both protocols is summarized in Table 6. For SLP, the normal discovery consists of a simple message exchange: a multicast ServiceRequest and a unicast ServiceReply. For SPDP, an 8 message long exchange is performed:

- An SPDP peer discovery message for peers providing the service name.
- An SPDP ACCEPT response for the peer request.
- An SPDP service discovery message for the actual service value.
- An SPDP ACCEPT response for the service request.
- Four SIP "200 OK" responses, one for each of the SIP messages that embed the SPDP messages.

Protocol	Bytes	Messages
SPDP	6877	8
SLP	95m+110	2

Table 6: Traffic generated for the normal discovery for Test 6 by SPDP and SLP

4.2.2.6 Test 7

Unlike the previous tests, Test 7 illustrates a search that ends without any hits. This has an important influence in the "discovery" time distribution, measured as the time that it takes for each protocol to consider a service unavailable or inexistent. This **time** distribution is shown in Figure 19.

SLP has a timer (set by default to 3 seconds in the implementation under study) to wait for ServiceReplies. The retransmission protocol implemented retries twice until it gives up. This is why all the service discovery times are in the range 3200 to 6400 ms for SLP. The average time was measured to be 6020 ms.

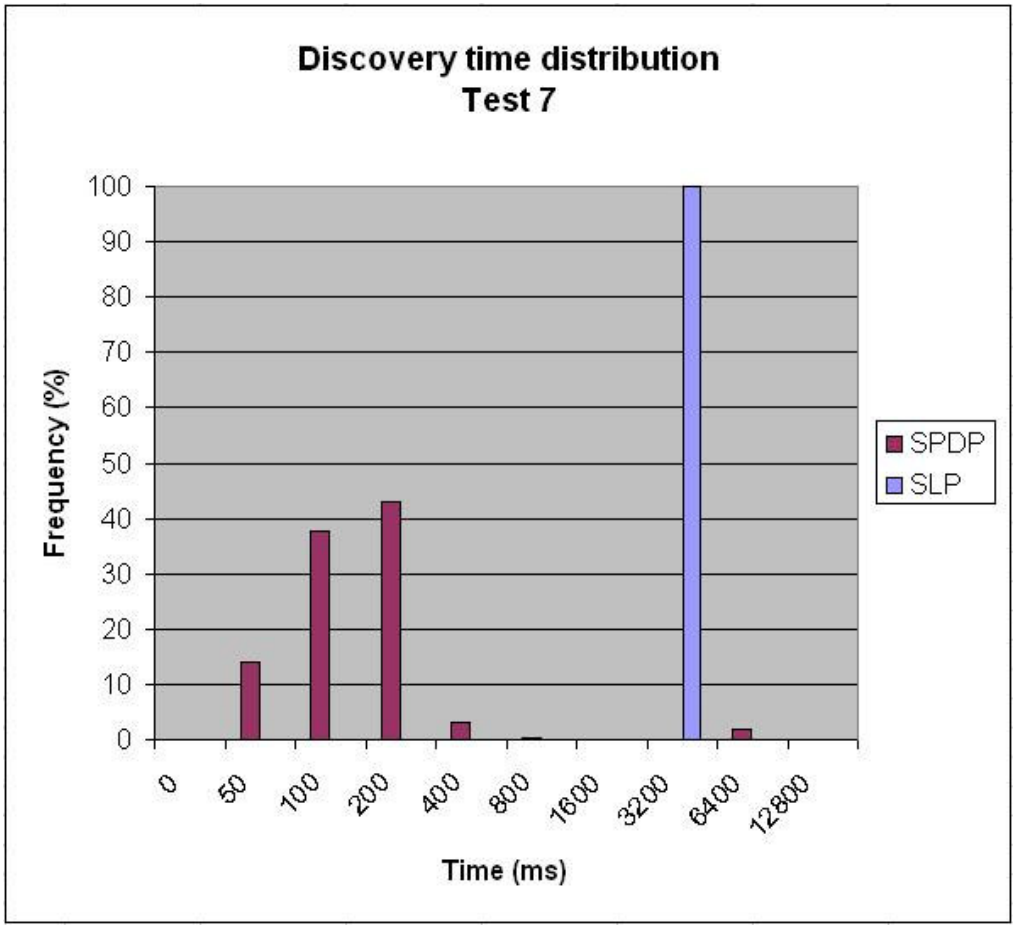

Figure 19: Discovery time distribution for Test 7 for SPDP and SLP

On the other hand, SPDP gives up searching when it has asked all the known peers for new peers, when it has asked these new peers for the requested service, but still got no successful response. This takes, on average, around 360 ms (around 200 ms on average if no packets are lost) in our study case. This time has a constant component due to the initiation of the search, and a component that is expected to grow linearly with the number of known peers.

The **traffic** exchange involved, summarized in Table 7, shows how both protocols work. While SLP multicasts three ServiceRequests, and then returns, SPDP sends the following messages:

- An SPDP peer request for new peers that provide the service id.
- An SPDP ACCEPT response to the peer request.
- An SPDP service request for the actual service value.
- An SPDP DENY response for the service request.
- Four SIP "200 OK" responses, one for each of the previous four SPDP messages.

Protocol	Bytes	Messages
SPDP	6489	8
SLP	297m	3

Table 7: Traffic generated for the normal discovery for Test 7 by SPDP and SLP

4.2.2.7 Test 8

Test 8 looks for a service id that is unavailable in the network. Both in terms of discovery time and traffic, this test is equivalent to Test 7 for SLP, since a ServiceRequest message is sent that receives no reply. It can again be seen that the **discovery time** distribution, depicted in Figure 20, has a single peak in the 3200 to 6400 ms region for SLP. In fact, the average time is 6035 ms.

The SPDP message exchange is simpler in this case than for Test 7, since no peers were found that provide the service id, and the search ends earlier. The average discovery time for SPDP is 179 ms (119 ms for exchanges without lost messages).

As far as **traffic** is concerned, SPDP involves a message exchange consisting of 4 messages for the normal case:

- The SPDP peer discovery message for peers providing the service name.

- The SPDP DENY response for the peer request.
- Two SIP "200 OK" responses, one for each of the two SPDP messages.

As in Test 7, SLP traffic consists of three multicast ServiceReplies for the requested service. The average numbers for both protocols are shown in Table 8.

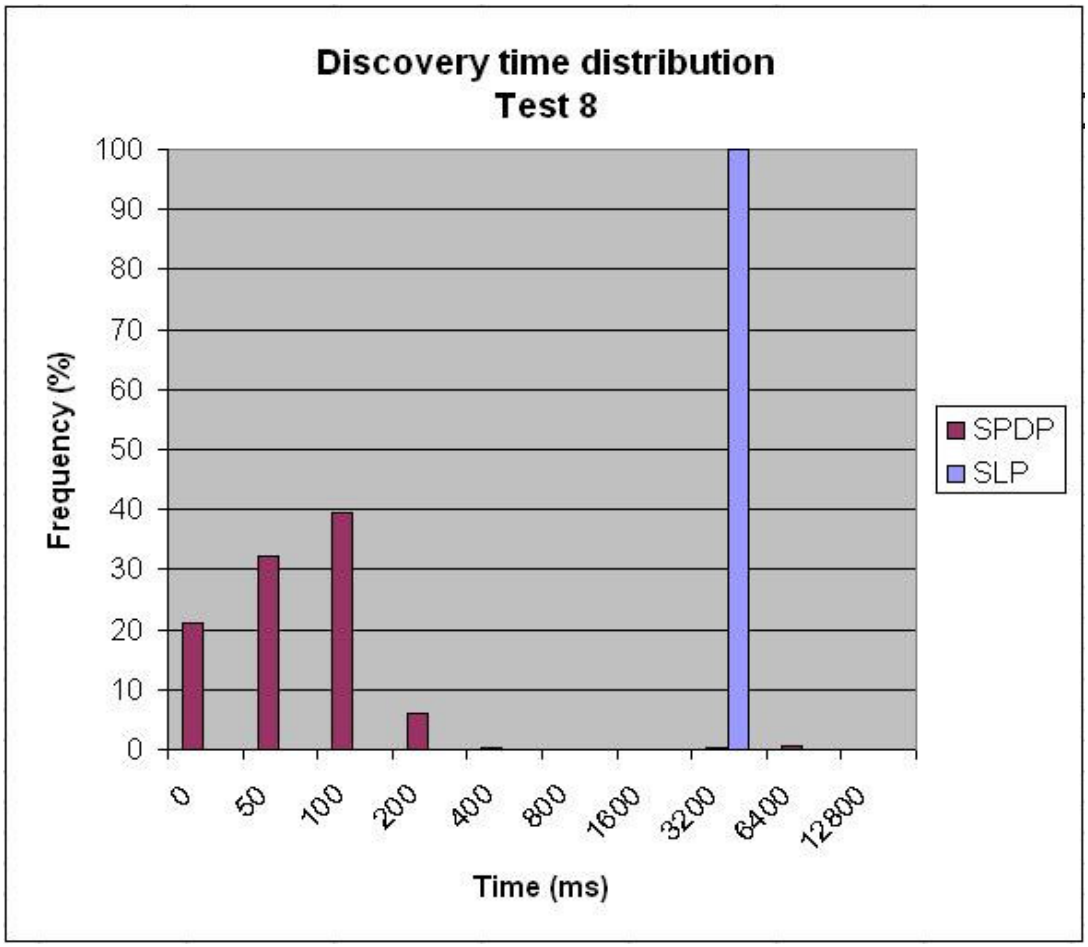

Figure 20: Discovery time distribution for Test 8 for SPDP and SLP

Protocol	Bytes	Messages
SPDP	3228	4
SLP	324m	3

Table 8: Traffic generated for the normal discovery for Test 8 by SPDP and SLP

4.2.3 Conclusions

From the results of the tests described above, some preliminary conclusions can be made about the performance of SPDP in comparison with SLP.

4.2.3.1 SLP is more traffic and time efficient in general

The tests show that SLP is, in general, a more time and traffic efficient protocol than SPDP. It uses shorter and in general fewer messages for discoveries, and it takes less time to complete the searches.

SLP is a protocol that works much closer to the operating system than SPDP. The SLP daemon runs as a background process that is in charge of managing all SLP local service information and message exchanges. Furthermore, SLP uses the transport layer (UDP in the case of the OpenSLP implementation) directly, while SPDP uses SIP to transport its messages. Last but not least, SLP uses extended URLs to describe services, while SPDP uses XML-formatted service descriptions. All of this explains why SPDP requires a longer time and generates more traffic for its discoveries.

4.2.3.2 Considerations with respect to traffic

When comparing both protocols in terms of traffic, it must be noted that SLP makes extensive use of multicast messages to perform discoveries. It is therefore not straightforward to state which protocol performs more efficiently in terms of traffic for all cases. In network technologies where there is a shared media, such as all the LAN variants, using multicast does not involve sending more messages if the communicating peers are in the same subnet, since the shared media allows for multicast and broadcast naturally. But if that is not the case, a multicast message can result in more traffic being generated, depending on the size of the SLP network.

Some more important considerations about multicast in the peer-to-peer wireless scenario are made in section 4.2.3.5.

4.2.3.3 SPDP traffic saving for known services

However, SPDP does outperform SLP in terms of traffic for two relevant cases, those tried in Tests 1 and 1b. In those cases, SPDP's time performance is also comparable to that of SLP.

When searching for known services (whether they are locally provided or present in the service file), SPDP does not require any message to be sent to the network, since the search in the local file produces a hit. This can be a crucial advantage in some networks (such as for instance GPRS) where traffic is very costly.

The importance of these cases greatly depends on the frequency with which a service discovery of each type occurs. It can be assumed to be unlikely that a user chooses to ask the service discovery protocol for a service that is provided in the same peer, as in Test 1. But the probability that an already known service is requested (as in Test 1b) can be high if the spread of information around the network is efficient. While it is not trivial to design

an algorithm that would maximise the probability of having the requested services stored in the local file while also minimising the amount of traffic generated, a step in this direction would be to take advantage of every SPDP message sent to spread network knowledge, including some amount of **un**requested peer and service information, thus amortising part of the message overhead, and increasing the probability of future successful searches.

4.2.3.4 Considerations of non-successful searches

The evaluation of the protocols when dealing with searches that do not find any match must be different from the successful cases. The fact that a protocol returns an empty service list very quickly does obviously not make it better, especially if its counterpart can produce a non-empty list after a slightly longer wait. However, it is certainly desirable that a search for an inexistent or unavailable service takes up as little time and generates as little traffic as possible.

A number of different factors have to be taken into account before being able to state that a protocol performs better or worse than another in these cases. Firstly, the scope of the searches is different in both protocols. SLP searches inside the scope of the SLP multicast group, whereas SPDP queries peers that are known locally and those that are known to them (that is, two steps of "acquaintance"). There is a trade-off between a larger scope, which could possibly yield more hits, and more discovery traffic, since more discovery messages will have to be sent.

Another factor to take into consideration is the actual gain in terms of spreading network knowledge that is obtained even if no hits are found. In SPDP, for instance, new or updated information about peers may be obtained even if the requested service cannot be found, and information about the requesting peer is spread with every message it sends, possibly resulting in better chances for a (local) hit in subsequent searches.

4.2.3.5 Considerations of the peer-to-peer wireless scenario

The goal of this thesis was to study the performance of SPDP in a particular scenario, namely the peer-to-peer wireless scenario (described in section 2.1).

It has been mentioned that SPDP's use of XML both for the representation of peer and service information and for the format of its messages increases the size of the protocol messages. However, this approach provides SPDP with much greater flexibility and scalability for the representation and the search for services than SLP. The fact that SPDP uses XML allows for greater understandability and a standard representation of the information, and provides a powerful means to make searches with multiple levels of refinement, hence making it more expressive than SLP's service URLs and attributes. It also makes it easier to extend the protocol with new functionality. SPDP's designers consider that it is worth paying the extra price in terms of traffic in order to create a more user-oriented, high-level

representation of the information, in accordance with the user-oriented peer-to-peer wireless scenario. Furthermore, the use of XML also facilitates distributing information along with each query.

Note that traffic can be reduced by using schemes for compression which can exploit a preloaded dictionary [44, 45].

A key feature of the scenario studied is the role of SIP as an overall signalling protocol for all kinds of session-oriented services. In this context, SPDP has the advantage that it has been designed to work with SIP, and is therefore very smoothly integrated as a new functionality for a SIP User Agent, as shown earlier (see section 3.1.2).

Finally, consideration has to given to the type of messages used by both protocols. SLP's approach to service discovery heavily relies on the possibility of using multicast in the network. The SLP specification suggests that multicast messages should be used by User Agents to discover services in smaller networks, where no Directory Agent is present. In larger networks, even if the communication is unicast to and from the Directory Agent, multicast messages are still used as DA advertisements or DA service requests. SLP can work in networks with a Directory Agent and without multicast, but it was not designed for that purpose. On the other hand, SPDP only relies on unicast messages for its discoveries. Unicast is seen as the standard type of message for the protocol, and multicast is only contemplated as a possibility that could be taken advantage of when it is available (for instance, for the initial acquisition of information on start-up). This philosophy is better targeted for the peer-to-peer wireless scenario under study, in which some network technologies, such as GPRS do not allow for broadcast or multicast messages.

Moreover, the use of multicasting has crucial implications in the peer-to-peer wireless scenario, where a considerable fraction of all peers are battery-powered mobile devices. Even if a multicast message does not trigger a response from a peer, the peer must receive and process it, which results in some battery and processing power consumption. A protocol based in unicast messages ensures that only the intended recipient processes the message and issues a response.

4.3 SPDP in real networks

While the experiments described in the previous section permit us to make a number of interesting conclusions about the performance of SPDP, they are unrepresentative of the real settings in which SPDP is likely to be used. A **greater number of peers**, greater flexibility for the peers to **change or interrupt their services**, and the possibility that **peers enter or exit** the network at any time are some of the usual events that the protocol will have to deal with, and which will ultimately determine its performance and usefulness.

It has been shown that SPDP, unlike SLP, uses an algorithm based on unicast to search for services in the network. Considerations have been given to the influence that this has on the amount and type of traffic generated by both protocols. But SPDP's algorithm also specifies that only a limited set of peers (as explained earlier, the known peers and the peers known to the known peers) is queried for services in every search, thus resulting in a **probabilistic search**: there is a certain probability that a service is actually available in the network, but a peer will not be able to find it using SPDP (in the first search iteration), since the peer providing the service is too "far" in the knowledge graph. This is not the case for SLP, which uses multicast, and will therefore be able to find any service provided by a peer that listens to the SLP multicast group.

It is therefore interesting to evaluate this error probability, since it will determine the performance of SPDP. First of all, it is clear that this probability depends on the average number of **peers that each peer knows** with depth two[4]. If every peer knows, through its known peers, every other peer, then the search performed by SPDP will not be constrained; it will encompass the whole peer network. How likely is this the case?

In a real network, there are three types of events that influence the average "network knowledge" of the peers. If a **new peer comes** into the network, and assuming it does so by learning of the existence of one other peer, the average network diameter (and knowledge) decreases, since this peer is logically tied to only one other peer. If a **peer exits the network**, the peers that knew it now know one peer less (in fact, they will eventually do so after the timers for the entries in the respective service and peer files expire), thus decreasing the average network knowledge. The way to increase network knowledge is by **performing searches**, by which updated network knowledge spreads.

The rather complex nature of the SPDP algorithm makes it difficult to quantify precisely how much knowledge is gained or lost on average when any of those events occur. The knowledge increase/decrease in each case depends on the particular content of the peer and service files of each of the peers involved in the search. However, it is safe to say that, while a peer departure or arrival adds an amount of "uncertainty" to the network knowledge corresponding to one specific peer, a single search may create logical bindings among several different peers.

It is not obvious how to quantify the frequency with which each of these three events occurs. Several measurements have been carried out about the behaviour of users of wireless networks deployed in university campuses [46, 47], conferences [48], corporate networks [49] and metropolitan networks [50]. Those measurements try to establish patterns

[4] This number is implementation specific, but as discussed earlier in section 3.3.2, there are good reasons to choose it.

and estimate, among other things, the distribution of the time the users are connected to the network, the arrival and departure processes of users in the network, and the frequency distribution of the applications that are used. However, these statistics do not have straightforward application to the SPDP analysis: the arrival of a new user in the wireless network has a very different effect on the peer knowledge if the user has kept some of the knowledge it acquired during previous sessions, or if it enters the network for the first time; the fact that a user roams to another network or completely exits the wireless network does not necessarily mean that the peer will not be reachable through SIP any more, and it may still take part in searches and therefore contribute to the average knowledge of the network; finally, the measured distribution of the popularity of the different applications present in the wireless network needs to be extrapolated to the particular services offered in a peer-to-peer wireless scenario.

Nevertheless, it is sensible to assume that, in a real scenario, service searches will occur considerably more often than peer arrivals or departures; usually, a peer will require several services during a session, from the moment the session begins until it finally ends.

One important feature of the peer-to-peer wireless scenario is the presence of a few fixed peers in the network, along with several highly mobile peers. These fixed peers (context servers, access points,…, but also printer peers, fax peers, etc. present in a room) can constitute a backbone of network knowledge, since each of them is very likely to know all of the others which are relatively fixed, in a full mesh, assuming that they have been running for a long time[5].

All of the previous considerations indicate that it is possible to assume that the average network knowledge of the peers in the network is probably high, and therefore the probability of not finding an available service can be assumed small.

It has been mentioned that an SPDP peer learns with every discovery, particularly with the unsuccessful ones. The way in which information is spread ensures that a user that does not get tired of iterating the same search **will** eventually find a match, if the service exists in the network. But of course users get tired, and some do so very soon. Hence, another interesting statistic is the number of iterations of the same search that will be needed, on average, to find an available service.

As stated before, with respect to network knowledge, the best case topology for SPDP is a full mesh, i.e., when every peer knows every other peer in the network (see Figure 21). In this case, the probability of not finding an available service is zero, and there is no need to perform a second iteration of the search. Analogously, the worst case would be if the network knowledge topology is a line of peers, in which it is easy to see that

[5] This can of course be ensured by providing each of the fixed peers with static information about the rest of this static backbone.

the probability of not finding an available service is $\frac{N-3}{N}$, where *N* is the total number of peers, and that $\frac{N}{2}-1$ iterations of the same search will be needed on average to find the match.

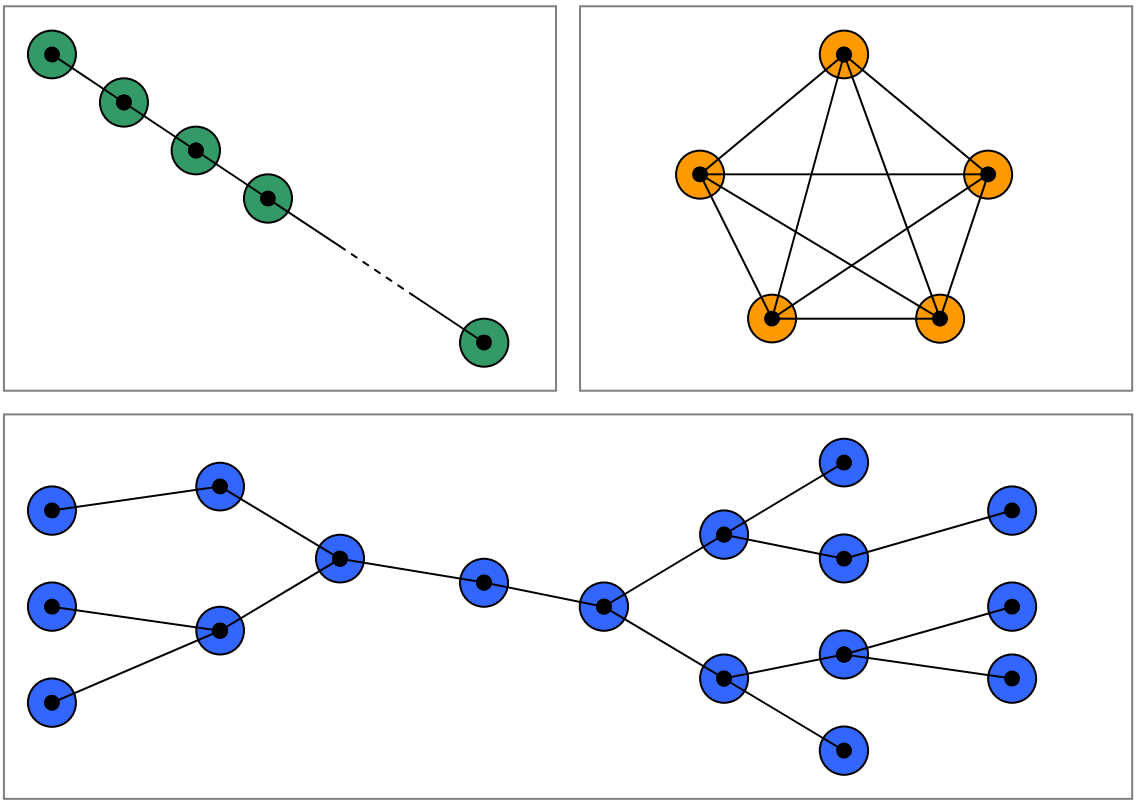

Figure 21: Network knowledge topologies: line (top left, worst case), full mesh (top right, best case), random binary tree (bottom)

A line is a particular case of tree topology, with a branching factor of 1. It can be seen that just by increasing the branching factor of the tree to 2, thus making it binary, the average number of iterations needed is drastically reduced. In fact, [51] proves that the average depth of a random binary tree is approximately 2·ln(*N*), where *N* is the number of nodes in the tree[6]. In the case where there are thousands of peers connected to the network, which is a reasonable worst case, an upper bound for the average depth would therefore be 2·ln(1000), that is, around 14. This means that **every** peer can be reached after an average of 12 iterations.

[6] In fact, the obtained expression [51] for the average depth is:

$$\bar{d} \approx 2\left(\frac{N+1}{N}\right)(\ln N + \gamma) - 4$$

where γ is Euler's constant.

However, in real networks, the presence of fixed peers and the activity of peers in terms of performing discoveries will likely make the network knowledge topology much closer to a mesh than a tree, further decreasing the search depth and increasing the probability of a successful search.

All in all, it can be stated that SPDP is well targeted towards the peer-to-peer wireless scenario, where it benefits from the environment in order to provide a battery and traffic-efficient service search mechanism without having to pay a high price in terms of incompleteness of the search.

5 Improvements and future work

After creating an implementation and evaluating the Service Peer Discovery Protocol, this chapter contains suggested improvements to the protocol that can constitute a follow-up to this master's thesis. Here they are described in order, from the most specific and concrete, to the most abstract and general.

5.1 Optimization of the implementation

As stated before, the first goal of this project was **not** to create an efficient all-round implementation of SPDP, but rather an illustrative working implementation that showed the main features of the protocol and allows a fair comparison with other protocols, SLP in our case.

This is why extensive documentation has been written about the implementation both inside this document and as Javadoc pages. Hopefully, the documentation will aid future developers of SPDP to produce a more targeted, efficient implementation of the protocol.

5.2 Redundancy in the protocol messages

In the comparison with SLP (section 4.2), it became evident that SPDP messages are larger that SLP messages. However, some of the fields in the SPDP message carry redundant information that is already present in the headers of the embedding SIP message. For instance, the **sender** SPDP field can be identified with the **From** SIP header field, and both messages contain a **replyTo** field that is very likely the same.

The implementation described in this document chose to keep this redundancy for simplicity: in this way, SIP is a mere carrier of SPDP messages, which are completely self-contained. But in order to improve the performance of the protocol, the information carried in the SIP header could be used to remove redundant fields or to provide extra functionality.

5.3 Taking advantage of the context servers

Currently, the implementation is designed to work in networks without a context server. The implementation **will** work if there is a context server present in the network, but it will not take advantage of its presence: the context server will be treated as a normal peer. However, a context server is likely to aggregate a lot of service and peer information, and therefore it should be taken advantage of when it is present in the network.

As future work, the implementation could be improved so that the presence of a context service is checked by every peer upon entrance to the network, and so that context servers are prioritized over the rest of the peers when searching for peers or services (queries should be sent to the context servers first). If the context server is provided with a smart way of passively acquiring network knowledge with every request received, its presence could largely improve the traffic and time performance of SPDP.

5.4 Taking advantage of XML

As explained before, SPDP uses the Extensible Mark-up Language (XML) to represent all the data structures of the protocol. Compared to the descriptions that other protocols make of the services (such as SLP's *service URLs*), SPDP provides an almost infinite wealth of possibilities both to represent services in all necessary detail and to exchange important service information in the messages, and these possibilities should be studied to make the most of the protocol architecture.

One way in which this could be done was suggested in section 4.2.3.3. In order to improve the spread of information through the network, and thus increase the probability of finding services locally, unrequested service and peer information could be included in the protocol messages. It is left as future work to implement this improvement, as well as to determine which information to include in order to maximise the probability of future local matches with the smallest possible traffic increase.

Of course, other criteria could be followed to determine which information to include in SPDP messages: peers may choose to advertise in order to attract users to their services, or to recommend services that the user liked,… The possibilities are numerous.

5.5 SPDP in a real setting

More conclusions and suggestions for improvements are bound to come up when the protocol is used in a real setting. This thesis was done in the framework of the Adaptive and Context-Aware Services (ACAS) project [6]. The prototypes developed in this project could very well be the appropriate test bed for the protocol.

Related work inside the ACAS project was performed by Konstantinos Avgeropoulos [52]. In his proposed architecture for policy management, SPDP can be the means by which the different agents discover and share information with each other, and with the service manager.

References

[1] P. Jarske. *The GSM System, Principles of Digital Mobile Communication Systems*, Technical report, Technical University Tampere, Finland, 2001.

[2] IEEE 802.11. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 1999 Edition.

[3] G. Söderström. *Virtual Networks in the Cellular Domain*. Master's thesis, Royal Institute of Technology (KTH), Department of Microelectronics and Information Technology, February 2003.

[4] R. Cascella. *Reconfigurable Application Networks through Peer Discovery and Handovers*. Master's thesis, Royal Institute of Technology (KTH), Department of Microelectronics and Information Technology, June 2003.

[5] J. Rosenberg, et al. *SIP: Session Initiation Protocol*. RFC 3261, IETF, June 2002.

[6] ACAS project, http://psi.verkstad.net/acas

[7] E. Guttman, C. Perkins, J. Veizades, and M. Day, *Service Location Protocol, Version 2*. RFC 2608, IETF, June 1999.

[8] whatis.com, *Peer-to-peer, a searchNetworking definition*, http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci212769,00.html, accessed November 2003.

[9] Napster, *Napster – About us*, http://www.napster.com/about_us.html, accessed November 2003.

[10] Gnutella, http://www.gnutella.com/, accessed November 2003.

[11] Kazaa, http://www.kazaa.com/us/index.htm, accessed November 2003.

[12] C. Perkins. *IP Mobility Support for IPv4*. RFC 3344, IETF, August 2002.

[13] R. Appleton. *The Kazaa Protocol*, http://cs.nmu.edu/~randy/Classes/CS442/The_Kazaa_Protocol.html, February 2003.

[14] FasttrackHelp Forums, *What are Supernodes?*, http://www.fasttrackhelp.com/forums/index.php?showtopic=3782, accessed November 2003.

[15] FTFakes, *My Kazaa tools*, http://www.fasttrackhelp.com/development/ftfakes/, accessed November 2003.

[16] Clip2, *The Gnutella Protocol Specification v0.4*, http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf, accessed November 2003.

[17] drscholl@users.sourceforge.net, *Napster messages*, http://opennap.sourceforge.net/napster.txt, April 2000.

[18] The Internet Engineering Task Force (IETF), http://www.ietf.org

[19] C. Bettsetter and C. Renner, *A comparison of service discovery protocols and implementation of the Service Location Protocol*, Technische Universität München (TUM), Institute of Communication Networks, http://citeseer.nj.nec.com/cache/papers/cs/16108/http:zSzzSzwww.lkn.ei.tum.dezSz~chriszSzpublicationszSzeunice2000-slp.pdf/bettstetter00comparison.pdf, accessed November 2003.

[20] R. Droms, *Automated configuration of TCP/IP with DHCP*, IEEE Internet Computing 3(4):45-53, July 1999.

[21] C. E. Perkins and E. Guttman, *DHCP options for Service Location Protocol*, RFC 2610, June 1999.

[22] Sun, Technical White Paper: Jini Architectural Overview, http://www.sun.com/jini/, 1999.

[23] Java, http://java.sun.com/ , Accessed November 2003.

[24] R. E. McGrath. *Discovery and Its Discontents: Discovery Protocols for Ubiquitous Computing*. Presented at Center for Excellence in Space Data and Information Science, NASA Goddard Space Flight Center, USA, April 5 2000.

[25] Microsoft Corporation, Universal Plug and Play Device Architecture, http://www.upnp.org, June 2000.

[26] Microsoft, http://www.microsoft.com, accessed April 2004.

[27] Y. Goland, T. Cai, P. Leach, Y. Gu, and S. Albright. *Simple Service Discovery Protocol/1.0. Draft,* draft-cai-ssdp-v1-03.txt, IETF, October 1999.

[28] S. Aggarwal, J. Cohen, and Y. Goland. *General Event Notification Architecture Base: Client to Arbiter. Draft*, http://www.upnp.org/download/draftcohen-genaclient-01.txt , September 2000.

[29] D. Box, et al. Simple Object Access Protocol SOAP 1.1. Draft, W3C, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/ , May 2000.

[30] R. Troll. *Automatically Choosing an IP Address in an Ad-Hoc IPv4 Network. Draft* draft-ietf-dhc-ipv4-autoconfig-05.txt, IETF, March 2000.

[31] JXTA community, http://www.jxta.org, accessed November 2003.

[32] *Peer Discovery Protocol*, JXTA v2.0 Protocols Specification, http://spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.html#proto-pdp, accessed November 2003.

[33] H. Schulzrinne and E. Wedlund. *Application-Layer Mobility using SIP*. Mobile Computing and Communications Review MC2R, vol. 4(3):47–57, July 2000.

[34] E. Wedlund and H. Schulzrinne. *Mobility Support Using SIP*. In Proceeding of Second ACM/IEEE International Conference on Wireless and Mobile Multimedia WoWMoM99, Seattle Washington, USA, August 1999.

[35] A. B. Roach. Session Initiation Protocol SIP - Specific Event Notification. RFC 3265, IETF, June 2002.

[36] G. Mola. *Interaction of Vertical Handoffs with 802.11 wireless LANs: Handoff Policy*. Master's thesis, Royal Institute of Technology (KTH), Department of Microelectronics and Information Technology, To appear.

[37] W3C, Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, 6 October 2000. http://www.w3.org/TR/REC-xml.

[38] J. Rosenberg, and H. Schulzrinne. *SIP: Locating SIP Servers*, RFC 3263, June 2002.

[39] A. Jarrar. *Context Server Support for Opportunistic and Adaptive Mobile Communication*. Master's thesis, Royal Institute of Technology (KTH), Department of Microelectronics and Information Technology, To appear.

[40] Java Architecture for XML Binding (JAXB), http://java.sun.com/xml/jaxb/, accessed April 2004.

[41] CSLAB's Java SIP User Agent by Ericsson, version 3.2. Experimental SIP-implementation (written largely in Java and ported to Windows, Solaris, Linux) provided by Ericsson Research for the ACAS pilot project (http://psi.verkstad.net/acas).

[42] JXTA demo, http://download.jxta.org/easyinstall/install.html, accessed April 2004.

[43] OpenSLP, http://www.openslp.org/, accessed April 2004.

[44] M. Garcia-Martin, C. Bormann, J. Ott, R. Price and A. B. Roach, *The Session Initiation Protocol (SIP) and Session Description Protocol (SDP) Static Dictionary for Signaling Compression (SigComp)*, RFC 3485, February 2003.

[45] G. Camarillo, *Compressing the Session Initiation Protocol (SIP)*, RFC 3486, February 2003.

[46] D. Kotz and K. Essien. *Analysis of a campus-wide wireless network*. In ACM MobiCom, September 2002.

[47] M. McNett and G. M. Voelker. *Access and Mobility of Wireless PDA Users*. Mobile Computing and Communication Review, Volume 7, Number 4, October 2003.

[48] A. Balachandran, G. Voelker, P. Bahl, and P. Rangan. *Characterizing user behaviour and network performance in a public wireless lan*. In ACM Sigmetrics, 2002.

[49] M. Balazinska and P. Castro. *Characterizing mobility and network usage in a corporate wireless local-area network*. In ACM MobiSys, 2003.

[50] D. Tang and M. Baker. *Analysis of a metropolitan-area wireless network*. In Wireless Networks V. 8, pages 107-120, 2002.

[51] B. R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley and sons. Available at http://www.brpreiss.com/books/opus4/html/page307.html. 1997.

[52] K. Avgeropoulos. *Service Policy Management for User Centric Services in Heterogeneous Mobile Networks*. Master's thesis, Royal Institute of Technology (KTH), Department of Microelectronics and Information Technology, March 2004.

Appendices

A List of acronyms

ACAS	Adaptive and Context-Aware Services
DA	Directory Agent
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name Service
FTP	File Transfer Protocol
GENA	General Event Notification Architecture
GPRS	General Packet Radio Service
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
J2SDK	Java 2 Software Developer Kit
JAXB	Java Architecture for XML Binding
JVM	Java Virtual Machine
JWSDP	Java Web Services Developer Kit
LAN	Local-Area Network
MCU	Multipoint Control Unit
MEGACO	Media Gateway Control Protocol
MP3	MPEG audio layer 3
NAPT	Network Address Port Translation
NAT	Network Address Translation
P2P	Peer-to-Peer
PDP	Peer Discovery Protocol
PMP	Peer Membership Protocol
RFC	Request For Comments
RMI	Remote Method Invocation
RTP	Real-Time Protocol
RTSP	Real-Time Streaming Protocol
SA	Service Agent
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SLP	Service Location Protocol
SOAP	Simple Object Access Protocol
SPDP	Service Peer Discovery Protocol
SSDP	Simple Service Discovery Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTL	Time To Live
UA	User Agent

UAC	User Agent Client
UAS	User Agent Server
UPnP	Universal Plug and Play
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WLAN	Wireless Local-Area Network
XML	Extensible Mark-up Language
XSD	XML Schema Definition

B Service and peer files used for the tests

File **services-diego.xml**:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<serviceList>
    <service>
        <serviceId>filemp3</serviceId>
        <name>The Spring.mp3</name>
        <source>
            <entityId
expire="3780">sip:diego@130.237.15.211</entityId>
            <entityAddress
entityType="IPv4">130.237.15.211</entityAddress>
        </source>
        <protocol>RTP</protocol>
        <expires>2004-04-06T12:03:37+01:00</expires>
    </service>
    <service>
        <serviceId>filemp3</serviceId>
        <name>The Autumn.mp3</name>
        <source>
            <entityId
expire="3000">sip:pojken@130.237.15.247</entityId>
            <entityAddress
entityType="IPv4">130.237.15.247</entityAddress>
        </source>
        <protocol>RTP</protocol>
        <expires>2004-04-07T11:33:48+01:00</expires>
    </service>
</serviceList>
```

File **peers-diego.xml**:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<peerList>
    <peer>
        <identity>
            <entityId
expire="3780">sip:diego@130.237.15.211</entityId>
            <entityAddress
entityType="IPv4">130.237.15.211</entityAddress>
        </identity>
        <network>WLAN</network>
        <cellId>3454FG</cellId>
        <expires>2004-03-26T12:03:37+01:00</expires>
        <services>
            <service>filemp3</service>
        </services>
    </peer>
    <peer>
        <identity>
            <entityId
expire="3780">sip:pojken@130.237.15.247</entityId>
```

```
            <entityAddress
entityType="IPv4">130.237.15.247</entityAddress>
        </identity>
        <network>WLAN</network>
        <cellId>fsdf</cellId>
        <expires>2004-03-30T09:00:37.673+02:00</expires>
        <services>
            <service>filemp3</service>
            <service>sipphone</service>
        </services>
    </peer>
</peerList>
```

File **services-pojken.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<serviceList xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="serviceList.xsd">
    <service>
        <serviceId>filemp3</serviceId>
        <name>The Autumn.mp3</name>
        <source>
            <entityId
expire="3000">sip:pojken@130.237.15.247</entityId>
            <entityAddress
entityType="IPv4">130.237.15.247</entityAddress>
        </source>
        <protocol>RTP</protocol>
        <expires>2004-04-02T17:58:17+01:00</expires>
    </service>
    <service>
        <serviceId>sipphone</serviceId>
        <name>Pojken</name>
        <source>
            <entityId
expire="3000">sip:pojken@130.237.15.247</entityId>
            <entityAddress
entityType="IPv4">130.237.15.247</entityAddress>
        </source>
        <protocol>RTP</protocol>
        <expires>2004-03-07T01:00:00+01:00</expires>
    </service>
    <service>
        <serviceId>printer</serviceId>
        <name>EasyPrint</name>
        <source>
            <entityId
expire="3000">sip:pojken@130.237.15.247</entityId>
            <entityAddress
entityType="IPv4">130.237.15.247</entityAddress>
        </source>
        <protocol>TCP</protocol>
        <expires>2004-03-02T12:03:37+01:00</expires>
    </service>
</serviceList>
```

File **peers-pojken.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<peerList xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="peerList.xsd">
    <peer>
        <identity>
            <entityId
expire="3780">sip:diego@130.237.15.211</entityId>
            <entityAddress
entityType="IPv4">130.237.15.211</entityAddress>
        </identity>
        <network>WLAN</network>
        <cellId>3454FG</cellId>
        <expires>2005-03-09T12:03:37+01:00</expires>
        <services>
            <service>filemp3</service>
        </services>
    </peer>
    <peer>
        <identity>
            <entityId
expire="3780">sip:pojken@130.237.15.247</entityId>
            <entityAddress
entityType="IPv4">130.237.15.247</entityAddress>
        </identity>
        <network>WLAN</network>
        <cellId>fsdf</cellId>
        <expires>2005-03-13T07:20:00+01:00</expires>
        <services>
            <service>filemp3</service>
            <service>sipphone</service>
            <service>printer</service>
        </services>
    </peer>
    <peer>
        <identity>
            <entityId
expire="3780">sip:gonzalo@130.237.15.248</entityId>
            <entityAddress
entityType="IPv4">130.237.15.248</entityAddress>
        </identity>
        <network>WLAN</network>
        <cellId>abcde</cellId>
        <expires>2005-03-11T07:21:01+01:00</expires>
        <services>
            <service>filemp3</service>
            <service>sipphone</service>
        </services>
    </peer>
</peerList>
```

File **services-gonzalo.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<serviceList xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="serviceList.xsd">
    <service>
        <serviceId>sipphone</serviceId>
        <name>Gonzalo</name>
        <source>
            <entityId
expire="3000">sip:gonzalo@130.237.15.248</entityId>
            <entityAddress
entityType="IPv4">130.237.15.248</entityAddress>
        </source>
        <protocol>RTP</protocol>
        <expires>2005-03-15T01:00:00+01:00</expires>
    </service>
    <service>
        <serviceId>web</serviceId>
        <name>My page</name>
        <source>
            <entityId
expire="3000">sip:gonzalo@130.237.15.248</entityId>
            <entityAddress
entityType="IPv4">130.237.15.248</entityAddress>
        </source>
        <protocol>HTTP</protocol>
        <expires>2005-03-17T12:55:37+01:00</expires>
    </service>
    <service>
        <serviceId>filemp3</serviceId>
        <name>The Summer.mp3</name>
        <source>
            <entityId
expire="3000">sip:gonzalo@130.237.15.248</entityId>
            <entityAddress
entityType="IPv4">130.237.15.248</entityAddress>
        </source>
        <protocol>RTP</protocol>
        <expires>2005-03-17T17:07:06+01:00</expires>
    </service>
</serviceList>
```

File **peers-gonzalo.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<peerList xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="peerList.xsd">
    <peer>
        <identity>
            <entityId
expire="3780">sip:gonzalo@130.237.15.248</entityId>
            <entityAddress
entityType="IPv4">130.237.15.248</entityAddress>
        </identity>
        <network>WLAN</network>
        <cellId>3454FG</cellId>
```

```
        <expires>2005-03-11T07:21:01+01:00</expires>
        <services>
            <service>filemp3</service>
            <service>sipphone</service>
            <service>web</service>
        </services>
    </peer>
</peerList>
```

File **empty.xml**:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<peerList>
    <peer>
        <identity>
            <entityId
expire="3780">sip:diego@130.237.15.211</entityId>
            <entityAddress
entityType="IPv4">130.237.15.211</entityAddress>
        </identity>
        <network>WLAN</network>
        <cellId>3454FG</cellId>
        <expires>2004-03-26T12:03:37+01:00</expires>
        <services>
            <service>filemp3</service>
        </services>
    </peer>
</peerList>
```

IMIT/LCN 2004-06

www.kth.se