Master Thesis

A modular SLP directory agent leveraging backend functionality for high performance and scalability

Author(s):
Breu, Lorenz

Publication Date:
2009

Permanent Link:
https://doi.org/10.3929/ethz-a-005906492

Rights / License:
In Copyright - Non-Commercial Use Permitted
Masters Thesis

A Modular SLP Directory Agent
Leveraging Backend Functionality
for High Performance and Scalability

by
Lorenz Breu

Due date
14. April 2009

Advisors:
Jan S. Rellermeyer
Prof. Gustavo Alonso

ETH Zurich, Systems Group
Department of Computer Science
8092 Zurich, Switzerland
Acknowledgments

The author would like to thank the following people:

- Jan Rellermeyer, for all his ideas and comments during the process of developing the project and writing this thesis.
- Nico Schottelius, for his technical support during the evaluation process on the IKR cluster.
- Emmanuel Lecharny, for his help with embedding ApacheDS and tracking down and fixing a really ugly bug in ApacheDS, as well as for the discussions on LDAP and RDBMS approaches.
- Samuel Pasquier, for his help with deploying jSLP DAs on the Cirrostratus framework and for the coffee.
- The guy in the MSc student lab whose coffee machine we used during the first half of the project.

And last but surely not least his family and friends who kept him going, provided him with coffee and offered the odd distraction to give his mind a break from jSLP.
Abstract

Today, the overwhelming majority of computing devices is permanently or temporarily connected to a computer network, and often providers offer services or resources on the network which clients then use. Discovering these resources and services is a key process for such networked systems and has led to the development of various approaches of how to solve this discovery. One such discovery system is the Service Location Protocol (SLP, RFC2608), and one implementation of said protocol is jSLP. This thesis describes the extension of the existing jSLP project to provide directory agent functionality. This directory agent component is designed to make use of existing replication functionalities of the systems used as backends to ultimately provide a scalable high-performance directory agent for SLP systems. Two different approaches, one based on the Apache Directory project and one on the Cirrostratus virtual OSGi framework, are described and evaluation results comparing these to an implementation in C (OpenSLP) are discussed. While no stable and productively deployable directory agent providing high scalability and performance based on a replicated backend was built, the foundation for future development has been laid.
## CONTENTS

4.3 Setup ............................................. 39
   4.3.1 Directory Agents .............................. 40
   4.3.2 Clients ...................................... 40
4.4 Profiling ........................................... 41
4.5 Cirrostratus-Specific Tests ......................... 41
   4.5.1 Cirrostratus Nodes and DAs .................... 41
   4.5.2 Registration Propagation ..................... 42
   4.5.3 Lookup Times with Varying Numbers of Directory Agents in the Network .......................... 42

5 Results .............................................. 45
   5.1 Individual Directory Agent Comparisons ............... 45
       5.1.1 Details for Job sr_predicate (Service Request) .... 45
       5.1.2 Details for Job ar_tags (Attribute Request) .......... 47
       5.1.3 Details for Job str_basic (Service Type Request) .... 48
       5.1.4 Profiling Results ............................ 58
   5.2 Cirrostratus Backend ............................... 62
       5.2.1 Registration Propagation ..................... 62
       5.2.2 Multiple Directory Agents .................... 63

6 Conclusions ........................................... 67
   6.1 ApacheDS as a Backend for jSLP .................... 68
   6.2 Running jSLP Directory Agents on Cirrostratus ........ 68
   6.3 Final Conclusions ................................ 69
   6.4 Contributions and Future Work ...................... 69
Chapter 1

Introduction

1.1 Motivation

Today, the overwhelming majority of computing devices is permanently or temporarily connected to a computer network [21]. While static configuration of networks may have been suitable when a device had to be set in place and connected to the network with a wire, this is no longer the case in today’s world of wireless and ad-hoc networking. If, for example, a researcher takes his laptop with him to a conference, it would be a hassle for him and the administrators of the conference if they had to manually reconfigure each such device to work in the local setting e.g., enabling these portable devices to use the local printer. Similarly, a less tech-savvy user might just want to take his new video streaming gaming console out of the box and have it access video files on the local file hosting device and play them on the TV, or an employee from the United States branch of an office may want to transfer his flashy new spreadsheet to his co-worker’s computer when visiting his office in Zurich. In all of these cases networking should work automatically without user intervention or manual configuration, as described by the Zeroconf Working Group in [14]. In addition, such a configuration scheme or protocol should also scale well, from two computers performing a quick file exchange to a home network with a dozen devices to an enterprise wide network. One of the main topics a zero configuration protocol has to address is thus service discovery [14].

Besides the discovery of hardware-based services or resources such as printers, DHCP, DNS or file sharing servers, service discovery can also play a role in the development and deployment of service oriented architectures (SOA). Developing systems as a loosely coupled set of modules instead of a highly coherent monolithic entity provides certain advantages. It can make it easier to change or update specific parts of the application without having to touch others, simply by localizing changes in a certain module while leaving the interface to the rest of the world as it was [27]. This may even allow for updates to be applied while the system is running by simply swapping out the component or service in question. An example of how such an approach is seen as beneficial even when used within a single machine can be seen in the development of the OSGi framework for the Java virtual machine [26]. The OSGi framework allows developers to deploy software modules (so-called bundles) which can make use
of other modules within the framework and offer their own services to be used by others. This use of modularization has been taken to the next level with R-OSGi [30] by allowing bundles to be used not only within the same OSGi framework, but also across a network connection. Of course, in order for such an architecture to work, it must be able to locate and bind to the services it needs. In a dynamic environment where changes or failures may be the rule rather than the exception this can be a challenging task.

This challenge exists in a very wide range of settings, and while the general problem may be the same at each scale, different scenarios impose unique restrictions and demands on the solution. Providing, finding and using services in a large enterprise may require a different approach than in a small office or even a home network. In addition, service discovery should also work in both relatively stable environments such as a wired network with fixed IP addresses as well as in highly dynamic environments such as ad-hoc wireless networks of mobile devices. Finally it would be an advantage if service location were not bound to specific platforms or programming languages, such that a client can discover services of any kind in a network without having to meet language or platform requirements before using the service discovery mechanism.

There are a few basic mechanisms that can be used to locate services in a network:

- Statically provide the address of the service in question to the client and directly use unicast communication between client and service.
- Use a central well-known repository where service providers register the services they offer and clients look for services they need, using unicast between all involved parties.
- Use a completely decentralized system, where all service announcements and service lookups are broadcast or multicast. Once a client discovers a suitable service it can then use unicast to communicate with the service provider.

The first approach, while providing an administrator with a high level of control, immediately leads to problems when the address of a service changes. If that happens, all clients using that service have to be updated. Such an approach is hardly feasible in a dynamic setting. The second approach avoids the problem mentioned before in a sense that now only the central repository has to be located (or known), all other services are available from there. The explicit address of the central repository or directory may still have to be known in advance by clients, however all the actual services available in the system can move around at will and just have to reregister with the directory. Alternatively one of the well established protocols such as DHCP [8] can be used to automatically configure clients with the location or address of the directory or find it based on a name and not an explicit address. Using the third approach clients need no initial knowledge of where to look for services, they simply ask for or advertise them to all other nodes in a network using broadcast or multicast. This however can lead to increased load on the network. If services move around much, or clients frequently ask for new services, or there is a high churn of network nodes (e.g. in wireless ad-hoc network), this approach may not scale well to networks with many nodes. However, due to the fact that such systems should work
1.1 Motivation
dynamically with as little static configuration as possible and adjust to changes
in their environment, this approach is often encountered [7, 6, 22, 14]. One
important benefit of using multicast messaging is that when two participants
exchange information, all others have access to this information as well, thus
e.g., detecting that a specific node has disappeared from the network [7].
An example of a protocol leveraging multicast discovery is the Service Loca-
tion Protocol (SLP). SLP was first proposed in 1997 [34] and updated to and
replaced by SLP Version 2 in 1999 [33]. SLP is a protocol used to discover ser-
vices in IP networks [13]. Designed to be scalable, SLP can make use of a central
repository (called a Directory Agent or DA), yet it can also work without one
using multicast service discovery. User agents (UA) perform service discovery
for clients, while service agents (SA) advertise services for their providers. SLP
can also be extended to meet the requirements of various networks or systems,
as the protocol offers so-called extensions. These can be used to pass additional
information not included in the messages of the base protocol to the client [15]
or even add new functionality such as subscription and notification [17]. SLP
defines a series of messages as well as how these messages should be handled by
implementations of the protocol, hence it is not bound to any specific program-
ming language or architecture. Various implementations of SLP are available
to be used as libraries in bigger applications, such as OpenSLP programmed in
C [23, 24] and jSLP programmed in Java [29].
Besides being able to advertise itself and be discovered, a service in a networked
environment should itself also be able to handle the load produced by clients
using it. In this respect it plays a big difference if a service is running in a SOHO
environment with a limited and rather static number of clients, or if it has to
provide its services to a large and possibly dynamic number of clients in an
enterprise size network. One way of solving this is to add multiple instances of
the service provider. This is also the case for SLP DAs, which are themselves also
services and can be discovered using SLP [33]. However, simply adding a new
instance of a stateful service can lead to inconsistencies between the information
or service received by clients, depending on which instance they contact. One
way to keep these instances consistent is to have the registering entities ensure
consistency by contacting all known registration points. Alternatively, instances
of services can be deployed not as independent instances, but rather as replicas,
representing the same logical instance or simply keeping their state consistent
(e.g. replication of the Apache Directory Server [1]).
The goal of this project was to extend the existing jSLP 1.0 implementation
by adding a modular high performance DA. Performance was to be achieved
by having DAs replicate their cache of stored services without having to extend
or alter the base SLP protocol. This was to be achieved by leveraging the
replication features of the actual storage backend. Two possible approaches
were selected for this purpose: the Apache Directory Server LDAP server [1]
and a backend deployed as an OSGi [26] service and replicated through the new
Cirrostratus virtual OSGi framework (Work in Progress, undocumented). In
order to achieve this goal various subtasks had to be completed:
• Porting of jSLP to the MINA communication framework [2]
• Adding basic multicast functionality to the MINA framework
• Developing the modular DA as part of the jSLP project
• Devising a mapping between SLP services and LDAP entries without using SLP templates for the Apache Directory Server backend

• Deployment of the new DA as a bundle on the Cirrostratus virtual OSGi framework

• Comparison of the jSLP base implementation with the existing OpenSLP DA implementation [23, 24]

• Comparing the various jSLP approaches with a basic no-frills implementation of the jSLP DA as a baseline value

1.2 Related Work

As the importance and prevalence of networked systems has grown, there have been a number of attempts, approaches and implementations of systems to locate services, resources or components within a network ([13, 22]). Several such systems deal with the automatic discovery of services in some way or another and are thus related to SLP and well worth mentioning. All of the following systems provide means for service discovery in a network. The first three (DHCP, DNS and Jini) use a centralized approach i.e., they rely on the existence of a central server for service registration and subsequent lookup. UPnP on the other hand is fully decentralized. SLP uses a hybrid approach, making use of one or more DAs to reduce network traffic when present, but also being equally able to function in the absence of DAs using multicast techniques. The following is not an exhaustive list of available technologies for service discovery, but lists those most often compared to SLP in the literature (e.g. [22, 13]).

1.2.1 DHCP

The dynamic host configuration protocol DHCP, as described in [8], was proposed in 1997 as a means to automatically configure network hosts using DHCP servers that determine how these should be configured. The DHCP servers themselves are found by clients using broadcast. The DHCP protocol allows a server to include "options" to be sent to a client, which can include service locations. Among these options are those with code 78 and 79, which can pass DA addresses or scope information, respectively, to SLP clients [28]. However, this address and scope data must be managed outside of the scope of DHCP. Besides, using DHCP alone a DHCP server cannot determine if a service located at an address provided to a client using the options field is alive [13, 8]. The only time a DHCP server checks if an IP address is in use and responding is when a DHCP client requests a (new) IP address. DHCP also does not allow a client interested in a particular kind of service to specify this, whereas SLP allows a client to pass along a search filter with a request, limiting replies to services that meet the exact needs of the service.

1.2.2 DNS

The domain name system DNS, used mainly to resolve host names to IP addresses, has also been proposed as a source of service location information [12].
1.2 Related Work

Similar to the use of the existing “option” field with DHCP above, the DNS system is extended to provide service location information in the form of special SRV resource records, which have been assigned the resource record (RR) type value 33 by IANA. Due to the limited amount of information provided by such an RR however, a client looking for specific services matching certain needs in a network with possibly a large number of similar services must retrieve all resource records for a specific service type and then contact each in turn to determine service properties. The result of using a method for dynamic configuration of link-local IP addresses [6] and multicast DNS, which provides domain name resolution on the local link [7], would lead to a similar behaviour as with SLP running without a DA. Hosts on the local link could ask for DNS SRV resource records using multicast and get appropriate answers if such services are present, just like a SLP UA would multicast service request messages and SAs would answer. However the multicast DNS approach would be restricted to the local link, and as with a dedicated DNS server, service records do not include any information on the actual service. Yet just like with DHCP, SLP itself can be extended to make use of these SRV RR such that clients looking for services using SLP can now find services outside of their local domain [40]. Using SLP, which uses multicast to discover local services if no DA is present, a user agent cannot find services in remote domains to which there is no multicast (or broadcast) route. In such a scenario clients can use DNS SRV to discover SLP directory agents in remote domains and then use plain SLP to communicate with those DAs to get information about services available in that remote domain.

1.2.3 Jini

Jini is a Java-based system that provides a set of conventions, programming mechanisms and APIs that allow the registration and discovery of services in a network, making it possible to build distributed Java applications in the form of “federations” of components [37]. While SLP is an IETF standard protocol which is not bound to any specific programming language, JINI makes extensive use of functionalities provided by the Java programming language and generally expects components to run on JVMs. However it is possible to provide proxies that bridge non-Java components into a JINI federation, thus also making it possible to bridge the gap between SLP and JINI systems [13]. JINI uses a lookup service to register and discover services, similar to the directory agent in SLP. Yet unlike SLP, a lookup service must be present for JINI to work, while SLP will also work without a DA. While both SLP and JINI allow clients to look for services meeting certain requirements, SLP uses string-based filters whereas JINI makes use of the Java type system and matches services to requests based on the requested interface. One of the main advantages of JINI is that JINI provides proxy objects when a service request is answered. This places the implementation of the communication between the client and the service provider solely on the side of the provider. The client interacts with the local proxy object and all that goes on between that proxy and the actual service is hidden from the client, making service invocation transparent. Therefore, communication details or even the complete protocol can be changed by the service provider without the client needing any kind of update, it just looks up the service as before and interacts with the proxy it receives. It also means that
no drivers have to be installed on clients, as all the code they need to use the service is provided by the lookup service in the form of said proxy. However, this mechanism is already outside of the scope of pure service location.

1.2.4 UPnP

The UPnP framework was designed by a forum of vendors of networked appliances as a flexible, scalable, platform-independent way of communication between network-enabled devices [32]. UPnP allows clients (called control points in the UPnP terminology) to discover devices and devices to advertise themselves to control points using the Simple Service Discovery Protocol (SSDP) based on the multicast exchange of HTTP messages [32]. UPnP is a fully decentralized peer-to-peer architecture. In order to make the framework truly independent of any kind of drivers it relies on the exchange of SOAP messages and makes extensive use of XML documents and strings for the description of a device’s or service’s properties. This allows the UPnP framework to provide more than simple service discovery by extending this with with specifications about description (what can a service do), control (how can it be accessed) and presentation (allowing a device to present a graphical interface to the user). UPnP allows control points to discover devices and services of a specific type and version, however unlike SLP it cannot provide detailed search filters on a device’s or service’s attributes or properties. After discovery UPnP-enabled devices can of course retrieve a service’s description in XML format, and while this use of standard notation supports interoperability between devices of different vendors, it adds additional overhead in the form of XML parsing and transmission (e.g. [30]).

1.2.5 OpenSLP

OpenSLP is, as the name suggests, an open source implementation of the SLP protocol [33, 18] programmed in C currently available in version 2.0 [23, 24]. This implementation provides SLP functionality both in the form of a daemon process (slpd) for the SA and DA parts of SLP as well as a library providing SLP methods for UA development. OpenSLP implements all the required features of the SLP protocol as well as secure operation as suggested in RFC2608 [33], however early versions switched the sequence of certain fields before signing. Other interoperability problems arose during the implementation and testing phases of jSLP 2.0, hence all security related issues are considered outside the scope of this project and remain to be fully implemented and tested. OpenSLP was considered a reference implementation during the development of jSLP 2.0 as the implementation in C was considered to be adequately efficient. In addition it provided a counterpart for interoperability testing of the basic SLP functions. However, the jSLP implementation aims to provide more than OpenSLP, as UA, SA, and DA functionalities can be used independently or combined. The jSLP DA also can plug into a range of implementations of the actual service storage, making it much more flexible than OpenSLP with its slpd daemon. Besides adding functionality beyond the scope of SLP through this storage backend, direct access to the cached services is also possible in jSLP.
1.3 Outline of the Thesis

After the brief introduction to service discovery and the role of SLP, chapter 2 provides a more in-depth description of SLP functionalities as well as the additional technologies used to develop the jSLP directory agent. Chapter 3 then describes the details of the actual design, followed by a description of methods used in the evaluation of the implementation comparing it to OpenSLP in chapter 4. Results of this evaluation are shown in chapter 5. Chapter 6 concludes this thesis and provides a short outlook at and suggestions for future work in this area.
Chapter 2

Background

This section provides more in-depth background information on the topics most relevant to this thesis. It describes the SLP protocol in detail, and provides an overview over the components on which the jSLP version 2.0 implementation (especially the DA part) of the SLP specification was based: MINA [2], the Apache Directory Server [1] and OSGi/Cirrostratus . It concludes with the closer look at the actual problem this thesis tried to tackle and how this was approached.

2.1 Service Location Protocol SLP

The Service Location Protocol was first presented in 1997 [34] and later fixed and extended to the current version, SLPv2 [33]. It was designed to be scalable both in terms of network size (use of directory agents) and administrative complexity (scopes and other restrictions) and provides support for extensions that can be used to adapt the protocol to specific environments or requirements. Besides basic service discovery functionality, the protocol also provides mechanisms to browse available services and their attributes.

2.1.1 Services and Service Types

Services in the SLP specification are defined by a service type, a URL and a set of attributes defining that service [33]. Service types in SLPv2 can be either abstract or concrete, whereas the earlier SLPv1 only supports concrete types. The service’s URL specifies the location where a specific service can be reached and begins with ”service:” followed by the service type and the site URL path as defined in [16]. As an example a line printer service may offer a service of abstract type ”printer” and concrete type ”printer: lpr” at printer.example.com on port 1234 and would thus advertise itself with a URL of ”service:printer:lpr://printer.example.com:1234”. A file transfer service may use the URL ”service:ftp://ftp.example.com:21”. A service may also have a series of specific attributes describing the service in detail. An attribute in SLP is a string of the form ”(attributeName = attributeValue)” where attributeValue can contain multiple values separated by commas. Describing the exact syntax used in SLP is outside the scope of this thesis and can be found in [33]. SLP supports
2.1.2 Service Discovery

Services in SLP are advertised by so-called service agents (SA), while user agents look up services on behalf of client applications. In a basic setting UAs simply broadcast "Service Request" messages to multicast IP address 239.255.255.253 over UDP on a specific port. The default port for SLP is 427, however in some operating systems, such as most Linux systems, binding listener sockets to ports below 1024 is restricted to the root user. The proposed API for SLP [18] does not explicitly provide a property to specify an alternative port, yet jSLP introduces the property "net.slp.port" already in version 1.0 [29], which remains unchanged in version 2.0. The OpenSLP implementation (in C) does not yet have this property in the release code [23, 24], yet a patch introducing this property was proposed (independently of the jSLP project) by R. van de Kraats (see [24]) and was applied to the version used for comparison during the course of this project. A "Service Request" message can include an LDAPv3 filter string to specify exactly what kind of service the UA is looking for [36, 35, 33]. A UA looking for a printer that is located in the main building and supports color printing could include the following search filter

"(&(location=main)(color=true))"

in a request for a service of type "service:printer". The request is multicast, so all service agents listening to such multicast messages receive the request and compare it to the services they offer. If there are one or more matches, these are advertised to the user agent that sent the multicast request in a unicast "Service Reply" message. This basic exchange of multicast requests and unicast replies allows for spontaneous operation of SLP user agents and service agents without prior manual configuration. If a service wants to advertise itself to clients using the SLP protocol it simply has to start a service agent that listens for requests. Similarly a client that wants to use SLP-enabled services can easily discover them by multicasting requests which will be heard by all service agents that are present as shown in Figure 2.1. As messages are sent over the inherently unreliable UDP protocol user agents use the so-called "multicast convergence algorithm" [33]. This forces them to resend request messages several times with exponentially increasing time periods between each send event. The algorithm terminates when no new replies arrive. In order to prevent the same service agent from unnecessarily repeating to the same request multiple times, the user agent adds a list of previous responders to each request, and a service agent only replies if its address is not in that list.

While the method mentioned above works fine in smaller networks, using multicast messaging increases the number of messages sent on the network. In addition, a user agent has to potentially wait for several timeout events before continuing when running the multicast convergence algorithm. IP multicasting also requires infrastructure support as all routers in the network must support multicasting [20]. IP multicast also requires the routers to build and maintain a multicast tree, which also involves additional traffic e.g., in the form of IGMP
2.1 Service Location Protocol SLP

Figure 2.1: Service discovery using multicast between User Agents and Service Agents

messages [20]. To prevent these effects and to make the SLP protocol scale to larger networks, a directory agent (DA) may be deployed in the network in the same multicast group [33]. This DA is basically a cache of known services in the SLP framework, as service agents now register their services with all known DAs using unicast "Service Registration" messages, and user agents look services up directly from a DA using unicast "Service Request" messages sent to the DA instead of multicast messages sent to the group, as shown in Figure 2.2).

Figure 2.2: Service registration and discovery in the presence of a DA

Multiple DAs can be deployed at any given time, with all SAs registering services with all DAs they know and UAs send requests to one of them. UAs and SAs find DAs in two ways: active DA discovery or passive DA advertising. In the former, agents send multicast "Service Request" messages for services of type "service:directory-agent" to which all available directory agents send "Directory Agent Advertisement" (DAAdvertisement) messages. In the latter method DAs periodically send unsolicited DAAdvertisement messages to the multicast group (see Figure 2.3).

When a UA wants to discover a service and it knows of a DA in the network, it will try to unicast requests to it as mentioned above, if this fails, it will fall back to the simple multicast alternative and commence the multicast convergence algorithm to which all SAs (and DAs) will respond with unicast Service Replies. Thus DAs reduce network load when present by removing the need for multicast lookups, yet in their absence or if they fail, the SLP system will continue working correctly.

Besides the aforementioned messages, SLP also provides a series of messages used to browse the services available in the network.
Table 2.1: SLP Message Types

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Required</th>
<th>jSLP</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Request</td>
<td>X</td>
<td>X</td>
<td>Used to discover services of a given type matching certain criteria.</td>
</tr>
<tr>
<td>Service Reply</td>
<td>X</td>
<td>X</td>
<td>The reply to a Service Request, contains URLs of all services matching the request.</td>
</tr>
<tr>
<td>Attribute Request</td>
<td></td>
<td></td>
<td>Can be used to either get all attributes of a specific service, or to get a list of all attributes and their values present in all registered services.</td>
</tr>
<tr>
<td>Attribute Reply</td>
<td></td>
<td></td>
<td>The reply to an Attribute Request, contains the matching attributes as strings in the form &quot;(name=value)&quot;.</td>
</tr>
<tr>
<td>Service Type Request</td>
<td></td>
<td></td>
<td>Used to discover the types of all registered services.</td>
</tr>
<tr>
<td>Service Type Reply</td>
<td></td>
<td></td>
<td>The reply to a Service Type Request, contains a list of strings of service types in the form &quot;service:abstract_type:concrete_type&quot;.</td>
</tr>
<tr>
<td>Service Registration</td>
<td>X</td>
<td>X</td>
<td>Used to register a Service with a Directory Agent. (^a)</td>
</tr>
<tr>
<td>Service Deregistration</td>
<td>X</td>
<td></td>
<td>Used to deregister a service from a DA. (^b)</td>
</tr>
<tr>
<td>Service Acknowledgement</td>
<td>X</td>
<td>X</td>
<td>Used by the DA to notify an SA of a successful registration or deregistration.</td>
</tr>
<tr>
<td>Directory Agent Advertisement</td>
<td>X</td>
<td>X</td>
<td>Sent by DAs as a reply to a Service Request for a &quot;service:directory-agent&quot; service. Optionally also sent out periodically by DAs to inform other agents of their presence.</td>
</tr>
<tr>
<td>Service Agent Advertisement</td>
<td></td>
<td></td>
<td>Sent by SAs as a reply to a &quot;Service Request&quot; for a &quot;service:service-agent&quot; service. (^c)</td>
</tr>
</tbody>
</table>

\(^a\)Service Registration messages include a FRESH flag, which when set specifies this registration to be incremental i.e., either adding new attributes or changing existing ones. This optional feature is not supported by OpenSLP, but it is by jSLP 2.0.

\(^b\)Service Deregistration messages include a FRESH flag, which when set only removes the given attribute(s) from the registration. This optional feature is not supported by OpenSLP, but it is by jSLP 2.0.

\(^c\)This feature is only used when a UA is not configured with scopes, allowing a UA to discover all scopes in its SLP neighbourhood. Not supported by jSLP at the moment.
The SLP specification describes how the basic SLP protocol can be extended. This is done by defining so-called extensions which can be added to all message types according to [33]). These extensions have a number, either determined by an authority such as IANA or by the deployer/implementor of the SLP protocol, and they can be either mandatory (error messages are sent back if an extension is received but not understood) or optional (unknown extensions are ignored). Various extensions have been suggested to allow the SLP protocol to fulfill a variety of additional functions. This ranges from the aggregation of information that usually requires several messages to be exchanged into one single message (Attribute List Extension, [15]), over processing of results on the SA/DA (Select and Sort Extensions, [39]) to the implementation of a subscription and notification framework over SLP [17]. These extensions are not part of the required minimal SLP functionality and therefore have been part of neither OpenSLP nor jSLP 1.0. jSLP 2.0 supports the attribute list extension specified in [15], and further extensions may be added at a later date using the attribute list extension as a template.

SLP allows administrators to restrict which services can be discovered by specific clients using scopes. By default all agents participating in the SLP protocol are in the (case-insensitive) scope "DEFAULT" [33]. However it may make sense to deploy agents in specific scopes, e.g. mirroring administrative areas or location. An administrator may specify certain agents to be in the scope "lab" and others in the scope "main" for example, in which case a service agent in the "lab" scope would only answer to requests from user agents in the same scope. An agent can also support multiple scopes. Figure 2.4 shows an example of how a UA with scopes X and Y uses unicast to communicate directly with the DA it knows in scope X. The UA has not found any DA in scope Y and hence falls back to the multicast convergence algorithm to discover services in scope Y.

SLP also provides security mechanisms to sign specific SLP messages using public-key cryptography, however they are either rarely used or incorrectly implemented as in the case of OpenSLP, which incorrectly signs certain messages in versions 1.x [23] when compared to the SLP RFC [33]. Therefore, all security questions are not considered in this thesis, and the jSLP 2.0 implementation, while providing the methods and hooks necessary to run SLP with security enabled, has not been tested to work in such a setting.
2.1.3 SLP and LDAP

The Lightweight Directory Access Protocol (LDAP) provides clients access to information stored in directories [36]. When SLP was initially developed, the idea was that service types would be defined in the form of "templates" [16]. Such templates describe a service type in detail, specifying what attributes a service of a given type must declare and which ones are optional, as well as which of the five types (String, Integer, Boolean, Opaque, keyword) supported by SLP these attributes should be of. Templates would also be registered with the Internet Assigned Numbers Authority (IANA), such that these types could be universally used in the same manner. This is similar to the way entries in LDAP servers implement certain classes of objects, which are in turn defined using so-called schemas [36]. Schemas describe which attributes an object of a certain class may have or must have, how many values a certain attribute may contain as well as which types these attributes are, including the syntax to use when comparing two attributes. As the two concepts of service type templates and LDAP schemas are so similar, ways to map a given SLP service type template into an LDAP schema and vice-versa have been elaborated [19]. LDAP also plays a role in the basic SLP request functionality, as SLP requests messages e.g., service requests, can contain a filter specifying the exact kind of service a client (UA) is looking for [33]. These filters use the same syntax as used in LDAP filters used to retrieve entries matching attribute criteria from a directory [35].

These relationships between LDAP and SLP were also reasons why an LDAP backend was initially chosen to hold the service registrations by the new jSLP 2.0 directory agent (see chapter 3). However, while the use of LDAP filter strings to refine SLP lookups poses no limitations, the static definition of service type templates are a major obstacle when deploying an SLP DA in an unmanaged network, in a dynamic network with many SLP SAs joining and leaving the network, or in a network where new services are designed and deployed. In such a scenario the DA would need to know in advance what kind of registrations to expect. This means that a network administrator has to either provide new templates to the DA directly each and every time a service of a new type is deployed in a network or the attribute set of a specific type change. If, for example, an administrator has a series of custom services of type mycompany.printer deployed and now has just bought a shiny new printer that offers the new "add-moustache" feature when printing photographs. He may
want to advertise this by adding the keyword "moustache-enabled" to the new service providing access to that printer. In this case he would have to either change and redeploy the templates of his mycompany.printer service type, or introduce a new service type mycompany.moustacheprinter. Now imagine the same problem in a dynamic enterprise network with dozens of service types, one DA per floor and a team of administrators. Templates also pose a problem on the other side of the scale. While SLP will probably be used without a DA in SOHO networks, an administrator of such a network may also want to deploy a DA. Now a new device with an embedded SLP SA is hooked up to the network. If a DA receives registrations of the new service for which it doesn’t have a template, things may get a bit messy. Of course the problems with templates do not only apply to DAs, but also to UAs expecting service types to have certain attributes, forcing SA deployers to stick to such static templates.

While in certain cases an administrator may find it practical to have full control over the kinds of services available in his SLP system, the use of templates is a hindrance in dynamic and unmanaged networks. Therefore, it is not surprising that OpenSLP does not enforce templates and just caches any registration without checking if a service of a specific type matches any known template [23, 24]. The directory agent part of jSLP 2.0 does also not enforce any templates in the current state, however this functionality may be added as an optional feature to future backends. The jSLP does check that attribute strings in registration messages are valid according to the SLP specification [33], however all service registrations are then simply stored in the cache backend and can be retrieved by UAs without the UAs having to know what kind of attributes to expect. Once a service of a specific service type is registered however, the DA and SA add an entry for that attribute in the attribute type registry, enforcing the type for that attribute name for any subsequent registrations of services of that same abstract type. The browsing feature of SLP using "Service Type Requests" and "Attribute Requests" allow a UA to determine which kinds of attributes and service types are stored in the DA, removing the need to know such information explicitly before using the SLP system.

This dynamic "cache-anything" behaviour, when mapped to a directory backend, introduces a new problem, as LDAP directories store and retrieve entries that are of a certain object class with a set of attributes defined in the respective schemas. This introduces a new form of "impedance mismatch" between the dynamic SLP approach and the static LDAP entry definitions. Luckily version 3 of the LDAP protocol introduces the new "extensibleObject" object class [35]. An entry of the type extensibleObject can have any attribute known to the LDAP system. This removes the "impedance mismatch", allowing the creation of service registration entries in an LDAP-based SLP DA: SLP accepts any attributes that are correct according to the SLP specification and stores the service registration as an entry of the extensibleObject class, allowing it to have any attribute (see chapter 3 for implementation details).

2.2 Relevant Projects and Frameworks

This section describes the products, projects and frameworks used during the course of this project and in the development of jSLP 2.0.
Apache MINA is an open-source network communication framework that provides an API for asynchronous event-driven communication [2]. The framework provides Connectors ("client-side") and Acceptors ("server-side") as communication endpoints. When a message is sent or received by one of the two endpoints, that message passes through a chain of filters that allow processing of messages before they are handed over to the application using the framework. One of the great advantages of this framework is the possibility of adding protocol codec filters into this chain, which decode the data contained in buffers (as received by the Acceptor for example) into Java objects and encode Java objects into byte buffers for sending, such that the application above the framework no longer has to deal with byte level manipulations or parsing but can directly access the corresponding message objects. Protocol level connection details are taken care of by the framework and abstracted away in the form of IoSession objects, which provide asynchronous operations such as writing or reading data or closing a session to a handler (in the form of an IoHandler object). The handler finally implements the actual business logic of the application. The current release version of the MINA framework at the time of writing was 2.0.0-RC1 [2]. This version provides all the functionality mentioned above for unicast TCP and UDP communication using the Java NIO library. Work had also been started on an implementation using the Tomcat Native Java wrapper of the Apache Portable Runtime (APR) library written in C [3, 5, 2]. The APR library provides a consistent interface to the system-specific implementation of network sockets. Unfortunately, the NIO library does not include multicast support, which is required for SLP (and thus jSLP). However, as jSLP 2.0 was initially to become part of the Apache Software Foundation (ASF) [4] using the Apache Directory Server [1] (see Section 2.2.2), which in turn bases communication on the MINA framework, the goal was to also base jSLP itself on MINA. Therefore, the APR-based implementation had to be extended by implementing an Acceptor (AprDatagramAcceptor) and Connector (AprDatagramConnector) for UDP transport. jSLP 2.0 now uses these two implementations for all communication, both unicast and multicast. The code has been released for comment and inspection to the MINA community. Unfortunately, the APR-based implementation now requires platform-specific libraries to work, which makes jSLP 2.0 less easy to install and use and may make it impossible to embed on some systems at the moment. One of the major upsides of jSLP 1.0 was its small footprint and the fact that it was based purely on Java, making it suitable to embed on devices with limited resources, as long as they ran a JVM. It also made things easier when deploying jSLP 1.0 as an OSGi bundle, as dependencies were limited to packages provided by the OSGi framework, indeed even when used as a standalone component jSLP 1.0 only depended on a logging library. However, jSLP 1.0 can still be used when only UA or SA functionalities are required, as the SLP protocol allows such a device to interact with a jSLP 2.0 agent without any problems. A DA will most likely not be deployed on devices with limited resources, hence the development of jSLP 2.0 with additional requirements still makes sense.
2.2 Relevant Projects and Frameworks

2.2.2 Apache Directory Server

The previous sections defined the basis of this project: The SLP protocol (with jSLP as an implementation of this protocol) and the MINA framework over which the SLP messages are sent. This section provides a brief description of the Apache Directory Server (ADS), as the initial approach of this project was to use the ADS as the backend for the jSLP DA part.

Apache Directory Server is an LDAP server written entirely in Java and has been certified as LDAP v3 compliant by the Open Group [1]. ADS aims to provide a complete high-end LDAP server solution with functionalities similar to modern database systems such as stored procedures, triggers and replication. The LDAP entries are stored in partitions, which by default are based on JDBMS implementations of B+ trees, yet custom implementations are possible. Similar to the way messages pass through a chain of filters in MINA, requests for entries made to the ADS pass through a series of interceptors. These interceptors can handle such things as checking authorization, making referrals to other servers, normalizing requests or performing any custom actions on requests [1]. Once a request passes through the interceptor chain it is processed by the partition nexus, which does the lookup and returns the result(s). These again pass back through the chain of interceptors and are then sent back to the requester.

Instead of operating ADS as a standalone LDAP server however, this project made use of the possibility of embedding the ADS directly into the jSLP DA, such that the jSLP DA performed the SLP actions and stored services as LDAP entries in the embedded instance of the ADS. "Opening" the embedded ADS to access by LDAP clients is possible by starting the appropriate communication features. The idea was to be able to create multiple DA instances on various machines and have their ADS backends replicate their data among themselves, providing various access points (the jSLP DAs running on the various hosts) to the same cache of service registrations (the replicated ADS backends) in order to decrease individual lookup times for UAs.

2.2.3 OSGi and Cirrostratus

The OSGi technology is a specification for a module System in Java, allowing the development of component-based and service-oriented applications [26]. The specification is developed and maintained by the non-profit OSGi Alliance. There are various implementations of the OSGi specification, the one chosen for this project was Equinox, the OSGi framework upon which the Eclipse IDE is built [9]. As Eclipse was used for Java programming tasks, the necessary JAR files where thus already present. OSGi provides a foundation on which individual components (termed "bundles") can be deployed. Each such bundle contains a manifest file, in which all its dependencies are listed as well as all the packages that bundle exports for others to use. The framework takes care of matching these exports and dependencies and will not let a bundle be started before all dependencies have been resolved. The framework also allows components to be easily updated individually. In addition, bundles can register services with the framework which others can then use through the OSGi framework. The basic OSGi implementations are designed to run on a single host, but the use of these services across host borders was made possible by R-OSGi [30]. R-OSGi distributes the local service repository and uses SLP to locate services among
Background

all participating hosts. Remote calls to services are handled by proxies provided by the R-OSGi bundle running on the local OSGi implementation. Thus the benefits of modular, component-based and service-oriented software design provided by OSGi can be spread throughout the network.

Cirrostratus is the new "working title" of a project formerly referred to as "Virtual OSGi", which aims to virtualize a single OSGi framework running on a set of nodes. These nodes form a distributed hash table (DHT) and services can be replicated among nodes. The implementation of this project is apparently still in an early and messy stage, hence there are no literature, documentation or references available at the time of writing.

The jSLP DA implementation was to make use of the replication features of Cirrostratus in a similar fashion to the approach using the ADS as a backend mentioned in section 2.2.2. For this purpose a new backend was developed as an OSGi bundle that registers a service on Cirrostratus, which is then replicated among the participating nodes and can be consumed by DA bundles running on the individual nodes. Thus registering an SLP service on any of the DAs using that specific Cirrostratus service leads to the registration with all other DAs using the same backend Cirrostratus service. The system should then be scalable as increasing load on nodes running DA bundles could be reduced by adding a new DA on a Cirrostratus node consuming the same backend service used by the existing DAs. This means that all DAs are kept consistent even if SAs only register with a single DA or with a subset of all DAs present in the network.

2.3 High-Performance Replicated Directory Agent for SLP

The previous sections described the individual components of this project and jSLP 2.0, but how do they all fit together? This section aims to clarify that question.

jSLP in the previously released 1.0 version supported the basic SLP protocol, yet only the user agent and service agent parts [29]. This and the fact that jSLP was a pure Java implementation with few dependencies and a small footprint made it a good candidate for use on devices with limited resources. However, unlike OpenSLP, jSLP did not provide any DA functionality. The goal of this project was thus to extend jSLP to also provide DA functionality. In addition, the DA should provide high performance, being able to handle a high rate of incoming requests correctly and quickly. To make things even more scalable, it should be possible to add further DAs to the network, which should then decrease the load on the existing DAs and allow UAs to find services faster.

The existing implementations of SLP directory agents, OpenSLP in specific, provide DA functionality, yet they require a daemon process to be started to handle SLP messages. The jSLP implementation provides a modular approach to SLP, allowing all SLP agents to be used in any combination. jSLP can also be built as an OSGi bundle, registering the three agents as services with the OSGi framework on which it is started. The goal, as previously mentioned, was to add a DA to the existing jSLP project. The new DA was to be integrated as smoothly as possible with the existing SA and UA components. The DA should
also provide further features making it a suitable replacement for existing DA implementations such as OpenSLP. That also meant designing a DA that could match the performance of OpenSLP or even surpass it. The ultimate goal was to provide a DA that, even if it may be outperformed by an OpenSLP DA when running independently, is inherently scalable to multi-DA SLP deployments.

The main approach was to develop independent DAs that can tap into a single logical service cache without adjusting or extending the SLP protocol. DAs are not designed to communicate with each other in the SLP specification [33]. An extension to the SLP protocol has been proposed to allow DAs to keep each other’s caches consistent [38]. The approach of this project differs in that it makes use of existing protocols and replication features of data storage backends, such as LDAP servers, or application frameworks, such as Cirrostratus.

By separating the protocol component of the DA from the actual storage backend, the DA can be adapted to work in a range of scenarios. If a simple out-of-the-box solution with minimal size is required, the backend can be a simple HashMap. If a proper embedded database engine with transactional capabilities is required, the backend could be based on BerkeleyDB for Java [25]. If an administrator wants to integrate service location with a full-fledged SQL database providing remote access, the HashMap backend can be replaced by a backend that interacts with said database. This project made use of this modularity by developing two backends besides the simple HashMap solution: one based on an embedded Apache Directory Server and one based on a backend deployed as a replicated service on a Cirrostratus framework. In both cases the idea was to be able to deploy DAs on different hosts and have their backends replicate themselves so that the DAs keep a consistent service cache among each other.

In the former this was to be achieved by using the replication feature of the embedded ApacheDS backends, while in the latter the Cirrostratus framework was to replicate the storage service to all nodes on which DAs were started. The default behavior for SLP SAs is to register services with all DAs known, therefore one may be inclined to think that adding replication to a DAs cache of services is not required. Placing the burden of keeping DAs consistent on the SAs however requires them to keep track of all DAs on the network. The SAs must register every service with every DA. When a DA is rediscovered by an SA e.g., after a reboot of the DA or SA or after a partitioning of the network is fixed, the SA must again register all services with that DA to maintain consistency. If there are many SAs present, then this can lead to problems with scalability, as each new SA has to communicate with each DA, such that adding either an SA or a DA leads to a lot of additional communication on the network.

As it can be assumed that there are fewer DAs than SAs in the system, moving the task of keeping DA service caches consistent from the SAs to the DAs should reduce network load. The benefits of having the DAs keep each other consistent instead of replying on the SAs has however already been discussed in [38], where the problem is solved by extending SLP to mSLP by adding new extensions, attributes and one message type to the SLP protocol. This project uses a different approach and makes use of existing replication features of the backend used instead of modifying SLP. Forcing SAs to to register with only a subset of available DAs can be done using existing SLP configuration options and does not require changes to the SLP logic.
This chapter details the design of jSLP 2.0 in general and the jSLP directory agent in particular.

The basic design of jSLP 2.0 was adapted from jSLP 1.0. While adding the directory agent to jSLP 2.0, care was taken to remain as close to the jSLP 1.0 implementation and to insert the new features in a way that mirrored the existing modularity of the UA and SA parts of jSLP 1.0. However, some changes and API adjustments were inevitable in order to provide the new functionality. In addition, during the transition from 1.0 to 2.0 certain features of Java 1.5 were used which were not yet present in previous versions of jSLP. Hence the use of generics and the replacement of certain Iterators with "for" constructs means that jSLP 2.0 requires Java 1.5 or higher to run.

One of the goals behind the development of jSLP 2.0 was modularity, allowing a user to only use those components required for the underlying application to which SLP functionality is to be added. This should happen in a transparent manner, without requiring any explicit deactivations or workarounds when certain optional components are missing. It should also be possible to easily combine the three SLP agent functionalities (user agent, service agent, directory agent) in the same framework. Finally this idea of modularity was also applied to the new directory agent, by decoupling the SLP features from the actual storage of the cached services. Figure 3.1 shows an overview of the jSLP design, the individual components are described below. Besides these main components there are a number of other classes that provide modularity, flexibility and extensibility.

### 3.1 SLPCore

At the center of the jSLP implementation is the SLPCore class, which contains all the components required for network communication. This core is required no matter which SLP functionality is to be used, however the core only initializes resources for communication as they are required. Furthermore the core executes the multicast convergence algorithm in a separate thread. The core also provides access to configuration data to the various other components of jSLP. Requests are initiated using the core's message sending methods, and replies are passed to the core by the handler. The handler writes replies to incoming requests
back to the session in which the request arrived, however the handler does not manage any actual networking resources. The core also instantiates the daemon classes for the SA and/or DA if they are requested by an application using jSLP. The SLPCore checks if the required classes for the UA, SA and DA are present. When an application requests an SLP agent, the core instantiates an object of the appropriate class if it exists. Any of the agent classes can be removed from the code and jSLP will still work, allowing jSLP to be deployed in reduced form when certain functionalities are not required. E.g., it is possible to remove the UA and DA code from jSLP before packaging it for use on a device which only wants to advertise its own service using the jSLP SA part.

### 3.1.1 Network Communication

As mentioned above, the core is where all network communication is managed and initialized. With the addition of a DA to the existing jSLP components it
was deemed necessary to provide the project with enhanced communication ca-
pabilities beyond those provided by the previous direct use of the java.net Socket
classes. Therefore, all communication now runs through the MINA framework.
This required the addition of a series of encoder and decoders to map between
byte buffers and the individual SLP messages, an implementation of IoHandler
to implement the SLP protocol logic and the addition of a range of Acceptors
and Connectors to the core to send and receive UDP and TCP messages as
required by the SLP agents being used.
As mentioned in the previous chapters, the available versions of MINA (2.0 RC1
at the time of writing) implemented TCP and UDP transport using the Java
NIO classes, which do not support multicast [2]). Yet SAs and DAs must be able
to receive multicast requests, and it can be useful for UAs to hear DAAdvert
messages sent over multicast. Therefore, MINA had to be made multicast-
capable. This was done by extending the early version of the transport-apr
package of the MINA project by adding AprDatagramAcceptor and AprData-
gramConnector classes to the existing TCP counterparts. Using APR for mul-
ticast communication is a serious handicap for the jSLP project, as it now relies
on platform-specific source code in the form of the APR library [3] itself as
well as the Tomcat Native interface \(^1\) to that library [5]. As all communication
code is localized in the core, this will make it easier to change the acceptor
and connector classes to later implementations e.g., based on the NIO2 libraries
available in Java 7.

3.2 SLP Agents and Services
jSLP provides UA, SA and also DA functionality. These SLP agents can be
used in any combination. This section describes the design of the three agents
as well as how SLP services are modelled in jSLP.

3.2.1 The Service Class
Early implementations of jSLP handled all attributes as name/value pairs of
strings. In version 2.0, Services store these values as values of the four types
deﬁned in [33] or as arrays of such values. Strings, integers and booleans are
stored as the corresponding Java basic types, yet opaque values are wrapped in
objects of the new OpaqueValue class extending the Comparable class, allowing
access to the string representation as well as the raw byte data. Furthermore
toString(), equals() and compare() methods were overridden in order to provide
seamless integration into the rest of the jSLP framework, which handled such
values as strings before. All this also allows for checks to be made on incoming
attributes in registrations, enforcing single type values as speciﬁed in [33]).

3.2.2 User Agent and Service Agent
The UA part offers an interface in the form of a Locator as before, sending out
requests through the core and receiving answers from there after they arrive.

\(^1\)During the process of this implementation of multicast transport using the Tomcat Native
code, a series of typographical errors were discovered and fixed in the Tomcat Native project
(see https://issues.apache.org/bugzilla/show_bug.cgi?id=46457).
However instead of returning ServiceLocationEnumerations as before, which offer blocking access to the replies to a request as Strings, in version 2.0 Reply-Futures are returned, which offer asynchronous access to replies as they come in, such that a client can start processing requests in the first round of a multicast convergence algorithm, without having to wait until the algorithm terminates. In addition, the new UA also supports the Attribute List Extension [15], therefore returning simple strings to a ServiceRequest no longer makes sense when this extension is used. Hence such lookups now return ReplyFutures that can be accessed as before, returning the URLs of discovered services, yet in addition full Service objects including their attribute lists can also be retrieved.

The SA is implemented in two parts: the Advertiser provides and interface to the application using jSLP, whereas the SLPDaemon handles incoming messages and manages the list of registered services and their timeouts. The SLPDaemon receives requests from the handler and processes them according to the SLP protocol, returning replies back to the handler which then sends them back out. If no DA is present, service registrations/deregistrations end here, however an SA can also be run with a DA within the same framework, in which case all registrations pass through the SA before they are passed through to the DirectoryAgentDaemon.

### 3.2.3 Directory Agent

The DA is new to version 2.0 and was implemented in a similar way to the SA. When a DA is deployed, it must be able to handle all SAs (registrations/deregistrations) and UAs (SLP requests) that use it for SLP exchanges. The jSLP design of the DA splits the DA’s job into two parts: one is processing messages and implementing SLP logic, the other actual storage and retrieval of registered services. The first part is accomplished by passing the SLP messages to the DA part once MINA finishes decoding messages from raw byte data into Java objects, just like for the UA and SA parts, hence the performance of MINA has a major impact on the overall performance of the jSLP DA. The SLP logic part then decides if a service has to be stored or retrieved or if an incremental update has to be made. It also handles service lifetime management as well as sending out ”DAAdvert” messages at the specified interval. Access to the service cache is performed through the ServiceStore interface. The implementation of the service store can then be developed to match any number of criteria and handles the actual storage and retrieval of services according to the parameters passed to it by the logic part, such as scope and attribute filters. The performance of this storage backend is thus also a major factor determining the overall performance of the DA. In this way the complexity of the DA as a ”server for SLP operation” is split into two more manageable chunks.

The interface to the application is provided by the DirectoryAgent class, whereas the DirectoryAgentDaemon implements the actual SLP logic. The daemon class processes messages it receives from the handler, sends out periodic advertisements and stores the services registered with it using the ServiceStore interface. This is basically just the logic required to run a DA, no actual storage is performed, and only SLP messages also understood by the SA are handled, there is no communication between individual DAs at this point. A DA can be started in the same framework as the SA (and a UA). Messages are processed by the SLPHandler, which passes them on to the various agents. When an agent sends
a message for which it expects an answer through the core, it registers this with the core. When the handler receives a reply message or a service acknowledgment message as a response to such a request, it passes it on to the listener via the core, bypassing the SA and DA daemons. Service requests, service type requests and attribute type requests are passed to the DA daemon if present, as all services advertised by the jSLP DA are also stored in the DA if one is present within the same framework. If there is no DA daemon present, the message is passed to the SA daemon. If there is also no SA daemon present, such a message can not be processed and is dropped. Registration and deregistration messages are a special case, as the SA component registers services by sending the framework a message. Therefore, the handler first checks incoming registration and deregistrations messages for their source. If they originated from the same host, they are passed to the SA daemon, which registers it for itself and then passes the message on to the DA daemon, if one is present. As a result, all three components (UA, SA, DA) can be used independently or in any combination.

3.3 Service Store

The components described in previous sections implement the SLP logic of jSLP. These new components introduced a number of improvements and extensions to the jSLP 1.0 implementation and already provide a number of benefits over the OpenSLP implementation. Yet the goal of the project was to provide an extensible, scalable high-performance directory agent without modifying the SLP protocol. In order to achieve this, the features and benefits of either the backend itself (Apache DS) or the framework on which it was running (Cirrostratus) were leveraged. This introduced another form of modularity into the jSLP project by separating the protocol implementation from the actual storage. The DirectoryAgent class in jSLP was developed to work against a backend implementing the ServiceStore interface, which provides storage and retrieval functionalities. Using the replication features of the actual backend, which is completely separated from the SLP logic, it is possible to have distinct independent directory agents tap into the same set of cached services by accessing a replicated ServiceStore implementation. The class implementing the ServiceStore interface must also ensure type consistency among attributes of specific SLP service types by managing a corresponding registry for attributes (see Section 3.3.2). This kind of modularity also provides further benefits. E.g., it is possible to implement new backends for jSLP based on any form of storage, such as an RDBMS backend, a file-based backend or even a web-based backend. This allows jSLP to be a bridge between any number of protocols or service management schemes, as the backend could be implemented to access the service cache of a different system, or it could provide a connection between SLP and UPnP, allowing SLP UAs to access services described and advertised using UPnP by sending service requests to the DA which then accesses the UPnP system. In addition, it is now possible to directly access the service cache, as it is no longer necessarily a structure internal to the SLP implementation (such as a HashMap in memory), but can be stored in the existing directory system of an enterprise or a database of resources. Thus an administrator can add, remove or modify services directly in the cache using tools such as an LDAP browser or a SQL
client. The following paragraphs describe the three initial backends developed during this project.

### 3.3.1 SimpleServiceStore

This first ServiceStore implementation does not provide any of the benefits mentioned above. It is a minimal and very simplistic implementation caching services in a (non-persistent) Hashmap just like the SA component does. All registered services are stored in memory and are lost on restart. The SimpleServiceStore, while being able to store and retrieve services quickly, does not provide any form of additional manageability or durability. On the other hand, the SimpleServiceStore does not introduce any additional dependencies and thus provides a way to distribute and deploy a minimal jSLP-based system which nevertheless includes the full set of SLP features. Running two DAs with the SimpleServiceStore backend in a network should still decrease individual lookup times for UAs, as they choose a DA to contact at random from the set of known DAs in a scope, distributing load across the DAs in the system. In addition to reducing code footprint and providing a basic no-frills backend, comparing lookup times against a jSLP DA using a SimpleServiceStore backend with those against OpenSLP and a jSLP DA using the ApacheDSServiceStore described below provided a way to isolate the impact of using the Apache DS as backend from the impact of using the MINA framework, as both the SimpleServiceStore and ApacheDSServiceStore backends run on the same SLPCore using the same communication framework.

### 3.3.2 ApacheDSServiceStore

The first complex ServiceStore implementation is based on the Apache Directory Server (ADS) [1]. The ApacheDSServiceStore (AD3S) uses an embedded ADS to store service registrations as LDAP entries. This choice was primarily made for two reasons: the relationship between SLP search filters and LDAP search filters, and the ability of ADS to replicate. The idea was to have the AD3S backends of individual jSLP DAs connect to each other and keep each other concurrent, basically having the DAs tap into a single replicated service cache. In addition, using such a backend allows administrators to access the service cache directly using an LDAP client. Using such a backend also allows for persistent storage of services, as long as the lifetime specified by the registration message is not exceeded. Unfortunately, the replication functionality of ADS is currently broken and was not correctly implemented in time to be used in this project. However, implementing the AD3S in the absence of SLP service templates introduced new problems that had to be addressed and solved.

### Caching SLP Services in the Absence of Templates and LDAP Schemas

In the specification of SLP the use of service templates to describe all available services in a network is proposed [13, 33, 16]. This may make sense in an environment where the services to be deployed are strictly controlled and known in advance, such as an enterprise network with policies governing the use of services in the SLP framework. Yet in smaller networks or networks with dynamic services, this may prove more of a problem, as discussed in section
2.1.3. The mapping between such SLP service templates and LDAP schemas used to describe the entries and their attributes in an LDAP directory has been proposed and is not that complicated due to the limited set of types in SLP [19]. The jSLP DA can be adapted to enforce compliance of service registrations to templates by adding checks to the backend without affecting the rest of the jSLP system, yet in the eyes of the author a DA should be able to accept any service registration without having to know in advance what kinds of service to expect, as the DA is just a cache of services used to remove the need for multicasting from the UAs and SAs. This however was initially a problem when hooking up the AD3S to the jSLP DA, as the attributes of LDAP entries of a certain objectClass are predefined in a schema. Therefore, the AD3S must know in advance what kind of attributes to expect in order to create entries of the objectClass "slpService", which would again require the use of service templates in SLP. One possibility would be to create a new objectClass dynamically for each concrete service type encountered during the operation of the DA and inject that schema into the directory. This would also mean that each time a new attribute for an existing service type shows up, that schema would have to be updated. The jSLP ApacheDSServiceStore implements a different approach made possible by the objectClass "extensibleObject" introduced with version 3 of the LDAP specification [35]. The attribute set of an entry of type "extensibleObject" is the set of all attributeTypes known to the directory. Thus an entry describing an SLP service in the AD3S has the objectClasses "slpService", which contains all the attributes present in all SLP services, such as lifetime and scope, plus the objectClass "extensibleObject" to cover all service-specific attributes.

This way every service registration translates to an entry in the LDAP backend as described above with all attribute types known to the server. Now the server just has to get to know attributes as they come in. Using the hierarchy of attribute types [36, 35] and the fact that SLP knows four types of attributes (keywords are considered to be strings by the LDAP backend), all service attributes that are not part of the SLP specification (such as lifetime or scope) are added to the LDAP schema as attribute types with a superior attribute i.e., as descendants of, one of "slpBooleanAttribute", "slpStringAttribute", "slpIntegerAttribute" or "slpOpaqueAttribute". These four attribute types are known to the LDAP backend and include all relevant matching and comparison rules. When the AD3S receives a registration of a service with attributes, it goes through all of them, parses their values and checks which of the four basic types they correspond to. Keyword attributes are stored as attributes of type String with a value of "jslpkeyword" throughout jSLP. Once the type of the attribute has been determined, the backend renames the attribute by prepending one of "slpInteger-", "slpBoolean-", "slpOpaque-" or "slpString-" to the attribute name accordingly. It then checks if such an attribute is known. If that is not the case a new schema element for the new attribute type is created. This makes it possible for an attribute name to be used by various service types. If the type of that attribute is the same be-

---

2 As a result of the implementation of this mapping between SLP and LDAP, a major bug in the (LDAP-Certified) Apache Directory Server was discovered and fixed. The Apache Directory Server compared attributes defined with the "Integer" syntax 1.3.6.1.4.1.1466.115.121.1.27 lexicographically instead of numerically, such that 42 was deemed larger than 417
tween service types, the existing attribute type is reused. If two (or more) service types have an attribute with the same name but of a different type, then this process creates to uniquely named attributes for storage and later lookup. As an example, if a DA with an AD3S backend receives a service registration for a service "service:osgi://myHost.com:123" with an attribute of 
"(owner=macgruber)" , then the backend will create a new attribute type named "slpString-owner" with a value of "macgruber" as a descendant of the "slpStringAttribute" attribute type. When later it receives a new registration for a service "service:printer://myOtherHost:321" with an attribute 
"(owner=42)" (obviously the printer service type keeps track of owners using their ids), then it will transform that to "slpInteger-owner" as a descendant of the "slpIntegerAttribute" type, preventing any sort of clash between the two service types and their "owner" attributes.

In order to maintain type consistency between services of the same abstract type, the ServiceStore implementations also include a type registry, mapping attributes for a given abstract service type to one of the four attribute types available in SLP. This mapping happens whenever a new attribute is encountered. Attributes of registrations are then checked against the registry, and if an attribute with that name has already been registered for the given abstract service type but with a different attribute type, then the DA or SA throws an INVALID_REGISTRATION exception and returns the corresponding error code in its acknowledgment message back to the SA and ultimately to the application on whose behalf the SA was registering the service. This registry should also be replicated along with the actual services in the cache. The ApacheDSServiceStore achieves this by simply managing the type registry as a set of LDAP entries, which are then also to be replicated among the LDAP backends participating in the replication process.

The downside of these transformations is that the direct use of the SLP predicate given in service requests can no longer be directly used to search for services, even though it uses LDAP syntax [33]. The AD3S backend therefore applies the same methods used to store services to transform each such predicate to match the attribute storage scheme described above.

1. Split the search string into individual attribute/value tuples of the form 
   (name operator value) where operator is one of "=", ">=" or "<=".
2. For each such tuple:
   - If <name> is one of the basic SLP "attributes" such as lifetime or scope, leave the tuple unchanged.
   - If there is no operator, only a name, then this is a keyword, replace it with a disjunction of all possible attribute names ("slpString-<name>", "slpBoolean-<name>", "slpOpaque-<name>" and "slpInteger-<name>") to be sure it is found if present in any form.
   - Else determine the type of <value> and rename <name> accordingly
3. Rebuild the search string using the transformed tuples and pass it on to the LDAP backend.

Once a matching entry is found by the backend, the entry must be transformed back into an SLP service, which is now straightforward, as the type of the
attribute can be read and stripped from the attribute name before adding it to the set of attributes of the service.

Using the above transformation scheme, the jSLP DA using the ApacheDSServiceStore can now store any service registration it encounters that is correct according to the SLP specification [33]. The same scheme can be applied anytime an LDAP-based backend is used, but a developer of a custom backend implementing the ServiceStore interface can use any other scheme. Indeed, if the use of service templates is desired, the backend can enforce adherence of service registrations to these templates.

3.3.3 CirrostratusServiceStore

As the replication feature of the Apache Directory Server was not available in time, a second ServiceStore implementation was designed as an OSGi service running on the Cirrostratus framework. As jSLP is also designed to be able to run as an OSGi bundle, this type of backend complements the ApacheDSServiceStore and shows the flexibility and extensibility of the jSLP DA. This would also be a proof-of-concept for the kind of applications that can profit from Cirrostratus as well as a test of the system, which was still being developed, debugged and fixed as the CirrostratusServiceStore was being built.

The CirrostratusServiceStore is basically a wrapper around a SimpleServiceStore which includes it in an OSGi bundle instrumented for Cirrostratus replication and registers it as a service when it is started.

![Figure 3.2: Overview of the jSLP DA with the CirrostratusServiceStore](image)

The development of this backend leverages the replication feature of the underly-
Designing framework without adjusting the SLP functionality, just like the ApacheDSServiceStore was to make use of the replication features of the backend it uses. In order to achieve this the CirrostratusServiceStore was developed as a bundle that registers an object that implements the ServiceStore interface. The service was then replicated to different nodes. The actual DAs were then deployed as separate bundles running on Cirrostratus nodes. These bundles fetched the CirrostratusServiceStore service and used it to store SLP services. As the service was replicated, when a DA on any node registered a service, this service became available to all other DAs using the same service.

The actual replication is achieved by storing services in a HashMap just like in the SimpleServiceStore. This HashMap is made public and during the startup process of Cirrostratus the CirrostratusServiceStore class is instrumented in a way that ensures changes made to the HashMap on one node are propagated to all other nodes onto which the service has been replicated. The type registry is implemented in the same way. Attributes are mapped to a type in a HashMap, which is also automatically instrumented and replicated by the Cirrostratus framework. Thus all DAs tapping into that service also share a common type registry and attribute type consistency is ensured not only locally on the individual DAs but also globally among the DAs running on the Cirrostratus framework.

3.4 Summary

jSLP 2.0 extends jSLP 1.0 with DA functionality and tries to inherit as much as possible of the previous modularity. In order to achieve this, the DA is separated into two parts, the SLP logic part and the actual storage part. Thus the storage part can be modified and implemented to suit certain goals and passed to the DA at runtime. To demonstrate the flexibility of this approach three distinct such storage components were implemented.

The SimpleServiceStore is a minimal implementation suitable for quick deployment of DAs without much configuration overhead. It keeps the footprint of the minimal (but complete) jSLP 2.0 implementation as small as possible. The SimpleServiceStore does not include any additional functionalities. This is the default ServiceStore for the jSLP 2.0 DA.

The ApacheDSServiceStore uses an embedded Apache Directory Server to store services as LDAP entries. The initial idea was to have individual DAs use one logical ServiceStore by having the ApacheDSServiceStore backends as replicas of one master. However the replication feature of the Apache Directory project was not completed in time. The implementation does however show how a specific ServiceStore can be adapted to specific needs, as it is very different from the basic SimpleServiceStore implementation. The ApacheDSServiceStore does however introduce a large number of dependencies and increases the project’s overall size. The implementation also provides a mapping of SLP service attributes to LDAP attribute types without using SLP templates, which allows a DA using this backend to cache SLP services dynamically without prior knowledge of the types of services and their attributes available in the network.

The CirrostratusServiceStore provides a replicated backend for jSLP DAs deployed on the Cirrostratus virtual OSGi framework. jSLP 1.0 was designed to be used both in a standalone version as well as in an OSGi environment. jSLP
2.0 is also available as an OSGi bundle using the SimpleServiceStore by default. The CirrostratusServiceStore makes use of the replication functionality of the Cirrostratus framework, allowing multiple DAs running on different nodes of the framework to tap into the same cache of SLP services. The three implementations above show how the jSLP 2.0 DA can be adapted to a range of scenarios without having to change the basic SLP part of the project. Additional logic, requirements, constraints or functionality can be added to the ServiceStore backend. Thus a backend could be developed to bridge SLP to any number of other service location protocols. Or a backend could be plugged into the DA that checks registrations for specific aspects, or injects new attributes as services are registered. The ServiceStore could store SLP services in any number of ways, be that as LDAP entries, in memory, persistently in a file or in a database. This also allows access to the cached services from outside the SLP protocol, by providing alternative access methods to the backend. While the actual SLP handling part of jSLP is kept as simple as possible, using a specific backend can tune the DA to suit a number of needs.
Chapter 4

Evaluation

This chapter describes the methods and materials used to evaluate the implementation of the jSLP directory agent. Results obtained from these methods will be shown in the next chapter.

Initial tests of the individual parts of the jSLP implementation were done at a low level, either by having all agents on the same host or by running them on two hosts. As the multicast code using APR did not work on the local Windows XP machines, all testing was done on hosts running various versions of Linux. Besides testing of correct adherence to the SLP protocol, the service storage backends were tested directly using jUnit. These tests are available in the project source code and are not discussed here. The UA and SA parts of jSLP 2.0 were also used in the presence of an OpenSLP directory agent to make sure interoperability was maintained during the transition from jSLP 1.0 to 2.0. The relevant evaluation for this project remains the performance of the new DA, especially when compared to the existing implementation in C in the form of OpenSLP. The following sections detail how this evaluation was devised.

Evaluation was performed in two main phases. In the first the response times for UAs querying specific DAs were evaluated and compared. After tests with the simple and ApacheDS DAs were run, some profiling was done to detect possible bottlenecks that could explain differences in performance. The second main phase involved the Cirrostratus-based DA and was designed to provide a quick view of the time it takes for a service inserted in one replica of the ServiceStore to be retrieved from another as well as how adding DAs to a network of SLP agents affected service request times. The following sections describe the procedures used in detail.

The results were visualized using either Gnuplot 4.2 [11] or directly from screenshots of tools used.

4.1 SLAMD

The major component used in these evaluations was the SLAMD distributed load generator [31]. This Java application comes in two parts, a server and a series of clients. Version 2.0.0-alpha1 was used for these tests, which was the newest version at the time of writing. SLAMD provides a server on which various jobs, which are also written in Java, can be scheduled. Once sched-
uled these jobs are then executed on the specified number of SLAMD clients. SLAMD jobs allow the collection of statistics using a series of trackers. For the jobs used to evaluate the jSLP DA two trackers were used, a TimeTracker to record the duration of lookups and an IncrementalTracker to count the number of successful and unsuccessful lookups. The TimeTracker implementation in SLAMD has millisecond resolution and stores events in a minimal interval of 1 second. Clearly this does not provide any exact measurements on the actual time it takes for a UA to receive an answer from a DA. However, this was deemed sufficient for two reasons. For one, this was an initial estimate of the performance capability of the jSLP DA using the various backends as well as the new MINA framework. Secondly all measurements were made at the client side and involved the message crossing the network to reach the DA, the DA processing the request and returning the result, which then had to return back across the network to the UA, such that resolutions below the millisecond threshold were deemed unnecessary under such circumstances.

All SLAMD benchmarks were carried out on the cluster provided by the ETH. In specific machines ikr20 through ikr32 were used, which had the following technical specifications:

- 3.6GHz Dual-Core AMD Opteron 275
- 4 GB DDR RAM
- Broadcom Corporation NetXtreme BCM5704 Gigabit Ethernet

The machines were interacted with using SSH (or CSSH) and scp for file transfers. Files were uploaded into the home directory and were thus always consistent between the various hosts. The jSLP implementation was left unchanged during the execution of the individual evaluation phases. However a small bug was fixed between the single-DA evaluations and subsequent multi-DA tests. The following components were used during the benchmarking phase (snapshots were left unchanged during the course of the benchmarking process):

- jSLP 2.0 (SNAPSHOT) \(^1\)
- OpenSLP 2.0 \(^2\)
- Apache Directory Server 1.5.5 (SNAPSHOT)
- Java 1.5.0_16
- MINA 2.0.0 RC1 (SNAPSHOT) \(^3\)
- Tomcat Native 1.1.16 \(^4\)
- Equinox OSGi framework 3.2.2.R32x_v20070118
- Cirrostratus (daily build) \(^5\)

\(^1\)in the OSGi configuration the ApacheDSServiceStore classes were removed
\(^2\)including a small patch to allow OpenSLP to run on a user-defined port
\(^3\)including additional custom classes for multicast support
\(^4\)after various typographical errors were fixed
\(^5\)various bugs were discovered and fixed during the evaluation process
4.2 Response Times for Clients Unicasting to a Single DA

This phase of the evaluation process was aimed at providing an overview of the basic performance of the jSLP DA with a different ServiceStore backends compared to the OpenSLP implementation. All times include the time it takes for a message to pass the MINA layer on it’s way out, be transferred across the network to the DA and get processed there plus the time it takes for the reply from the DA to cross the network, get passed through the MINA framework of the UA and get processed there. Measuring times directly at the DA was not done since it would involve making changes to the OpenSLP and jSLP DA code. This way all times could be measured with the same jSLP UA regardless of the DA implementation. The only changes made to the UA code included the possibility to force it to contact a specific DA, no matter what else happens during the lookup process. The ability to force a DA list on an SLP client can already be done with the right configuration, yet for testing purposes the UA was no longer allowed to drop DAs if a request timed out.

4.2.1 Jobs

A series of jobs were used to have clients send a row of requests to the DA with varying amounts of required processing by the DA. The jobs were run by SLAMD clients running on JVMs of the same version as the server and with access to the same libraries. Table 4.1 provides a quick overview of the eight jobs used to evaluate the DA in a first step, and the following sections provide further details. For all jobs the presence of a DA with a number of registered services as described in section 4.3.1 was expected.

All jobs made use of the jSLP library by requesting and using a Locator instance to send out requests. However, for the duration of these SLAMD tests, the IP address of the DA was hardcoded into the Locator’s list of known DAs, ensuring communication with only the known DA and preventing DA lookups in case something went wrong with a request (e.g. a timeout), which would normally remove the DA from the list.

These jobs start a timer and send out a specific request and wait for the result. When the result arrives or the request times out the timer is stopped. Then the result is compared to the expected outcome based on the known contents of the DAs service cache. If the result matches the expected one, a success counter is incremented, if the request times out a failure counter is incremented, and if a reply arrives but its contents does not match the expected one, an internal error counter is incremented. Once the result has been processed, the next request is immediately sent to the DA. An internal error is also counted when a reply with an error code not equal to zero is received.

All requests are repeated 1’000’000 times and are started simultaneously. An initial warmup time of 5s in which no results were processed, timers used or counters incremented was used to make sure all clients were ready and sending when statistics were started to be collected. It was also expected that this would get as much as possible cached in the DAs. Most clients seemed to complete their operations roughly at the same time, however two clients seemed to be consistently much faster that the others, possibly due to proximity to the
<table>
<thead>
<tr>
<th>Job Name</th>
<th>Request Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sr_basic</td>
<td>Service Request</td>
<td>Client repeatedly sends out a request for the same existing concrete service type registered at the DA and expects three URLs in the service reply. The search predicate is the empty string.</td>
</tr>
<tr>
<td>sr_random</td>
<td>Service Request</td>
<td>Client sends out a request for a random choice of one of the existing concrete service types registered at the DA and expects three URLs in the service reply. The search predicate is the empty string.</td>
</tr>
<tr>
<td>sr_nonexist</td>
<td>Service Request</td>
<td>Client sends out a request for a random choice of one of the existing concrete service types registered at the DA and expects three URLs in the service reply. Every fifth request however is for a non-existing service type, in which case an empty result is expected. The search predicate is the empty string.</td>
</tr>
<tr>
<td>sr_predicate</td>
<td>Service Request</td>
<td>Client sequentially sends out a request for one of the existing concrete service types registered at the DA and expects URLs in the service reply according to the predicate. The predicate used searches for either even or odd numbered hosts.</td>
</tr>
<tr>
<td>ar_basic</td>
<td>Attribute Request</td>
<td>Client sends out an attribute request for a service registered at the DA and expects all attributes for that service to be in the Attribute Reply.</td>
</tr>
<tr>
<td>ar_nonexist</td>
<td>Attribute Request</td>
<td>Client sends out an attribute request for a service registered at the DA and expects all attributes for that service to be in the attribute reply. However, every fifth request is for the attributes of a non-existing service, in which case the empty string is expected as a result.</td>
</tr>
<tr>
<td>ar_tags</td>
<td>Attribute Request</td>
<td>Client sequentially sends out an attribute request for the services registered at the DA using a tag to limit the attributes returned to &quot;even&quot; and &quot;hostNumber&quot;. The client expects those two attributes for that service to be in the Attribute Reply.</td>
</tr>
<tr>
<td>str_basic</td>
<td>Service Type Request</td>
<td>The client sends out a service type request and expects a list of all service types known to the DA to be returned.</td>
</tr>
</tbody>
</table>
server. Another problem showed up when running the jobs on 5 clients. After approximately 7,000,000 requests the host would run out of memory and kill the Java process. It is assumed that there is a memory leak either in the new AprDatagramConnector code or the underlying APR or Tomcat-Native libraries. Due to a lack of time the leak could not yet be tracked down, therefore all jobs running on 10 clients were changed to send only 500,000 messages instead of 1,000,000.

Service Request Jobs

The first set of jobs build service request messages that are sent to the DA, eliciting a service reply to be sent back to the SLP/SLAMD client. These jobs were used to test the basic service lookup function of SLP. It is assumed to be the main type of interaction between UAs and DAs. Having the DA process these messages tests the simplest form of lookup a DA must be able to handle as the concrete service type is provided.

sr_basic: Identical requests for existing services
This job type sends out a service request for service "service:osgi:type0" and expects the reply to contain the URLs of the three services registered with the DA:

- service:osgi:type0://host0:100
- service:osgi:type0://host1:100
- service:osgi:type0://host2:100

As this job keeps sending out requests for the same service type without any filters (except scope handling by the DA), it was expected that this job’s result would demonstrate the performance difference between MINA and the networking code used by OpenSLP as actual processing was minimal. It was expected that the three jSLP DAs would provide almost identical response times as they use the same networking and message handling code. Differences in lookup time were expected to be minimized by the warmup period allowing the ApacheDSServiceStore’s embedded ADS to fill its caches.

sr_random: Random requests for existing services
This job also sends out service requests, however this time it randomly builds the request by choosing an abstract type as one of "printer", "osgi", "toaster" or "brewery" and a concrete type as one of "type0", "type1", "type2", "type3" or "type4". It then sends a service request for all services of that randomly built type, expecting a reply containing the URLs of the three hosts providing such a service. It was expected that this would lead to about the same results as for the sr_basic job, as the limited number of service types used should not cause much overhead.

sr_nonexist: Random requests for existing and non-existing services
This job builds requests in the same way as job sr_random described above, however now every fifth request is for a service of type "service:nothinghere". No services of this type are registered with the DA, hence a reply containing the empty string is expected. Including a non-existant service type for lookups was
Evaluation

not expected to have an impact, but was included to cover more of the cases that a DA can expect to encounter when operational.

**sr_predicate: Sequential requests for existing services with predicates**
This job adds predicates to service requests. Requests are sequentially built by cycling through all abstract ("printer", "osgi", "toaster", "brewery") and concrete ("type0", "type1", "type2", "type3", "type4") types. In addition to the service type the request also includes a predicate, which can be one of:

- 
  
  
  
  
  "(&(even=true)(multi=even))"

- 
  
  
  
  
  "(&(even=false)(multi=odd))"

- 
  
  
  
  
  "(&(even=true)(multi=odd))"

In the first case the job expects the reply to contain the URLs of the two even numbered hosts providing that service, in the second case that of the odd numbered host is expected and in the final case the reply is expected to contain no URLs at all, as no service with that combination of attribute values is stored. Being able to find services using attributes is one of the major advantages of using SLP and therefore this job tests the performance of the DAs when attribute filtering has to be done. An increase in response times was expected when comparing the results from this job to the previous ones. This job was also designed to examine the performance of the ApacheDSServiceStore, as more complex LDAP lookups have to be performed to find service entries matching the given filter. Processing in this case includes matching of the single-valued "even" attribute as well as the multi-valued attribute "multi".

**Attribute Request Jobs**
This second set of jobs sends out attribute request messages, expecting attribute reply messages to be returned by the DA. Retrieving the attributes of a specific job is one of the browsing features of SLP and also expected to be used frequently when clients browse for matching services.

**ar_basic: Identical requests for existing services** This job class sent out attribute requests for the service "service:brewery:type0://host0:400" expecting an attribute reply containing all the attributes of that service and their values:

- 
  
  
  
  "(even=true)"

- 
  
  
  
  "(hostName=foo-0)"

- 
  
  
  
  "(hostNumber=0)"

- 
  
  
  
  "(multi=even,0string)"

This was designed to be a baseline for other attribute request jobs, as it should measure the time it takes to retrieve the attributes of a specific service without any additional processing. The warmup time was again expected to minimize the difference in performance between the various jSLP DAs.
ar_nonexist: Random requests for existing and non-existing services
These jobs built a random attribute request messages by randomly choosing an abstract ("printer", "osgi", "toaster", "brewery") and a concrete ("type0", "type1", "type2", "type3", "type4") type as well as with a random pick of one of the available hosts ("host0", "host1", "host2"). The job then expected to get values for all attributes with the correct values in an attribute reply, depending on which host was selected. Every fifth such request was for the attributes of the non-existing service "service:nothingthere://host0:100", in which case an attribute reply with an empty result string was expected. Adding requests for non-existing services was expected to slightly reduce response time as the DA should send back an empty response when it doesn’t find a service.

ar_tags: Sequential requests for existing services with tags
In this job attribute requests are sequentially built by cycling through the abstract ("printer", "osgi", "toaster", "brewery") and concrete ("type0", "type1", "type2", "type3", "type4") types as well as the available hosts ("host0", "host1", "host2"). In addition the tags "even,hostNumber" are provided with the request. The expected attribute reply should contain only the two attributes listed in the tags and these should have the correct value depending on the host selected while building the request. This job was designed to force the DA to perform some additional processing on retrieved services in order to only add the requested attributes to the reply. This is another job that should highlight the differences between the SimpleServiceStore and the ApacheDSServiceStore, as this processing is handled by the backend.

Service Type Request Job
This final job type was designed to return a large payload and force the DA to have to process a large part of its cache into a message. Service type requests also belong to the browsing set of SLP features. High rates of such messages are not expected to be encountered in actual operation scenarios, as these features are expected to be used interactively by users browsing the SLP system for possible service matches.

str_basic: Service type requests
The job numbered str_basic is a service type request with no naming authority specified. This request should elicit the DA to respond with a service type reply including all service types available in its cache, which in this case would be 20 services (4 abstract types combined with 5 concrete types). This job was added to the suite as these messages can lead to large responses depending on the number of service types stored in the DA and how specific the request is. This job should again show the difference between the jSLP backends because the backend is now forced to parse a number of entries from the internal storage form to the Service object that the SLP logic part of the DA expects.

4.3 Setup
The evaluation was run by starting the SLAMD server on the ikr20 machine of the cluster, the directory agent to be tested against on ikr21 and the SLAMD
Evaluation

clients on 1, 5 or 10 machines on machines in the range from ikr22 to ikr31.

4.3.1 Directory Agents
This section describes in detail how the directory agents were initialized. Service and service type names bear no significance, they just introduce a series of services with varying abstract and concrete types into the system. The attributes were chosen to cover a range of possible attribute types. Services have a single-valued boolean attribute, a single-valued integer attribute, a single-valued string attribute and a multi-valued string attribute. This was done to be able to force the DAs to perform varying degrees of attribute processing. The actual names and values of these attributes are listed below for reference, yet they bear no additional significance.

The directory agents were run on the ikr21 machine. Only one DA was active at any one time during the initial single-DA comparison phase. All DAs were configured to use port 1337 for SLP, not the standard SLP port of 427. A series of 15 services of each of the following types were registered with the DA for testing, where X ranges from 0 to 4 and Y from 0 to 2:

- `osgi:typeX//hostY:100`
- `printer:typeX//hostY:200`
- `toaster:typeX//hostY:300`
- `brewery:typeX//hostY:400`

Services on even numbered hosts (0 and 2) had an attribute "(even=true)" while the odds ones had "(even=false)". In addition all services had an attribute "(hostNumber=Y)" and "(hostName=foo-Y)". Also all services on a host whose number was 2 had an additional attribute "(two=yes)". Finally all services had a multivalued attribute "(multi=[even|odd], Ystring)" where Y was again the host’s number and the first attribute was either "even" for even-numbered hosts or "odd" for odd-numbered ones. As an example there was a service "service:osgi:type3//host1:100" with the attributes "(even=false)", "(hostNumber=1)", "(hostName=foo-1)" and "(multi=odd, 1string)".

The four types of DA used were OpenSLP 2.0 using the default configuration with the exception of the port, a jSLP DA with a SimpleServiceStore backend, a jSLP DA with an ApacheDSServiceStore and a jSLP DA with the CirrostratusServiceStore. The only difference between the two standalone jSLP directory agents (SimpleServiceStore and ApacheDSServiceStore) was the backend used. These two jSLP DAs and the OpenSLP DA were simply started in the default way. The services that had to be initially registered were provided to OpenSLP as a .reg file and programatically added to the jSLP Directory Agent by the main application class. The CirrostratusServiceStore DA used was deployed on the Equinox OSGi framework version as described in Section 3.3.3.

4.3.2 Clients
The SLAMD clients used were identical and used the same jSLP implementation. Even though all ikr machines had the same hardware, tests suggested that ikr25 and ikr26 were often the first to complete SLAMD jobs, possibly due to
proximity to the server. The clients used a slightly modified version of the jSLP implementation: DAs were not removed from the list of known DAs when a timeout occurred during a request and the IP address of ikr21 was hardcoded into the Locator class.

4.4 Profiling

After the SLAMD benchmarking was completed, a series of profiling runs were made to assess where the difference in performance between the ApacheDSServiceStore DA and the other two jSLP DAs was caused. This was done using JProfiler version 5.2.1 [10]. The same setup as listed above was used, with the SLAMD server on ikr20, the DA on ikr21 setup for remote profiling and the clients unchanged on ikr22 through ikr31. The profiling run was done with SLAMD job sr_predicate (service requests with predicate). Profiling was done over approximately 10 minute intervals after which the relevant reports were generated by JProfiler and converted into JPEG images by taking screenshots.

4.5 Cirrostratus-Specific Tests

After the basic response time evaluation described above was performed, additional evaluation was carried out using the Cirrostratus-based jSLP DA. As the ApacheDS-backed DA did not provide the replication functionalities the jSLP project was to make use of, the Cirrostratus-based DA was used as an example of a jSLP DA with a replicated ServiceStore as a possible candidate for a high-performance DA or set of DAs. However, as the Cirrostratus framework was still very much work in progress, this was to be viewed as a proof-of-concept evaluation, not the evaluation of a deployable product.

4.5.1 Cirrostratus Nodes and DAs

This series of tests required more than one DA using the same CirrostratusServiceStore service. In order to achieve this, three nodes were prepared on the following machines and named as listed (with the names of the DAs running on them in parentheses):

- ikr21.inf.ethz.ch -> bootstrap node (DA1)
- ikr31.inf.ethz.ch -> node2 (DA2)
- ikr32.inf.ethz.ch -> node3 (DA3)

Since all machines in the cluster accessed the same files in the author’s distributed home directory, each node was provided with an individual subdirectory and its own set of files and configuration directories. The nodes were initialized by starting the Equinox OSGi framework on which then the necessary bundles were installed: jSLP, logging, MINA and Cirrostratus. A bundle that replicates all local services running on Cirrostratus after 20 seconds was also installed on the bootstrap node. The jSLP bundles and their dependencies had to be installed locally as the SLPCore had to be instantiated locally for each node. Therefore these bundles were hardcoded into the
Cirrostratus code as bundles to be ignored for resolution as they were assumed to be present on any node running Cirrostratus. Subsequently the SLP and Cirrostratus bundles (plus the replication bundle on the bootstrap node) were started. The OSGi framework was then stopped and the configuration directory backed up. Later, initializations of Cirrostratus nodes then replaced the configuration directory of the OSGi framework with the backed up version to ensure a fresh start each time. The CirrostratusServiceStore (also called VirtualOSGiServiceStore) was then installed and started on the bootstrap node, which registered an implementation of the ServiceStore interface similar to the SimpleServiceStore as a service. The Cirrostratus code in its current state includes a list of classes that are to be replicated and the CirrostratusServiceStore implementation was hardcoded into that list, causing Cirrostratus to instrument the code for replication. The service was then automatically replicated to the other nodes after 20 seconds. An independent jSLP directory agent bundle was installed and started on each node, consuming the replicated service and using it as a ServiceStore backend, thus having all DAs logically access the same cache of SLP services.

4.5.2 Registration Propagation

An initial evaluation of the CirrostratusServiceStore-backed DA tried to determine how long it takes for a registration made to a DA on a specific node to be retrievable from a DA on a different node using the same CirrostratusServiceStore service. To accomplish this a SLAMD job was developed to register a service with an incremental component in its name at specific intervals (1000ms, 500ms, 250ms and 100ms) to the DA on the bootstrap node and at the same time constantly send service request messages for that specific service to the DA on node2. Times were compared by adding an attribute containing a timestamp to the service before registration and comparing that attribute’s value with the current time when the reply gets back. In order to prevent multiple messages being sent back and forth to get the attributes for the service, the attributes were directly included in an Attribute List Extension. This time included processing times and transfer times and did not reflect the actual time it takes for a service registration to be propagated between replicas of the CirrostratusServiceStore service. It did however provide a rough estimate of said time interval. This was done with 2 replicas (bootstrap node and node2) as well as with three replicas (by adding node3), with the SLAM job always contacting the same nodes. It was also the intention to repeat this with more than one SLAMD client registering and requesting services, yet the Cirrostratus replication mechanism proved yet to be too unstable and could not cope with that frequency of updates to replicated services.

4.5.3 Lookup Times with Varying Numbers of Directory Agents in the Network

In order to show how having multiple DAs in a network affects the times it takes for a jSLP user agent to request a service, jobs sr_predicate, ar_tags and str_basic...
were repeated with 10 user agents supported by 2 and 3 jSLP DAs running on the Cirrostratus framework and using a replicated CirrostratusServiceStore service as a backend. In a first run each job was run with 500’000 messages per client against a static number of DAs (2 or 3). Finally the same jobs were run with an initial set of 3 DAs, yet after approximately seven minutes the DA on node3 was shut down. After about another seven minutes the DA on node2 was also shut down. In order to prevent the user agents from entering the multicast convergence algorithm due to timeouts occurring at the switch to a single DA, the address of the bootstrap node was inserted into the list of DAs known to a jSLP user agent when the list was empty.
Evaluation
Chapter 5

Results

This chapter presents the results obtained from the evaluation of the jSLP directory agents with the various backends. The first section deals with the comparison of the jSLP DAs with OpenSLP and each other while the second section details the evaluation of the Cirrostratus-based DA that includes the desired replication functionality.

5.1 Individual Directory Agent Comparisons

The first series of tests run compared the lookup times of jSLP UAs using a range of request messages sent to OpenSLP (openslp), the jSLP DA with the SimpleServiceStore backend (simple), the jSLP DA with the ApacheDSServiceStore backend (apacheds) and the jSLP DA with the CirrostratusServiceStore (also named VirtualOSGiServiceStore) backend (cirro).

Figure 5.1 shows similar response time curves for all four jobs using service requests, even when a simple filter is applied as in job sr_predcate. It also shows that the OpenSLP DA responds quicker on average in all scenarios. However, the results also show high variability in the resulting response times.

Figure 5.2 shows the same behaviour for attribute request jobs. Again the OpenSLP DA answers faster on average in all scenarios, with the SimpleServiceStore DA being slightly slower on average. Again the ApacheDSServiceStore DA performs the worst.

Figure 5.3 finally shows that the results seen in the case of service requests and attribute requests are repeated in the service type request job. When five or even ten clients send requests to the DA a massive increase in the standard deviation of the lookup times was observed.

Figures 5.1 through 5.3 suggest that the SimpleServiceStore and CirrostratusServiceStore backends perform comparably.

The following sections provide some more detailed results for one job for each SLP request type.

5.1.1 Details for Job sr_predcate (Service Request)

Figure 5.4 shows that there is a lower-frequency component of noise in the lookup times of the ApacheDS DA when compared to the other two. It also
Figure 5.1: Average response times over 1s intervals for Service Request jobs using 1, 5 and 10 clients against the OpenSLP DA, and the jSLP DA with a SimpleServiceStore, ApacheDSServiceStore and CirrostratusServiceStore back-end (+/- 1x standard deviation). The standard deviation error bar for the cirro data in c) is truncated, full value is 311.819 shows that in all cases there are a number of extreme cases of lookups well above the general trend over time, most notably the 17ms average value towards the end of the experiment with the OpenSLP DA (Figure 5.4(a)) and the various spikes around 5ms throughout the course of the measurement for the ApacheDS based DA (Figure 5.4(c)). The curves for the SimpleServiceStore (5.4(b)) and the CirrostratusServiceStore (5.4(d)) look very similar.

Figure 5.5 shows an increase in noise for all three DAs. A series of increases in average lookup times over approximately 200s followed by a drop in the delay can be seen at the beginning of the experiment in Figure 5.5(b). The frequency of these increase/drop events increases towards the end of the experiment.

Figure 5.6 shows a further increase in the variability of the average response times for the ApacheDS DA (Figure 5.6(c)), however in the case of the other two such an increase is not clearly visible. The low-frequency component of variability can again be seen in Figure 5.6(b) for the SimpleServiceStore DA, however the time between two drops in lookup time i.e., the length of the period
5.1 Individual Directory Agent Comparisons

5.1.1 Details for Job ar\_basic (Attribute Request)

Figure 5.2 shows the variability of the average delays between an attribute request and the corresponding reply averaged over 1s intervals. It shows that in certain specific intervals extreme average values were recorded, as can be seen by the spike about half-way through the experiment using the OpenSLP DA (Figure 5.7(a)), which is also the maximum average value recorded for all four DAs. As with job sr\_predicate, Figures 5.7 and 5.9 suggest similar response times for the SimpleServiceStore and CirrostratusServiceStore DAs.

Figures 5.8 and 5.9 show an increase in noise. A low-frequency component is again visible for the jSLP DAs and is best observed in Figures 5.9(b) and 5.9(c).

Figure 5.2: Average response times over 1s intervals for Attribute Request jobs using 1, 5 and 10 clients against the OpenSLP DA, and the jSLP DAs with a SimpleServiceStore, ApacheDSServiceStore and CirrostratusServiceStore back-end (+/- 1x standard deviation).

5.1.2 Details for Job ar\_tags (Attribute Request)

Figure 5.7 shows the variability of the average delays between an attribute request and the corresponding reply averaged over 1s intervals. It shows that in certain specific intervals extreme average values were recorded, as can be seen by the spike about half-way through the experiment using the OpenSLP DA (Figure 5.7(a)), which is also the maximum average value recorded for all four DAs. As with job sr\_predicate, Figures 5.7 and 5.9 suggest similar response times for the SimpleServiceStore and CirrostratusServiceStore DAs. Figures 5.8 and 5.9 show an increase in noise. A low-frequency component is again visible for the jSLP DAs and is best observed in Figures 5.9(b) and 5.9(c).
5.1.3 Details for Job str_basic (Service Type Request)

This job type had the highest average of all 8 jobs. This was expected as the request led to the retrieval of many services from the cache and their processing into a reply. Again an increase in variability of the average response times (averaged over 1s intervals) with an increasing number of clients can be seen in Figures 5.10, 5.11 and 5.12. This job also lead to the highest variability in response times for the ApacheDS-backed DA, the extent can be seen in Figures 5.10(c), 5.11(c) and 5.12(c). Figure 5.3 shows this variability, however in order to still be able to see the other two curves, the standard error bars were cut at y=0 and y=20. The actual values were 126.597 for 5 clients and 128.753 for 10 clients. Peaks in average lookup time towards the end of the experiments with 5 and 10 clients respectively are clearly visible in Figures 5.11 and 5.12. These were caused by about 20s periods in which lookups took around 5s. The UA resends a request after a timeout of 2s, hence this reply required the UA to send three requests. Figure 5.13 shows a "zoomed" view of the lookup times of UAs querying the ApacheDS DA. Comparing Figure 5.13 with Figures 5.11 and 5.12 shows that the ApacheDS-backed DA indeed produced stronger fluctuations in response time than either of the other two DAs. Using this SLAMD job seemed to lead to similar response times and fluctuations for the SimpleServiceStore and CirrostratusServiceStore DAs (Figures 5.10 through 5.12), however the response time curve in Figure 5.11(b) seems more noisy than the curve in Figure 5.11(d), while the latter shows more spikes to higher values.

The comparisons mentioned above, even with the small number of clients deployed, suggest that the embedded ApacheDS does not scale well at all as the backend for a DA. The linear increase of the average response time from 1 to 10 clients does not bode well for a further increase in the number of clients. The high variability also suggests that increasing the load even further may lead to a number of timeouts. A part of the high standard deviation values for certain jobs may be attributed to a few individual periods with consecutive long response times, possibly caused by some other traffic happening on the network.
5.1 Individual Directory Agent Comparisons

(a) OpenSLP

(b) SimpleServiceStore

(c) ApacheDSServiceStore

(d) CirrostratusServiceStore

Figure 5.4: Details of lookup times averaged over 1s intervals for all four DAs with 1 UA sending service requests

or by something caused by the JVM or on a layer below the JVM. However such events occurred for all four DAs and therefor the ApacheDSServiceStore still clearly performs a lot worse than the other two jSLP DAs. Unfortunately the replication feature of the Apache Directory Server was not available at the time this evaluation was made. A personal communication by one of the main developers in the Apache Directory project mentioned that the final 1.5 (or 2.0) release of the Apache Directory Server is expected to have at least a twofold increase in performance compared to the state at the time of writing. If this is truly achieved, then using the ApacheDSServiceStore may actually lead to response times comparable to the ones achieved with the other two backends used for this evaluation. If replication also works then deploying multiple DAs with a replicated ApacheDSServiceStore backend may indeed be a viable solution for a scenario that requires high-performance transparent service discovery. This would also be scalable as increasing load can be caught by deploying a new DA with a replicated ApacheDSServiceStore backend.

The SimpleServiceStore backend without any replication functionality seems to provide similar performance to the OpenSLP implementation. This was a positive surprise, as the use of the MINA framework with the APR transport
classes that are still not included in an official release of MINA was suspected to have a greater negative affect on performance. Messages passing through the MINA framework pass through a filter chain and decoders, which was thought to have a significant impact on the performance of the jSLP DA when compared to the OpenSLP implementation in C. The results however suggest that the SimpleServiceStore-based DA performs adequately in the face of its C-based rival.

Placing the jSLP DA on the Cirrostratus framework was not thought to introduce additional lag, as the framework only manages bundle dependencies and replication, which should not affect the performance of a non-replicated DA handling requests once it is started on a node. The results back this hypothesis, as the average response times and variability is similar to the ones seen in the presence of the SimpleServiceStore DA. The difference in performance between the ApacheDSServiceStore-based DA and the other two jSLP DAs was thought to be caused by the backend, not the MINA framework or the use of Java as a programming language, as the two DAs with adequate performance used the same MINA infrastructure and SLP code and differed only in the part used to store and retrieve services. After an initial test run the transformation of SLP
services to LDAP entries and SLP predicates to appropriate LDAP search filters was thought to reduce the performance, so these parts were tweaked before the results listed above were gathered. In order to determine if the bottleneck was indeed in the jSLP part of the code or the ApacheDS part, further jobs were run while JProfiler was running remotely to gather information on CPU time and memory allocations. These results are described in the next section.

Based on these results the further development of the Cirrostratus-based DA was pursued in order to determine if the replication features of the framework would help produce a scalable jSLP DA with a replicated service cache consistent among multiple DAs. This lead to further work on the Cirrostratus framework, as the instrumentation and replication of classes by the Cirrostratus framework was not working properly. Once replication worked, even if in an unstable fashion, the CirrostratusServiceStore was replicated and used by individual DA bundles, the results of which are described later in this chapter.
Figure 5.7: Details of lookup times averaged over 1s intervals for all four DAs with 1 UA sending attribute requests.
5.1 Individual Directory Agent Comparisons

Figure 5.8: Details of lookup times averaged over 1s intervals for all four DAs with 5 UAs sending attribute requests
Figure 5.9: Details of lookup times averaged over 1s intervals for all four DAs with 10 UAs sending attribute requests
5.1 Individual Directory Agent Comparisons

Figure 5.10: Details of lookup times averaged over 1s intervals for all four DAs with 1 UA sending service type requests
Figure 5.11: Details of lookup times averaged over 1s intervals for all four DAs with 5 UAs sending service type requests
5.1 Individual Directory Agent Comparisons

Figure 5.12: Details of lookup times averaged over 1s intervals for all four DAs with 10 UAs sending service type requests

Figure 5.13: A “zoomed in” view of lookup times averaged over 1s intervals for the ApacheDS backend with 5 and 10 UAs sending service type requests
5.1.4 Profiling Results

Figures 5.14 and 5.15 show the first entries of the hotspot view of CPU time as provided by JProfiler [10]. The profiling was done for the ApacheDSServiceStore and the SimpleServiceStore with 1, 5 and 10 clients unicasting to a single DA. The screenshots in Figure 5.14 show how with increasing load more and more time is spent in the search method of the ApacheDSServiceStore’s embedded Apache Directory Server. The SimpleServiceStore on the other hand shows a similar distribution of CPU time with increasing load, as shown in Figure 5.15. Figure 5.16 shows that the threads processing messages (pool-16 threads) are often in blocking state with an embedded Apache Directory as a backend. This does not happen that often in the case of the simple jSLP DA as shown in Figure 5.17 (pool-15 threads).

![Figure 5.14: Profiling of the ApacheDS-backed DA over 10 minutes with 1, 5 and 10 UAs](a) 1 Client

![Figure 5.15: Profiling of the SimpleServiceStore over 10 minutes with 1, 5 and 10 UAs](b) 5 Clients

![Figure 5.16: Profiling of the SimpleServiceStore over 10 minutes with 1, 5 and 10 UAs](c) 10 Clients

These results suggest that the bottleneck leading to reduced performance of the ApacheDSServiceStore when compared to the other two jSLP DAs was located in the ApacheDS part of the code. The profiling data suggests that searching for an entry within the ApacheDS backend is slow and requires more and more CPU time as load increases, whereas the lookups in the simple DA are rather quick.
5.1 Individual Directory Agent Comparisons

<table>
<thead>
<tr>
<th>Hot spot</th>
<th>Inherent time</th>
<th>Average time</th>
<th>Invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.concurrent.ThreadPoolExecutor$Worker.run</td>
<td>508 ms (98%)</td>
<td>437465 ms</td>
<td>14</td>
</tr>
<tr>
<td>java.lang.String.replace</td>
<td>864 ms (0%)</td>
<td>74 μs</td>
<td>11927</td>
</tr>
<tr>
<td>org.apache.mina.core.session.TSession.write</td>
<td>481 ms (0%)</td>
<td>232 μs</td>
<td>1939</td>
</tr>
</tbody>
</table>

(a) 1 Client

<table>
<thead>
<tr>
<th>Hot spot</th>
<th>Inherent time</th>
<th>Average time</th>
<th>Invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.concurrent.ThreadPoolExecutor$Worker.run</td>
<td>320 ms (53%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.String.replace</td>
<td>88233 ms (8%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.StringBuilder.concat</td>
<td>37114 ms (3%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.StringBuilder.toString</td>
<td>31227 ms (3%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.apache.mina.core.session.TSession.write</td>
<td>30274 ms (3%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.String.substring(m, n, int)</td>
<td>19522 ms (1%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.StringBuffer.append(java.lang.Object)</td>
<td>10530 ms (2%)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) 5 Clients

<table>
<thead>
<tr>
<th>Hot spot</th>
<th>Inherent time</th>
<th>Average time</th>
<th>Invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.concurrent.ThreadPoolExecutor$Worker.run</td>
<td>108 ms (32%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.String.replace</td>
<td>150 ms (12%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.StringBuilder.concat</td>
<td>70945 ms (5%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.StringBuilder.toString</td>
<td>66121 ms (5%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.apache.mina.core.session.TSession.write</td>
<td>64459 ms (4%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Object.concat</td>
<td>30177 ms (3%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.String.substring(m, n, int)</td>
<td>29153 ms (3%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.StringBuffer.append(java.lang.Object)</td>
<td>35542 ms (2%)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) 10 Clients

Figure 5.15: Profiling of the SimpleServiceStore-backed DA over 10 minutes with 1, 5 and 10 UAs

In comparison. The lookup involves synchronized methods and thus the longer search times leads to longer blocking of threads, thus at a high rate of incoming messages it may happen that a request is not processed quickly enough, forcing the UA to send another message. In very rare occasions that second request also failed, leading to a complete timeout of the service lookup process, yet failure rates remained below 0.015% in all tests.
Figure 5.16: Thread view of a JProfiler session getting data from a remote ApacheDSServiceStore-based jSLP DA.
Figure 5.17: Thread view of a JProfiler session getting data from a remote SimpleServiceStore-based jSLP DA.
5.2 Cirrostratus Backend

This section presents the results of additional tests performed exclusively on the jSLP DA running on the Cirrostratus framework. They deal with replication and the benefits of having multiple DAs. As the CirrostratusServiceStore-based DA was the only one able to leverage replication features inherent to the backend, these tests were restricted to this specific DA implementation.

5.2.1 Registration Propagation

In order to get an idea of how quick registrations made to one node were propagated to another, a series of SLAMD jobs were to be run with varying registration frequencies, number of UAs making registrations and number of replicas to which registrations had to be propagated. However the first run of tests already showed that the replication function of Cirrostratus was still very unstable. Thus the following results are just to be seen as an initial idea of the possible performance of the Cirrostratus based DA.

Figure 5.18 shows how the time between sequentially registering an SLP service with the DA on the bootstrap node and the subsequent retrieval of that service from the DA running on node 2 is affected by adding a third node to the setup. These times include the time it takes for the internal registration of the service with the registering SA, the time it takes for the "Service Registration Message" to reach the DA plus the time it takes the DA to process the incoming registration. Additionally the times listed also include a part of the time it takes the "Service Request Message" to get to the DA from the querying UA and be processed there as well as the subsequent transfer time and processing of the "Service Reply Message", depending on when in the cycle of service lookup the service arrives at node 2 from the bootstrap node. The figure shows that whether adding a third node nor increasing the registration frequency from 1/second to 4/second have a visible impact on propagation time. Increasing the frequency to 10/second however leads to a change in the reported times, as shown in Figure 5.19. The figure shows lookup times averaged over 1s intervals for registration frequencies of 1/250ms and 1/100ms. Increasing the frequency to 10 registrations per second seemingly leads to a linear increase in the time between a service's registration on the bootstrap node and the retrieval from node 2. Increasing the rate even further lead to problems with the replication mechanism, leading to a breakdown in replication where no further updates were applied to the replica. This also prevented tests with multiple SAs registering services, as this breakdown occurred very soon in such scenarios. The tests also show that the update propagation schedule of Cirrostratus seems to push updates about once a second. Increasing the registration rate to above one registration every 250ms seemed to lead to registrations being queued up on the bootstrap node before being propagated to node 2, such that the services registered later were enqueued a long time, first on the bootstrap node and later on node 2, while waiting for the UA to lookup the ones registered before them. The UA was not considered to be a bottleneck at registration rates of well above 1/10ms, as the initial results show that a service request can be handled below 3ms on average for a single UA querying a single DA.
5.2.2 Multiple Directory Agents

This section describes the results obtained from SLAMD jobs sr_predicate (Service Request), ar_tags (Attribute Request) and str_basic (Service Type Requests) as described in Section 4.2.1. The results for one DA were taken from the previous evaluation phase, the results of which are discussed in Section 5.1. In addition, the same jobs were run with 2 and 3 CirrostratusServiceStore-backed DAs present.

Figure 5.20 shows an overview of the resulting lookup times for the three setups. They show a drop in average response time as well as standard deviation with increasing number of DAs in the network, most pronounced in the switch from 1 to 2 DAs.

Table 5.1 shows the distribution of requests between the 2 DAs present. It shows that the UAs choosing the DA to use when unicasting at random from the set of known DAs indeed leads to a fair distribution of requests between the two DAs present. Table 5.2 shows the same information for the setting with 3 DAs running. The distribution of requests between the three DAs is a bit more skewed than in the setting with 2 DAs with one DA receiving more than its expected share of around 33%. However, the DA that receives the highest number of requests is different in each of the three experiments.

Figure 5.21 shows the response times for the three jobs sr_predicate, ar_tags and str_basic averaged over 1s intervals. After approximately 7 minutes the DA on node 3 was shut down, and after about another 7 minutes the DA on node 2 was also shut down. These events are visible as marked spikes in the average response times in all three plots. An increase in the variability of the results between the first (3 DAs) and third interval (1 DA) is visible.

Table 5.3 summarizes the results for the same experiments shown in Figure 5.21. In all three cases the average lookup time increases slightly when DA3 is
shutdown and then markedly when DA2 is also shut down. Variability described as standard deviation remains more or less the same in the first two intervals, it even decreased in the first and third experiments. However there is a distinct increase between the first and third interval in all cases, which as mentioned above is also visible in Figure 5.21.

These results show that adding a second DA to support a possibly overloaded DA increases performance of the overall system as expected. They also show that adding more DAs to an existing set needs not evenly balance load among them, yet it makes a difference nonetheless. The results also show that there may be a time right after a service is registered on a DA and before it is propagated to the other participating DAs, in which that service will not be found by UAs, as it takes a while for the DAs to regain consistency. However, service registrations are considered to happen a lot less frequently than lookups, and registering a service every 250ms over longer periods of time might suggest that a SA is malfunctioning.

Overall the CirrostratusServiceStore-backed DA seems like a decent candidate for a high-performance DA able to process a high load of service lookups in an SLP system. The results suggest that it handles itself fairly well as a solo DA and that adding further DAs to the system as load increases reduces load on each DA by spreading it more or less evenly among the running DAs. The downside is that the Cirrostratus framework is still in an early stage of development and
Figure 5.20: Average response times for jSLP User Agents in the presence of 1, 2 and 3 jSLP Directory Agents running on the Cirrostratus framework (+/- 1 standard deviation)

Table 5.2: Overview of SLP requests by 10 UAs in a network with 3 DAs

<table>
<thead>
<tr>
<th>Job Type</th>
<th>Requests to DA1</th>
<th>Requests to DA2</th>
<th>Requests to DA3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sr_predicate (Service Request)</td>
<td>32.502%</td>
<td>27.498%</td>
<td>40.000%</td>
</tr>
<tr>
<td>ar_tags (Attribute Request)</td>
<td>25.044%</td>
<td>44.992%</td>
<td>29.963%</td>
</tr>
<tr>
<td>str_basic (Service Type Request)</td>
<td>39.984%</td>
<td>29.985%</td>
<td>30.031%</td>
</tr>
</tbody>
</table>

unstable, thus unsuitable for productive use at the moment. The MINA, jSLP and logging bundles had to be hardcoded into the framework to be ignored for resolution, as they had to be installed directly on the OSGi framework rerunning beneath Cirrostratus. The actual replication functionality is currently also hardcoded to include the CirrostratusServiceStore for instrumentation and replication. Yet after much work the framework finally did allow the deployment of three DAs using a replicated ServiceStore to cache services and gain the results discussed above, thus this may be seen both as proof that the jSLP DA runs on OSGi and Cirrostratus as well as a proof of what the Cirrostratus framework is capable of providing, once it is fully implemented and stable.
Figure 5.21: Average response times for jSLP User Agents in the presence of 3 jSLP Directory Agents running on the Cirrostratus. One DA was shutdown after approximately 7 minutes, the second after about 14 minutes.

Table 5.3: Overview of SLP requests by 10 UAs in a network with 3 DAs before and after shutting DA1 and later DA2 down. Data columns show average lookup time and standard deviation (in parentheses).

<table>
<thead>
<tr>
<th>Job Type</th>
<th>DA1</th>
<th>DA2</th>
<th>DA3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sr_predicate (Service Request)</td>
<td>2.51ms (0.95)</td>
<td>2.66ms (0.93)</td>
<td>3.39ms (1.26)</td>
</tr>
<tr>
<td>ar_tags (Attribute Request)</td>
<td>2.36ms (0.79)</td>
<td>2.74ms (0.87)</td>
<td>3.87ms (2.54)</td>
</tr>
<tr>
<td>str_basic (Service Type Request)</td>
<td>2.91ms (0.77)</td>
<td>2.93ms (0.76)</td>
<td>4.48ms (1.58)</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusions

The goal of this project was to develop an SLP directory agent for the jSLP framework which is scalable and performs well. The approach taken was to leverage the existing performance enhancing features of the backends used to store the SLP services registered with the directory agent. The first thing shown by the results of the evaluation is that the directory agent designed for jSLP 2.0 does indeed work and that the SLP user agents of SLP can communicate with both the OpenSLP directory agent and the jSLP directory agent. The addition of the attribute list extension [15] also works well, as it was used during evaluation to reduce the number of messages required to get data stored as attributes of an SLP service in the directory agent.

jSLP 2.0 also works either as a standalone library, as used by the SLAMD clients, or an OSGi bundle, as used by the directory agent bundles running on Cirrostratus nodes. The minimal jSLP build contains the SimpleServiceStore internally and provides full SLP functionality for user agents, service agents and directory agents in any combination either to standalone Java applications or to OSGi bundles as OSGi services. By using the MINA framework and asynchronous requests, clients using the user agent component of jSLP 2.0 can start processing discovered services as they arrive and do not have to wait for the completion of the multicast convergence algorithm. The user agent provided by jSLP 2.0 also provides methods to access results not just as strings but as Java objects, depending on which options are used and which kind of request is launched by the agent. The attribute list extension can also reduce the number of messages required to get all attributes of an SLP service as demonstrated during the evaluation process [15]. Instead of sending a service request to find the actual service followed by an attribute request for that service, a client can send a service request with an attribute list extension prompting the DA to return a service reply including all attributes of that service [15]. This benefit may be lost when the attribute list becomes too long, forcing the DA to send the reply over TCP in several packets, therefore the use of the extension can be toggled on and off in jSLP.

In a first step, the jSLP 2.0 implementation not only extends the existing jSLP functionality but also adds a functioning DA that is compatible with the OpenSLP implementation. The modularity of jSLP allows the deployment of a full SLP implementation either as a standalone library or an OSGi bundle, but it also allows for the use of reduced versions lacking specific parts, such as the DA,
in order to provide only UA/SA SLP functionality for use on devices with memory restrictions. Overall it was expected that the basic jSLP DA would perform worse than OpenSLP due to message parsing, attribute transformations and the difference in overhead between the C and Java implementations. The results show that the jSLP DA indeed consistently performs worse than OpenSLP on average. The difference between the SimpleServiceStore and the ApacheDSServiceStore DAs shows the impact of both the core jSLP machinery including MINA as well as that of the backend on performance. This comparison shows that the main SLP performance bottleneck is located within the ApacheDS. As the jSLP DA adds additional functionality not only in the core SLP processing but also thanks to the backend used, a slightly lower performance can be tolerated.

6.1 ApacheDS as a Backend for jSLP

The second step of the project, providing a scalable high-performance directory agent by leveraging the features provided by the storage backend, was not as successful as hoped for. The first candidate for such a backend, the embedded Apache Directory Server (ADS), not only lacked the required replication functionality but also performed much worse than expected. Clearly in the current state this approach is not suitable for a DA required to handle high SLP discovery load. Using an LDAP server as a backend may still be an option for certain scenarios, such as where the integration of the service cache into an existing LDAP infrastructure is desired. For this purpose the ApacheDSServiceStore implementation provides a way to map SLP services to LDAP entries without the need to know the attributes and their types in advance. This allows the bridging of SLP and LDAP without the need for SLP templates being mapped directly to LDAP schemas.

Once the Apache Directory Server project releases its final 1.5 (or 2.0) version, its use as a backend should be reevaluated, as it is expected to handle twice the number of requests per second than it can now (personal communication) and replication should also be available. If performance becomes comparable to the simple jSLP DA, then the additional replication functionality could make the ApacheDS-backed DA an option for a scalable high-performance SLP directory agent. Using ADS as a backend can also provide additional benefits besides the ability to replicate the DA’s cache of services. It is not a must for the ADS to be embedded and restricted for use by the DA for SLP service storage. It can be deployed as a full LDAP server listening for LDAP queries from clients. Thus the DA can provide LDAP access to its internal store of services, which can be accessed using an LDAP client. By doing this the DA’s ADS can be used for additional purposes e.g., as a user directory. In this way replicating the DA when load increases also replicates those additional features.

6.2 Running jSLP Directory Agents on Cirrostratus

The CirrostratusServiceStore DA running on the Cirrostratus framework can be seen as a proof-of-concept for the use of existing replication functionality
6.3 Final Conclusions

To conclude this thesis, it can be said that jSLP 2.0 provides the basis for the development of a wide range of DAs by leveraging backend-specific features for scalability, access and performance. While no stable and ultimately commercially deployable high-performance directory agent was developed in the given timeframe, the two approaches used provided the groundwork for further development in the form of the basic jSLP 2.0 code, providing SLP functionality and the “extension point” to plug in future DA backends. The two more complex backends, ApacheDSServiceStore and CirrostratusServiceStore, show the flexibility of the jSLP system, both in standalone and OSGi deployments. Furthermore the Cirrostratus deployment demonstrates how backend functionality can be used to extend performance and scalability of jSLP beyond what is available out-of-the-box by the basic jSLP project.

6.4 Contributions and Future Work

This section concludes this thesis by providing a list of contributions made by the author. It also offers some ideas for future work in this field. The main contribution of this project is the further development of jSLP 1.0 to version 2.0. This includes a row of changes, including the migration of jSLP
Conclusions

70

to the MINA framework for communication and the addition of asynchronous service discovery. jSLP 2.0 also adds the SLP directory agent component to the existing user agent and service agent implementations in jSLP 1.0.

The modular DA developed for jSLP 2.0 is designed to leverage the capabilities of the backend ultimately used to store SLP service registrations, making it scalable and adaptable to various systems. This project provided two examples of possible ServiceStore implementations for standalone and OSGi deployment as well as some baseline performance analysis of these. While the ApacheDS-based DA did not work in the way intended, it nevertheless led to preparatory code for possible future deployment, including the dynamic mapping of SLP services to LDAP entries without the need for SLP templates.

In order to use MINA as a communication framework it had to support multicasting, which it did not at the time this project was started. Therefore, the addition of the datagram acceptor and connector based on the Apache Portable Runtime to the MINA code can also be considered a contribution.

During the process of developing the various components used in the evaluation process a series of bugs were discovered and fixed in the Tomcat Native and Apache Directory Server projects. The deployment and evaluation of the CirrostratusServiceStore also led to fixes in the Cirrostratus project and provided testing of the framework.

Future work based on this project could include the development of different ServiceStore implementations and their evaluation against the existing ones. These implementations could be based on a range of backends and include further functionality, such as the bridging of SLP systems across the internet or the integration of SLP into other service discovery systems, as well as increase performance and scalability. They could also include their own additional features such as the implementation of an internal replication mechanism. Once such a backend reaches a stable and easily deployable state, it should be extensively evaluated in terms of scalability and performance, not just with 10 clients, but in a realistic environment with dozens of UAs and a number of SAs using the system concurrently.

There are also still a range of issues to be dealt with for jSLP, the main ones being the correct use of the language tags (localization) and the implementation of the basic SLP security mechanisms. Furthermore, there are also still a range of extensions to the SLP protocol that can be added besides the attribute list extension implemented in the course of this project.
Bibliography


