NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG ELEKTROTEKNIKK

MASTEROPPGÅVE

Kandidaten sitt namn: Glenn Hisdal

Fag: TDT4900 - Datateknikk og informasjonsvitenskap

Oppgåva sin tittel (norsk):

Oppgåva sin tittel (engelsk): Service Discovery Techniques for Distributed Systems Using SmartFrog

Oppgåvetekst:

SmartFrog is a recent framework developed at HP Labs Bristol that enables easy configuration, deployment and management of distributed software systems. At times distributed software components may have to communicate with other components. In order to do this, a way to discover the location of such components is needed.

This Masters thesis should extend the SmartFrog environment to use a service discovery protocol in order to provide this feature. One such protocol is the Service Location Protocol (SLP) standardised by IETF.

Key features of this work should hence include:

- An SLP implementation according to the given standard.
- SmartFrog components that can use SLP for discovering other components or services.

The work will be performed at CERN and may include other tasks related to the SmartFrog framework.

Oppgåva gjeve:	14. januar 2004
Svar leveres innen:	27. juli 2004
Svar levert:	27. juli 2004
Utført ved:	Det Europeiske Kjernefysikklaboratoriet CERN
Fagleg rettleiar:	Anne C. Elster (IDI), Sverre Jarp (CERN)

Trondheim, 27. juli 2004

Anne C. Elster
Faglærar

Abstract

When a large number of computers are to co-operate as a distributed computer system, it is essential that each node in the system is configured correctly. To simplify this configuration a tool that allows one to configure the entire system in an easy way is needed.

SmartFrog is a recent system developed at HP Labs in Bristol, for describing, activating and managing distributed applications. With SmartFrog one can describe a system as a set of software components running on the available nodes. One description file can hold the configuration for the entire system. All components in the system can then be started with a single command. SmartFrog can also handle automatic reconfiguration of a running system, for example by inserting a backup node if one node fails.

The Service Location Protocol is the IETF standard for how services can be advertised and located over a network. This thesis demonstrates that by using this protocol in combination with SmartFrog, one can automatically locate services needed by the SmartFrog components through the SLP protocol as they become available or disappears. SLP removes the need of having to know which host the service one wants to use runs on. The required information to communicate with the service is obtained by searching for the service using SLP. This is very useful in environments where the service can be moved to a different host in time. By using SLP to advertise the service, it can be found even if it is moved to a new host.

Our Java implementation of the SLP includes all the mandatory parts of the SLP standard as well as a set of SmartFrog components which allow the discovery and advertising features of the protocol to be used within SmartFrog. A number of tests were performed in order to test the correctness of the library. The tests included checking that the library implemented for this thesis is able to communicate correctly with other implementations of the Service Location Protocol.

In addition to the SLP library, this thesis shows how a set of components for controlling the Portable Batch System (PBS) were made using the features of our SmartFrog SLP implementation. These components made use of the SLP library to have a pool of execution nodes that could be dynamically added to the PBS system. The nodes were advertised through SLP so that a component could search for available execution nodes to add to the system.

Acknowledgements

First of all I would like to thank my Supervisor at NTNU, Anne C. Elster for picking me as candidate for the CERN Openlab student programme, summer 2003. This was the first step towards doing my thesis at CERN. I also thank her for her advice while writing this thesis.

I also thank CERN for accepting me as a Technical Student this year, which allowed me to do my thesis there. In particular, I thank Sverre Jarp for being my supervisor at CERN and getting me interested in the SmartFrog framework. I also thank Andreas Unterkircher for his interest in my work and his help and ideas with the SmartFrog components used to control the PBS system. Everybody else at CERN Openlab also deserves a big thank you. It was great working with you all.

This thesis would not have been what it is without all the help from the people at Hewlett-Packard Laboratories. The help and suggestions while working on this are very much appreciated. In particular, I thank Patrick Goldsack, Peter Toft, Julio Guijarro and Guillaume Mécheneau. I also thank them for allowing me to put my code for the SLP library in the SmartFrog CVS repository.

Finally, I would like to thank all my friends and family for their support.

Contents

List of Figures

Chapter 1

Introduction

This thesis marks the end of my master studies at the Norwegian University of Science and Technology, NTNU. It describes my work on a Java library that allows components within the SmartFrog framework to find services in a network using the Service Location Protocol. This chapter will give the motivation for such a library, and present an outline of the rest of this thesis.

1.1 Motivation

In recent years, distributed computing has become increasingly common. A network of ”off the shelf” computers provide very high computing power for a fraction of the cost of a traditional super computer. Tasks are split into smaller parts that can be executed on each computer, and the results are merged into the final answer. This is particularly well suited for CPU intensive tasks that can be split into independent parts requiring little or no communication during the computation. Until now, cluster computing has been a common way to create such systems. A cluster is a number of computers, usually placed in the same room, connected over a local area network. Recently the focus has been on connecting clusters and individual computers over the Internet to create a computational Grid. By combining the computing power found at several sites in many parts of the world, a very powerful system will be available to run large, CPU hungry jobs. A good resource for information on Grid technology is the CERN GridCafe Web site[11].

For such a distributed system to work properly, a number of software components has to be installed and running. It is essential that every part of the system is configured correctly. If one has a cluster of one hundred machines, it is generally not a good idea to set up every machine by hand. A configuration tool is needed to do this. A lot of tools are available for doing the static configuration of the system, getting all the required software installed. Usually the configuration software focus on setting up one type of machine in a specific way and do not know anything about the system as a whole.

SmartFrog is a framework for describing, activating and managing distributed applications. It is developed at HP Labs in Bristol, UK and is now released under the GNU Lesser General Public License. A SmartFrog application is a collection of components, Java objects, that run on a number of hosts. The life cycle (instantiation, initialisation, startup, termination) of the components in an application is controlled by SmartFrog through a set of interfaces. There are different components included with SmartFrog that provides different life cycles. The Compound component, which will be used in this thesis, provides an all-or-nothing life cycle.

What separates SmartFrog from other configuration systems, is that SmartFrog does not focus on getting one specific node configured in a certain way. SmartFrog is used to configure a system of distributed components running on any number of hosts. It requires that some basic setup of the nodes is already in place. At least Java and SmartFrog have to be installed on each node. For SmartFrog to be able to use a node, the SmartFrog daemon must be running on that node. The entire configuration for a SmartFrog application can be stored in one place. There is no need for separate files depending on which node the software is to be installed on. Once a description of the system is written, the SmartFrog application can be started with one single command. This will bring up the required components on the hosts specified in the configuration. SmartFrog makes it easy to do dynamic reconfiguration of a running system. Included in the SmartFrog distribution, is an example of a dynamic web server. A number of nodes in a cluster is configured to serve web pages. When the load of the system goes over a given limit, new nodes are brought in to decrease the load. When the load drops below a certain limit, one server is shut down. The dynamic web server example is described in ”A Brief Description of the Dynamic Web Server Demonstrator”[6].

The Service Location Protocol[9] (SLP) is standardised by the Internet Engineering Task Force, IETF. It specifies how services can be advertised and located on a network. A service is advertised using a ServiceAgent component, and is identified by a Service URL. Each service can also be assigned a number of attributes. Locating services is done using a User Agent. The User Agent will send requests for the given service type over the network. An LDAPv3 search predicate[17] can be used to say which attributes must be present for the requested service.

The use of a service location protocol becomes interesting in dynamic environments where services may be added, removed or moved around. By using SLP, clients can discover that a service has been added or removed and reconfigure themselves. Instead of hard coding the location of the needed services, a client is configured with the service type it needs. This allows the client to connect to any server providing the needed services. One specific service can be provided by many servers, leaving the client with a choice of which server to connect to. If the server goes down, another one can be selected. Todd Poynor - ”Automating Infrastructure Composition for Internet Services”[19] describes how SLP can be used to dynamically configure web services.

SmartFrog in combination with SLP offers a very good solution for deploying distributed applications in a dynamic environment. A SmartFrog component can represent a service or client in the system. The service is advertised using SLP. Clients use

an SLP locator to discover the service they need. The SmartFrog application can then be made to automatically reconfigure itself as services are removed or added.

The GridWeaver project used SmartFrog in combination with LCFG[1] to run a grid enabled print service. Printers were advertised and located using SLP. A description of the work can be found in "SmartFrog meets LCFG Autonomous Reconfiguration with Central Policy Control"[2]. More information on the project is also available on the GridWeaver homepage[12].

1.2 Outline of Thesis

The outline of the rest of this thesis is as follows: Chapter 2 will give some background information on SmartFrog and the Service Location Protocol. It will also present Rendezvous, which implements an alternative service discovery method. The last section of the chapter will give some information on mSLP. mSLP is an open source implementation of the Service Location Protocol and has previously been used to provide SLP support for SmartFrog.

Starting in Chapter 3, the work done on this thesis is presented. Chapter 3 will show how the SLP library was implemented. It gives an overview of the classes and interfaces developed, and shows how everything is put together to create a working implementation of the different parts of the Service Location Protocol. Chapter 4 will show how the SLP implementation was tested in order to verify that it worked correctly. A set of components for controlling the Portable Batch System (PBS) was also developed during the work on this thesis. These are described in Chapter 5. The PBS system used SLP to advertise available execution nodes. This allowed nodes to be dynamically added to the PBS system.

Chapter 6 gives a summary of the results of this thesis. Finally, Chapter 7 concludes the work, giving some ideas on future improvements that can be done on the SLP library and PBS components.

Chapter 2

Background and Related Work

This chapter will give an introduction to SmartFrog and the Service Location Protocol. It will also give an overview of Rendezvous, which provides an alternative service discovery method to that found in SLP. The last section will look at how a previous implementation of SLP for SmartFrog was done. That implementation was based on mSLP which is an open source implementation of the Service Location Protocol. mSLP is, however, not a fully standard implementation of the protocol.

2.1 SmartFrog

This section will give an introduction to SmartFrog. Not every aspect of the system will be covered here. The section includes the things that are important to understand the work described in this thesis. It starts with a quick overview and then look at the different parts of the SmartFrog system. More detailed information on SmartFrog can be found in the SmartFrog Reference Manual[7].

2.1.1 Introduction

The SmartFrog system has been in development over several years at HP Labs Bristol, UK. SmartFrog is a framework for describing, activating and managing distributed applications. The framework is as of 13 February 2004 an Open Source project hosted on Sourceforge[16]. The software is released under the GNU Lesser General Public License[21]. A copy of this license can be found under ”Licenses” on the CD-ROM.

A SmartFrog application is a set of components, Java objects, that run on a number of hosts. The components within an application is contained in a component hierarchy, with one component being the root of the hierarchy. Components can locate other components by following the links through this hierarchy. The life cycle of an application is controlled by the life cycle of its components. More on this in Section 2.1.3. A SmartFrog system may have several SmartFrog applications running. Each application is independent of the other applications. There are no direct links between components belonging to different applications.

The SmartFrog Reference Manual[7] splits the SmartFrog system into three parts:

- **SmartFrog configuration description environment**.
  This part is the description notation and tools to enable the storage, validation and manipulation of these descriptions.

- **SmartFrog Component model**.
  The component model defines the interfaces to be implemented by components. The purpose of the interfaces is to support the various life cycle operations (creation, initialisation, termination) and management actions like getting status information.

- **SmartFrog configuration management system**.
  The management system uses the component descriptions to start components and manage them throughout their entire life cycle in a secure way.

This thesis will give an introduction to the SmartFrog description language, the component model and some of the tools used to manage a SmartFrog system. Section 2.1.5 will look more at how SmartFrog deals with references within an application description. This is important for understanding how the service location protocol is being integrated into SmartFrog.

To allow a host to run SmartFrog components, it must have the SmartFrog daemon running. A SmartFrog description is started on one host. If a particular component is to be deployed on another host, the daemon will contact the daemon running on that host and tell it to start the relevant parts of the description. Creating and running a SmartFrog application typically involves the following steps:

1. Write SmartFrog components in the Java language.

2. Write a SmartFrog description of the system in the SmartFrog description language.

3. Start the application using the provided tools.

The next section will give an introduction to the SmartFrog description language, and show how a simple SmartFrog application can be described.

2.1.2 SmartFrog Description Language

The SmartFrog description language is the language used for describing the configuration of a distributed system in SmartFrog. The link between the language and the rest of the system is the parser. The description file is on startup run through a parser that converts the information into internal structures representing the application. If a parser for another language is written, one could use that language for writing the descriptions. Currently only the SmartFrog description language is supported.

A SmartFrog description is created by a collection of key-value pairs, called attributes. The value of an attribute can be a simple value (String, Integer, Boolean, ...) or a SmartFrog component description. A component description is a collection of attributes describing a component. This way components can be nested and one gets a component hierarchy. A special attribute, sfConfig, gives the start of an application. This can be seen a something similar to the main function found in C and Java. sfConfig becomes the root of the component hierarchy.

The features of the SmartFrog description language includes:

- **Prototypes**
  The attributes given in a description do not have any type. Any attribute that is a component description (collection of attributes) can be used as a prototype for another. This provides a form of inheritance within the language. Another attribute can extend a prototype and modify this to create the value for itself. The modification can be changing the values for some attributes defined in the prototype, or add new attributes.

- **References**
  Instead of providing a value to an attribute directly, the value can be a reference to another attribute. The value given for that attribute is then copied when the

description is processed by the system. In some cases one may want the reference to be the value. For example when the referenced attribute is not available before the component has been initialised. The LAZY keyword is used to delay the resolution of the reference. In that case the link becomes the value of the attribute.

- **Comments**
  The SmartFrog description language supports single and multi line comments. The syntax for comments are the same as one will find in the programming languages C++ and Java.

- **Include files**
  A description may #include other description files. These files are not copied into the text as with C include files, but are parsed as a separate stream. This means that the included files must be a syntactically correct collection of attributes. Normally one will use this feature to include previously defined component descriptions into an application.

- **Functions and Operators**
  A number of functions and operators are defined for use within the descriptions. These includes string concatenation, summation, appending elements to a vector, and more. It is also possible to use if-then-else statements when defining attributes.

Detailed information on the SmartFrog description language is given in the SmartFrog Reference Manual[7]. A simple description showing some of the features of the language is given in Figure 2.1.

```
/* Simple SmartFrog description.
   Shows the use of prototyping, references, include */
#include "mycomponent.sf"  // include description
#include "org/smartfrog/components.sf" // std smartfrog components.

theText "AmigaOS4"; // aString attribute

sfConfig extends Compound {  // sfConfig is the starting point for SF App.
      c1 extends MyComponent { // the attribute c1 extends the MyComponent prototype
            name theText;  // simple reference (=> value copied)
      }

      c2 extends c1 {
            c1Comp LAZY c1; // LAZY reference (=> Link is the value)
      }
}
```

Figure 2.1: Simple SmartFrog Description

2.1.3 SmartFrog Component Model

The SmartFrog component model defines how components are created and how they interact. When a SmartFrog description is given to the system, this description is parsed and a collection of components created based on the attributes found in the description. SmartFrog has a number of interfaces that defines the behaviour of different types of components within the SmartFrog framework. Components implement these in order to receive notification of life cycle changes and other events in the framework.

The main interface is the interface Prim. A typical SmartFrog component implements the life cycle methods found in this interface. One will also in most cases make the component a subclass of the PrimImpl component. PrimImpl has default implementations for the various life cycle stages. Three methods are important for the life cycle of the component. These are:

- **void sfDeploy()**
  The sfDeploy method is called when the component goes from the instantiated state to the initialised state. Component specific initialisation code is added here. For proper functionality within the SmartFrog framework the super class' sfDeploy method should be called within this method.

- **void sfStart()**
  sfStart is called when the component goes from the initialised state to the running state. Again, any component specific initialisation/startup code is added here. As with sfDeploy, it is important to call the super class' method.

- **void sfTerminate(TerminationRecord)**
  Called when the component goes to the terminated state. Component specific termination code is added here, and the super class' method called.

For sfDeploy and sfStart the super class' implementation is normally called first in the method. For sfTerminate the call to the super class comes at the end.

The life cycle of a component is implemented as a simple state machine, as seen in Figure 2.2.

The Prim interface also defines other methods that can be used to override the default behaviour of the component. Interesting for this thesis is the sfResolve method. More information on this method will be given in Section 2.1.5. This method is called when resolving a reference to an attribute within the component. As shall be seen later, this is used by the SLP locator component to trigger the service discovery.

Components within an application are bound to other components through parent-child links. This is also true for the life cycle of the components. Some components can take a collection of other components as their attributes. One such component is the Compound. The compound provides an all or nothing life cycle. On startup it creates all its children. Whenever one child terminates, all other children are terminated and the compound itself terminates.

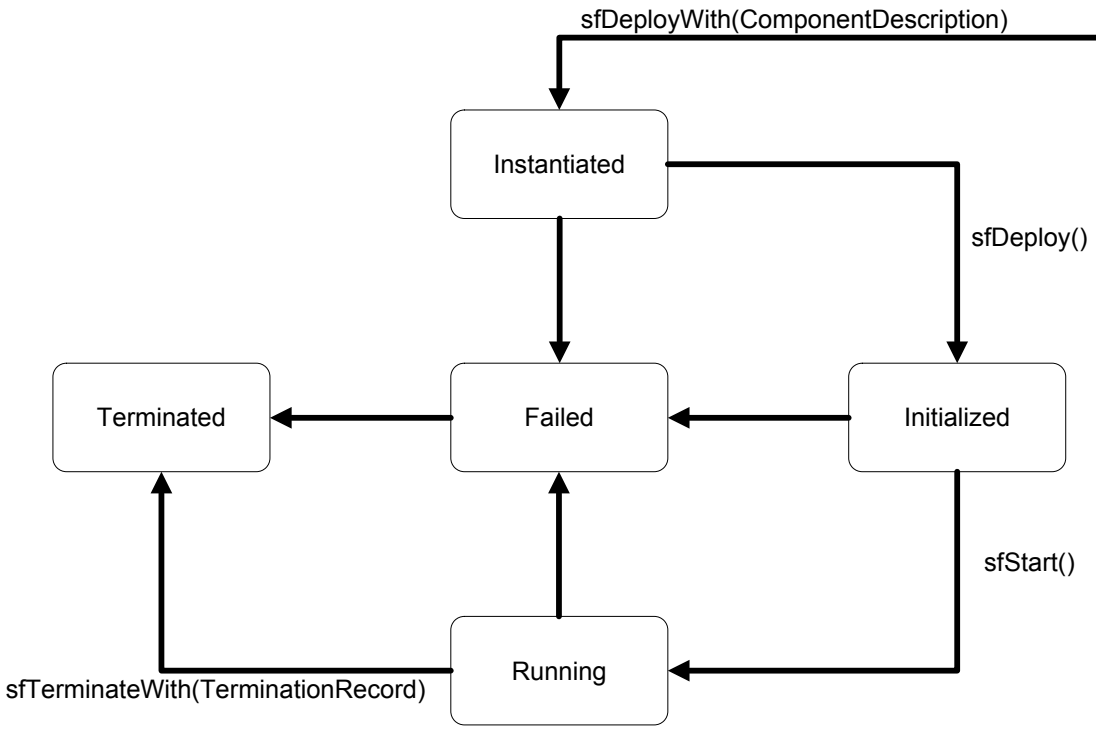

Figure 2.2: Life cycle of a Component
Diagram heavily inspired by the identical diagram found in the SmartFrog Reference Manual[7], Section 6.4

The examples found in this thesis will only use Prim and Compound components. Components providing a different life cycle from that given by the Compound are also included with SmartFrog. Information on these can be found in the SmartFrog Reference Manual[7].

2.1.4 Management Tools

The standard SmartFrog distribution contains a number of tools used to control a SmartFrog system. This section will present the tools used to start the SmartFrog daemon, start and stop applications, and to manage running applications.

- **sfStartDaemon**
  The sfStartDaemon command is used to start the SmartFrog daemon on the local machine. Once the daemon is running, SmartFrog components can be started from a remote machine if required. The SmartFrog daemon must be running for a machine to be a part of a SmartFrog system.

- **sfStart**
  sfStart is used to start a SmartFrog application. The description may tell Smart-

Frog to start certain components on a remote computer, in which case the daemon running on that machine is given the parts to start. The syntax for sfStart is:

```
sfStart <hostname> <app> <description>
```

<Hostname> is the host on which the description is to be started. This does not need to be the local computer. Any computer with a running SmartFrog daemon can be told to start the application. <App> is the name of the SmartFrog application. Each application must be given a unique name. The application will be known by this name within the SmartFrog framework. <Description> is the description file that describes the application to start. This must be written in the SmartFrog description language, and contain the ”sfConfig” attribute which will be the root of the application’s component hierarchy.

- **sfTerminate**.
  Terminates a running application. The syntax is:

  ```
  sfTerminate <hostname> <app>
  ```

  Where <hostname> is the name of the host where the application was started, and <app> is the name of the application. In other words, these are the same as for the corresponding sfStart.

- **sfManagementConsole**
  Starts the SmartFrog management console. The syntax is:

  ```
  sfManagementConsole -h <hostname>
  ```

  where <hostname> is the name of the host to manage. The management console can be used to get information about the running applications. It shows the components in each application, and can also be used to terminate a component, or detach it from its parent. A detached component becomes a root component. That is, it has no parent and is no longer part of the controlled life cycle of the application it belonged to.

2.1.5 SmartFrog References

This section will give some more information on references, or links, used in the smartfrog framework. As seen in Section 2.1.2, it is possible to have links from one attribute to another within the description. A normal link. That is, a link of the form

```
attribute otherAttribute
```

is resolved when the description is parsed. The value given by otherAttribute is simply copied, so if we have the following example

```
attribute1 4;
attribute2 attribute1;
```

this will become

```
attribute1 4;
attribute2 4;
```

A special type of link is the LAZY link. The LAZY keyword tells the system that the reference is to be resolved on runtime, thus the link becomes the value of the attribute. There may be a number of reasons for using a LAZY link instead of normal link. A typical example is that the attribute is not available before the component has been initialised. For example when we want a reference to a running component. Given the following example:

```
comp1 extends SomeComponent;
comp2 extends SomeOtherComponent {
    c LAZY comp1;
}
```

The attribute ”c” of ”comp2” is to be a reference to the running ”comp1”. Using the LAZY keyword enables this. If the LAZY keyword is omitted, the value of ”comp1” would be copied down to the attribute ”c”, giving this situation:

```
comp1 extends SomeComponent;
comp2 extends SomeOtherComponent {
    c extends SomeComponent;
}
```

Obviously this does not give the desired result.

Within a SmartFrog application, links to other attributes can easily be created. The parent-child relationships can be followed to resolve these links. If a LAZY link is created as above, referencing another component, one will get the Java object for that component back from a call to ”sfResolve”. sfResolve is used within the Java implementation to get the value of an attribute. There will be more on this method later when looking at how references are resolved. In some cases one may want to bind to a component belonging to a different application. Since two applications are completely independent, one needs to know a bit about the other application to be able to do this.

Coupled Binding

Coupled binding is what we find within a single SmartFrog application. The components all have their place in the hierarchy of components belonging to that application. By following parent-child links, references to attributes in the application can be resolved. The default reference type is the ATTRIB reference. This looks for the given attribute somewhere in the path from the root and down to the component in question. The attribute closest to the component is chosen. So we may have something like:

```
sfConfig extends Compound {
    comp1 extends Prim;
    comp2 extends Prim {
        attr LAZY comp1;
    }
}
```

De-coupled Binding

When we want to use components from another SmartFrog application, things get a bit harder. There is a special reference type; HOST, which can be used to allow this. In order to use this, we must know which host the other application was started on, the the name of the application and the location of the component in which we are interested in the component hierarchy of the other application. An example is:

```
sfConfig extends Compound {
    comp2 extends Prim {
        attr LAZY HOST <hostname>:<appname>:comp1;
    }
}
```

This is the same application as given in the previous section, except that the comp1 attribute now belongs to another application. The HOST reference is used to locate the attribute. <Hostname> is the name of the host where the interesting application was started. <Appname> is the name of the interesting application, and ”comp1” is the interesting attribute within that application. The <hostname> and <appname> is the same parameters as would be given to sfStart when that application was started. If someone decides to restart the referenced application on another host or under a different name, the HOST reference must be updated accordingly.

SLP provides an alternative method of doing de-coupled binding. With SLP one does not need to know the details about the other applications, nor where it runs. One needs to know only one thing: The service type used to advertise the interesting component. The use of SLP within SmartFrog will be further explained in Chapter 3 and the SLP manual found in Appendix E.

Resolving References

When a SmartFrog component needs to get the value of an attribute, it does this by creating a reference to the attribute and call the sfResolve method to resolve this reference. If the attribute is a normal value, that value is returned. When the attribute is a reference, this reference is again resolved in order to get the value of the referenced attribute. The usual way of getting the value of an attribute is:

```
Reference ref = new Reference("myAttribute");
Object attrValue = sfResolve(ref);
```

There are also many other variants of the sfResolve method. These can be found in the Javadoc files for the SmartFrog framework. Of interest in this thesis, is the method

```
Object sfResolve(Reference ref, int index)
```

This method can be overridden by components in order to create the value of the attribute on runtime. This is used in the SLP locator component to find when we want to resolve the ”result” attribute. The Reference is a SmartFrog Reference consisting of many reference parts. The index parameter gives the reference part we are interested in. Reading that reference part, it is possible to check if it is for the attribute one is interested in. Chapter 3 will show how this is used in the SLP locator component.

2.2 Service Location Protocol

This section will give an introduction to the Service Location Protocol, SLP. A more detailed description can be found in the SLP specification, RFC-2608[9].

2.2.1 Introduction

The Service Location Protocol defines a way of advertising and discovering services over a network. It works by sending a number of messages either using multicast to all nodes or unicast to one specific node, depending on the configuration of the system.

An implementation of the Service Location Protocol is made up of three main parts: User Agent, Service Agent and Directory Agent. Each of these will be explained in the following sections. The parts on the agents will mostly focus on service requests, which is the main idea of the SLP. Other requests, Service type request and attribute request, are also part of the protocol. The main difference between the types of requests is that different messages are sent depending on which request is being issued.

Each service is represented by a service URL and a set of attributes. The URL specifies the type of service and the location of the service. Each service has a unique service URL. The attributes are used to define certain properties of the services. When searching for a service, one can specify that some attributes must be available for the requested service. This limits the number of replies if there are multiple services of the correct type. SLP use something called scopes to group services together. Each SLP agent is configured with a set of scopes, and can only operate on services found in at least one of the scopes it is configured with.

2.2.2 Service URLs and Attributes

The Service URL is what identifies a service in SLP. Each service has a unique URL that gives the location of the service. The URL is made up of a service type and the location of the service. A service type may be a special service: type of the form ”service:abstract_type:concrete_type” or just ”type”. The latter gives a normal URL, like ”http://hostname/path”, where http is the type of service. The concrete_type part of the service: type is optional. It is useful for example when a type of service can be accessed in multiple ways. E.g.

```
service:printer:lpr
service:printer:ipp
service:printer:smb
```

These are all printer services, but different protocols are used to access the printers. In order to find all printers one would do a search for the service ”service:printer”. If just one type of printer is wanted, the entire service type is used for the search.

Attributes is a way of assigning a set of properties to a service. An attribute is given with an id and a set of values, e.g.

```
x=5,6,7
```

gives the attribute named x with the values 5, 6 and 7. Each service can be assigned any number of attributes. It may also have no attributes defined. When searching for a service, one can specify that certain attributes must be present in the service one is looking for by using an LDAPv3 search predicate[17].

2.2.3 Scopes

An SLP scope is a character string giving the name of a scope. Scopes can be used to group services. Since an SLP agent can only see services within the scopes it is configured to use, scopes can be used to control access to services. The default scope in SLP is ”default”. The Directory Agent and Service Agent must always be configured with at least one scope. The User Agent may be configured with an empty scope list, in which case it is able to discover services in any scope.

2.2.4 User Agent

The User Agent, or Locator, works on behalf of one or more clients needing to locate a certain service. It will send a message requesting the service to Service- and/or Directory Agents in the network. The UA waits for the other agents to reply to the request, and then pass the results on to the client. A service discovery may result in multiple services being found. The client must know how to select the service it wants. The client must also be prepared for the fact that there may be no service of the given type present.

When the User Agent needs to look for a service, it will first check if it knows any DAs that supports the requested scopes. If this is the case, the request is sent directly to that DA using unicast. If no known DA supports the given scope, the UA will send the request using multicast. The request will then be received by all Service and Directory Agents reachable by multicast. If any of these are configured with the correct scope, and has the service in question registered, they will reply with the Service URL identifying the service. It may also be the case that the Directory Agents known by the DA supports some of the scopes in the request but not all. When this happens, the UA will send a unicast message to the DA as well as a multicast message in order to discover all services. This is illustrated in Figure 2.3.

2.2.5 Service Agent

The Service Agent works on behalf of one or more services. It advertises these services so that user agents are able to find their location. When the SA gets a request from an UA, it checks that it is advertising a service of the requested type. If it is, The location of that service is returned to the client. The Service Agent may have several services

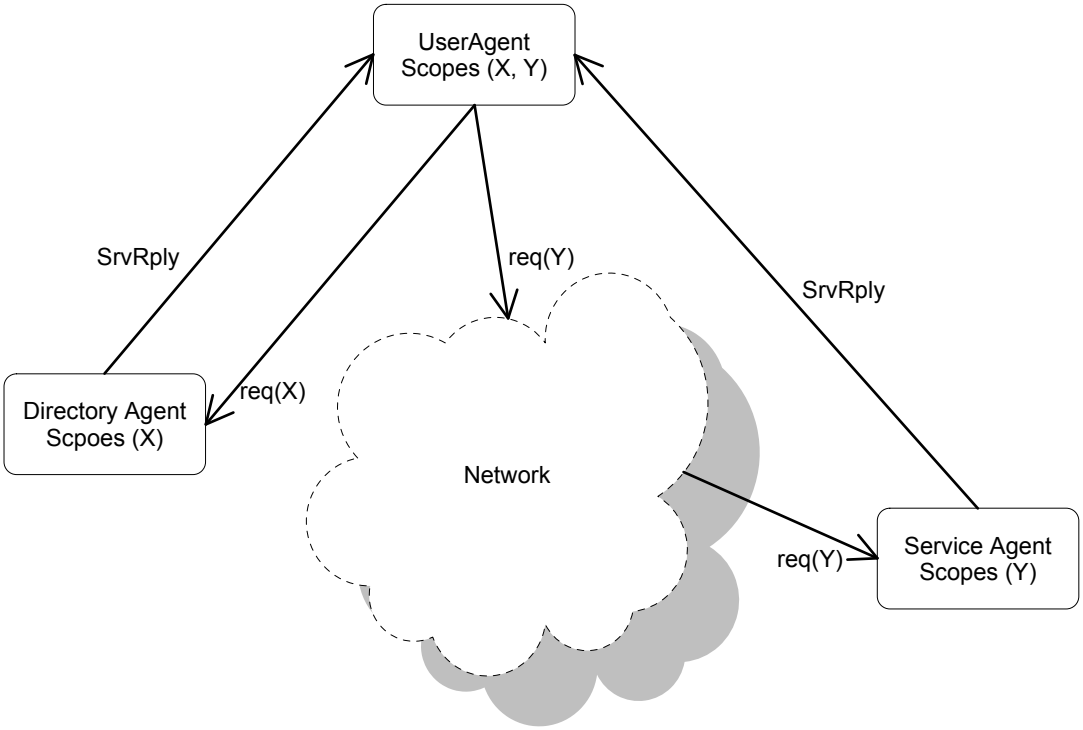

Figure 2.3: Sending SLP Requests
Messages in scope X, go directly to the DA. Messages in scope Y goes to the network using multicast, and are eventually received by a SA. All replies are unicast to the UA.

that match the given type registered. All matching services will be returned in the reply.

At times, the Service Agent will receive a request for a service it does not know. If this request was received on the multicast address, the request is simply ignored and no reply generated. If, on the other hand, the request was received on the unicast address, a reply with no URLs will be returned. Similarly for errors. If an error is encountered while handling the request, no reply is sent if the request was received on the multicast address. If received on the unicast address, a reply indicating the cause for the failure will be returned. In general, requests directed to Service Agents should be sent to the multicast address.

The Service Agent is responsible for registering all its services with every Directory Agent that exists in the network, as long as scoping rules allow it. When a service agent quits, or it receives a de-registration for a service, it must also deregister the service from the DA. This is to decrease the chance of getting a reply for a non-existent service.

2.2.6 Directory Agent

The Directory Agent works as a cache for service advertisements. It receives service registrations from each SA in the network, and is then able to advertise those services. Adding one or more DAs in the network will reduce network traffic, improving the scalability of the system. This because User Agents will then send their requests to the DA instead of multicasting the request to the entire network.

The Directory Agent operates by the same rules as the Service Agent when it comes to when a reply is returned. A User Agent can expect the same reply from a DA as it would get by contacting a SA. The User Agent does not need to do anything special when receiving a reply from a Directory Agent. In fact, it does not even need to know that the reply was received from a Directory Agent and not a Service Agent.

When the DA starts up, it will send out a DA Advert message. This is to notify other agents about its existence so that they know it is there. This message will also be resent periodically.

2.2.7 SLP Messages

The Service Location Protocol works by sending and receiving special SLP messages. This section will give an overview of each of the message types defined in the SLP. The messages SrvReq, SrvRply, DAAdv, SAAdv, SrvReg and SrvAck are required to be supported in any implementation of the service location protocol. All other messages are optional. Below is a list of all the messages supported by SLP, and a brief overview of what they do. When a message containing a request for a service, service type or attribute is sent, a reply to that message is expected. For one request there is always one special reply message that is returned, with the exception of the service request message which may get one of the messages service reply, directory agent advert or service agent advert in return.

- **Service Request (SrvReq).** The service request message is the message sent by User Agents when they need to discover a service. The message contains the type of service the UA is looking for, the scopes to look in and optionally a search filter used to check for certain attributes. Any SA or DA who advertise the requested service will reply with a service reply message. If the request contains the special service types ”service:directory-agent” or ”service:service-agent”, a directory agent advertisement or a service agent advertisement is returned instead of the normal service reply. The ”service:directory-agent” request is used when a User Agent or Service Agent is looking for a Directory Agent. The ”service:service-agent” can be used to get information about running Service Agents.

- **Service Reply (SrvRply).** The service reply message is sent as a reply to service requests. It contains a list of service URLs for services that matches the request. If an error occurred while handling the request, the reply message will include an

error code indicating the kind of error that happened. If the number of matching services is so large that they can not all fit inside the message, a flag is set to let the client know that the message was truncated. The client can then decide what to do. It may resend the request by TCP or use the Service URLs included in the reply. It could also modify the request trying to limit the number of possible services.

- **Directory Agent Advertisement (DAAdvert).** The DAAdvert message is sent periodically by the DAs to let other agents know about their presence. It contains the location of the DA as well as the scopes the DA is configured with. The DA may also be configured with a set of attributes, in which case those are included in the message. The DAAdvert is also sent as a reply to service requests when the request is for the special service type ”service:directory-agent”.

- **Service Agent Advertisement (SAAdvert).** The SAAdvert is used to advertise a service agent. The message is sent as a reply to a service request when the request is for the special service type ”service:service-agent”. The message contains the location of the service agent, the scopes supported by the agent and any attributes the agent is configured with.

- **Service Registration (SrvReg).** The service registration message is sent by a service agent when it needs to register a service with a DA. The message includes the information required for the directory agent to advertise the service. When the directory agent has received the registration it will reply with a SrvAck message saying whether or not the registration was successful.

- **Service Acknowledgement (SrvAck).** The SrvAck is sent by the DA when a service registration or deregistration has been received. The message includes only an error code. If no error occurred, the code is set to zero. In case of an error, the error code is set to indicate which type of error happened. Look in the SLP specification[9] for a list of possible error codes.

- **Service Type Request (SrvTypeReq)** The SrvTypeReq is sent when one wants to find which types of services are available. The reply to this request is a service type reply with all available services. A service type may have a naming authority set, typically this is the name of the organisation which came up with the type. A service request can be limited to only look for service types by a given naming authority.

- **Service Type Reply (SrvTypeRply)** The SrvTypeRply is sent as a reply to a Service type request. It contains a list of all service types known by the Service Agent or Directory Agent replying to the request. If a naming authority is given in the request, only service types given by that naming authority are returned.

- **Attribute Request (AttrReq)** The attribute request is used to get all the attributes, with their values, of a service. One can search for the attributes of one specific service by providing the service URL for that service. It is also possible to get all attributes for a given service type. One then specifies a service type instead of a full URL. All attributes of all services of the given type is returned in an Attribute reply message. If only some particular attributes are of interest, one can provide a list of the interesting attributes in the request. Only attributes with their names listed will then be included in a reply.

- **Attribute Reply (AttrRply)** The attribute reply is sent as a reply to an attribute request. The reply contains a list of all attributes for the given service or service type.

All SLP messages contains an SLP message header. This header holds information on which type of message we have, the version of the SLP protocol used, and an ID for the message. Each SLP message has a unique ID that is randomly chosen. The full details on the header and various SLP messages can be found in Chapter 8 and 10 of RFC-2608[9].

2.3 Rendezvous

This section will give a presentation of a new technology developed by Apple Computers[10] called Rendezvous[15]. One of the features provided by Rendezvous is service discovery. Rendezvous is made by putting three technologies together: Zeroconf[3], multicast DNS[5] and DNS based service discovery[4]. The latter, DNS based service discovery is of most interest for this thesis as it provides an alternative to the service location protocol, SLP.

Each part of Rendezvous will be presented in the following sections. Section 2.3.4 will look at how Rendezvous, or more specifically, DNS based service discovery differs from the service location protocol.

2.3.1 Zeroconf

Zeroconf[3] allows computer to automatically configure themselves with a valid link-local IP-address. This is useful in cases where the normal network infrastructure is not in place. For example two or more computers with wireless network access can be able to talk to each other without any configuration being necessary by the user and no other networking devices in place. The link-local addresses is only valid for communication on the local link. I.e. they can not be used for communication across routers.

When many computers configure themselves with an address, one can of course get a situation where there is a conflict. Two computers try to use the same IP-address. In order to solve such problems the protocol specifies that a computer must check for such conflicts, and select a new address if its address conflicts with another one.

2.3.2 Multicast DNS (mDNS)

Although Zeroconf enables automatic configuration of IP-addresses, it does not provide a way of associating a host name with that IP. When no name server (DNS) is available this would mean that users have to know the IP-address of the computer they want to connect to. This address may change each time the machine is switched on or moved to a new location, so one would have to look this up every time.

Multicast DNS[5] is a way to let computers provide a simple DNS service using multicast. The domain .local. is reserved for use on local networks, so a computer can set up its hostname as myhost.local. When no DNS service is available in the network, the computers can use multicast DNS to look-up the IP-address of a host instead of normal DNS. The requests and replies used in the communication are the same as for normal DNS. The main difference being that mDNS messages are sent to a multicast address reserved for that. Thus users only need to remember their host name which will stay the same even if the address changes.

In some cases, two users may have selected an identical hostname. In that case the name has to change to avoid conflicts. This is however very rare.

2.3.3 DNS-based Service Discovery (DNS-SD)

DNS based service discovery[4] is Rendezvous' solution for advertising and locating services. The protocol make use of the existing DNS technology to do the advertisement and discovery. Only standard DNS requests are used for the service discovery. The protocol defines a standard for how the DNS entries should be formatted for DNS based service discovery. If no conventional DNS server is available, mDNS can be used.

The protocol reserves the subdomains ._tcp and ._udp for service discovery. A service is identified by a service instance name. This is given as

```
<Instance>.<Service>.<Domain>
```

The <Instance> part is a user friendly description of the service. e.g. "My ftp server". The <Service> part gives the type of service, e.g. "_ftp._tcp". Finally, the domain is a normal domain name, like "smartfrog.org".

The registration of a service is handled by sending a registration message to the DNS server. A DNS TXT record can hold additional information, like special attributes of the service. These are given as name-value pairs of the form "name=value".

When a service is to be found, the client sends a DNS PTR request to the DNS server. This will return the service instance names of all the advertised services of the requested type. When an instance has been selected, a DNS SRV request is sent to get the needed information about how to contact that service.

In Mac OS X, Rendezvous (DNS-SD) is used to advertise and locate shared file systems, ftp servers, shared music libraries, printers and more. For example to configure a printer, the user just starts the printer setup utility and selects a printer from a list of available printers. This will of course only work when the printers advertise their services using DNS-SD. As an alternative discovery method, Mac OS X can also use SLP.

2.3.4 Rendezvous vs. SLP

As seen, Rendezvous is more than just service discovery. It also enables automatic configuration of IP-addresses, and the use of multicast DNS. The DNS based service discovery used in Rendezvous provides an alternative to the service discovery method used in SLP. Rendezvous relies on the existing DNS technology so it needs no special software for storing the advertisement. SLP on the other hand requires new software for storing advertisements, the Directory Agent. This reuse of existing technology makes Rendezvous interesting, as little work is required to get it running. Most networks will already have a DNS server running. It is possible to use a separate server for the service discovery by assigning the subdomains _tcp and _udp to that server.

Rendezvous is a quite new technology and has until lately only been available in Mac OS X. Versions for Windows and Linux/Unix are now also available. SLP has been around for a few years now. The latest version of the protocol is from 1999.

A number of implementations, both free and commercial are available. In addition SLP is supported by many hardware devices. Many network printers, for example, can advertise themselves through SLP. A number of manufacturers have stated that future products will support Rendezvous, so in time, Rendezvous may be as well supported as SLP is now.

So, Rendezvous requires little extra software to work as it relies on the existing DNS system. SLP requires special software to be deployed, but is currently more supported than Rendezvous.

2.4 Mesh Enhanced SLP (mSLP)

Mesh enhanced SLP is an open source implementation of the service location protocol. It is developed at Columbia University, New York. mSLP is not a fully standard implementation of SLP. Instead, they focus on an extension to SLP. This extension enables multiple directory agents to communicate in a mesh structure. mSLP does not have a normal service agent that can answer multicast service requests. The implementation requires that at least one DA is always running. A combined SA/UA application can register services and send requests to this DA. When a SA registers a service with one DA, this registration will be propagated through the mesh of DAs so that eventually all the DAs know about the service. The SA thus only needs to register its services with one DA. The UA will send its requests to one of the available DAs.

To make this work with SmartFrog, a wrapper library was written. This library implemented the standard Java API for SLP[8], and could be used by SmartFrog components to register and deregister services. While this worked, it would be better to have a standard implementation of the protocol. For example if no DA was available when a service was to be registered, the registration would fail. Having a real service agent would remove this problem.

The implementation created for this thesis will follow the SLP standard very closely, and allow user agents and service agents to work without any DA available. The implementation of the directory agent will also be standard and not use mesh enhancement or other extensions. The API provided to SmartFrog components and other programs using SLP will essentially be the same as in the mSLP wrapper library, as the implementation described in this thesis will also use the standard Java API for SLP.

Chapter 3

SLP Library Implementation

This chapter will explain the implementation of the Service Location Protocol library that is the result of the work on this thesis. The chapter starts with a quick introduction giving the implementation goals and some thoughts on how the library was implemented. It then goes on to show how the implementation was done.

The library itself is divided into several Java packages grouping similar classes together. To explain how the library is implemented, I will first explain the classes in each package. Then I will show how everything is put together into a working implementation of the Service Location Protocol. In this chapter I will only say what each class does. I will not list and explain the methods in the classes. The methods that are important for the use of the SLP library is explained in the SLP user guide given in appendix E. The Javadoc files included on the CD-ROM will also provide a lot of information.

3.1 Introduction

The Service Location Protocol version 2 is standardised by the Internet Engineering Task Force, IETF. The specification of the protocol is given in RFC-2608[9]. Another document, RFC-2614[8] defines a suggested Java API for the protocol.

The goal of this implementation is to follow the specified standard as closely as possible. In order to maintain easy portability between this library and other implementations, the public API provided by this library follows the specification in [8].

Even though the SLP library is to be used within the SmartFrog framework, the core library itself is in no way tied to SmartFrog. The binding to the SmartFrog framework is done through special SmartFrog components capable of using the features present in the core SLP library. This means that the library can be used directly from standalone applications outside of the SmartFrog framework. Such applications will use the SLP API directly in their program code instead of going through the SmartFrog components.

Some parts of the library do not fully comply to the standard. The parts in question

are put forth in Section 3.9. The deviation from the standard should not introduce any major incompatibilities with other SLP implementations. The library should be fully capable of communicating with other SLP implementations.

3.2 Package org.smartfrog.services.comm.slp

This package contains the interfaces and classes defined in RFC-2614 - An API for Service Location[8]. The package also contains the SmartFrog components for integrating SLP into a SmartFrog application. People wanting to use the SLP library only need to know the classes and interfaces found here. Everything else is in general only for internal use by the library itself. The classes and interfaces found here are:

- **Interface Advertiser**
  The Advertiser interface is defined in RFC-2614[8] and has all methods required for registering and deregistering the services to be advertised by SLP. This interface is implemented by the ServiceAgent. Users of the library will get an advertiser object from the service location manager. The implementation of the service agent is of no concern to the user, as they only have to worry about the API defined by the Advertiser interface.

- **Interface Locator**
  The Locator interface defined in RFC-2614[8] has all methods required for finding services, service types and service attributes through SLP. The Locator interface is implemented by the UserAgent. As with the Advertiser, the user will get a Locator from the service location manager and does not have to worry about the actual implementation of the User Agent.

- **Interface SFSlpAdvertiser**
  The interface for the SmartFrog advertiser component. This is currently an empty interface, but methods to register and deregister services through the component may be added.

- **Interface SFSlpDA**
  Interface for the SmartFrog Directory Agent component. This interface is currently empty, and will most likely stay that way. The DA receives registrations and requests from Service Agents and User Agents over the network. It has no methods that are meant do be accessed from the outside.

- **Interface SFSlpLocator**
  Interface for the SmartFrog locator component. This is currently empty, but methods to use the discovery methods of the SLP library through this component may be added.

- **Interface ServiceLocationEnumeration**
  Interface for the enumerations returned by service requests. The interface extends the Java enumeration interface, and defines one extra method: next() which returns the next object in the enumeration. The specification say that this method should block until results are available. In this implementation, the discovery methods are synchronous so the enumeration will contain all possible results when we get it. This method therefore returns immediately. If no more results are available it returns null. Thus this method is identical to the nextElement() method found in the Java Enumeration interface.

- **Class SFSlpAdvertiserImpl**
  Implementation of the SmartFrog advertiser component. This component is given a link to the thing to advertise. This may be another component, a String or some other Object. One must also provide the service type to use for the service, the attributes of the service (if any) and the life time of the advertisement. The advertiser will create a service URL to advertise the given object. This URL is then registered with SLP through an Advertiser.

- **Class SFSlpDAImpl**
  Implementation of the SmartFrog DirectoryAgent component. This component will create a DirectoryAgent object which will get requests and registrations over the network. The component works by creating a DirectoryAgent object on startup. When the component terminates, it also tells the DirectoryAgent object to terminate, thus shutting down the DA.

- **Class SFSlpLocatorImpl**
  Implementation of the SmartFrog locator component. The locator is given a service type to search for and optionally a search filter to use in the search. The actual search is performed in its own thread, and can be repeated periodically. A component needing the result from the discovery will have a link to the ”result” attribute of the locator component. Whenever sfResolve is called to resolve that link, it triggers the SLP search in the locator. If the locator has completed its search, the result of the search is returned immediately. If not, the method blocks until results are available. For the component using the locator this is no different than resolving any other SmartFrog link.

- **Class ServiceLocationAttribute**
  A class that represents an attribute for a service advertised through SLP. An attribute has an ID and a Vector of values. A vector of ServiceLocationAttribute objects are given along with a ServiceURL when a service is registered.

- **Class ServiceLocationException**
  Exception thrown whenever an error occurs in the SLP library. The exception includes an error code indicating which kind of error occurred. The exception

may also include a text description of the problem. The possible error codes are given in RFC-2614[8].

- **Class ServiceLocationManager**
  Manages access to the SLP framework. The service location manager is used to get Advertiser and Locator objects that can be used to register services and find services through SLP. Any program wanting to use a Locator or Advertiser object should get this through the ServiceLocationManager class. The methods used for obtaining the advertiser or locator are given in the SLP user guide found in Appendix E.

- **Class ServiceType**
  Represents a service type in the SLP library. The class has methods to get the different parts of the service type string.

- **Class ServiceURL**
  Represents a ServiceURL in SLP. A service URL has a service type, a hostname and a path to the service on that host. The URL is a standard URL of the form service_type://hostname/path. The different parts of the URL can be obtained by using the appropriate methods provided by this class. As well as the URL itself, the ServiceURL class holds the life time for the URL. This controls how long the service is advertised. Advertisements that are not renewed will eventually be deleted as their life time reaches zero.

More information on the usage of the library can be found in the SLP user guide in Appendix E.

3.3 Package org.smartfrog.services.comm.slp.messages

The messages package contains classes representing all the different SLP message types. All message classes are subclasses of the SLPMessageHeader class, also included in this package. Figure 3.1 shows the header class along with two message classes. These should give an impression on how the implementation relates to the definition of the messages in the SLP specification[9].

The most important feature of the message classes are that they know how to read and write their data in the wire format specified by the service location protocol. The classes also have methods which can be used to get different parts of the SLP message. Typically an empty message of a given type is created when a message is received. The message object is then told to read its data from an SLPInputStream. When messages are to be transmitted, a message object containing all the relevant data is created, and the object told to write its content to an SLPOutputStream.

An overview of the different message types used in the Service Location Protocol was given in Section 2.2. There are one class for each type of message implementing what is required to represent that message type.

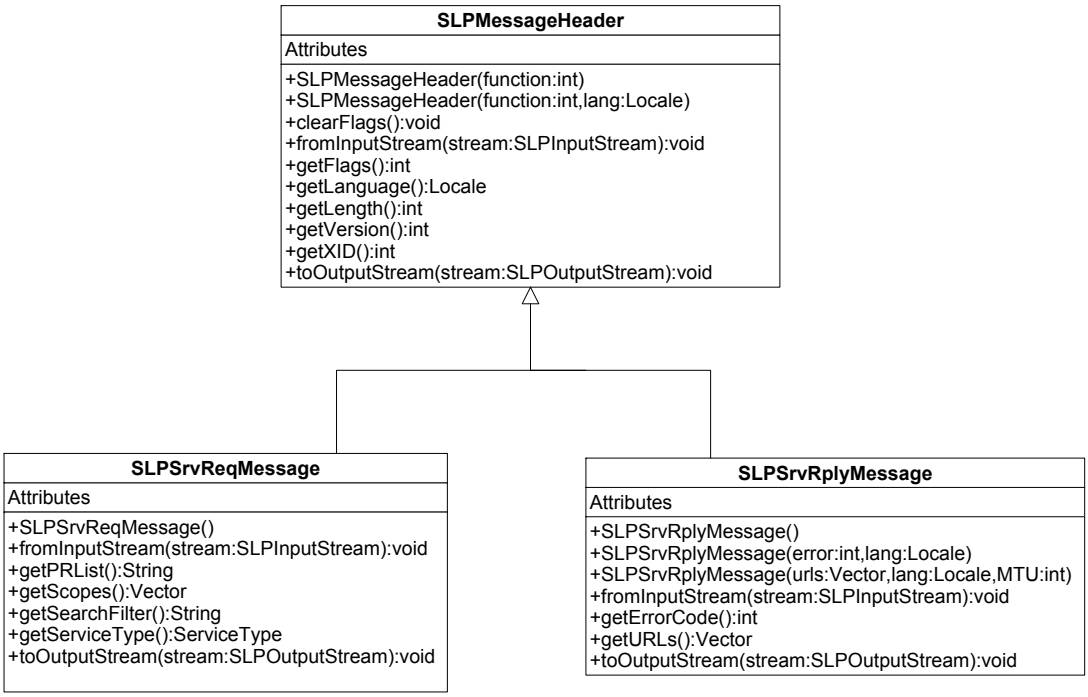

Figure 3.1: SLP Message Classes (Class Diagram)

3.4 Package org.smartfrog.services.comm.slp.network

The network package contains classes to handle network traffic. The service location protocol must support both UDP and TCP communication. These classes simplify the use of this within the library. The network classes will listen for incoming data on the network, and call the appropriate callback functions when a message is received. Classes that are to receive data from the network must implement these callback functions. Figure 3.2 shows the classes handling UDP traffic.

Overview of network classes:

- **Class SlpUdpClient**
  This is the base class for all UDP network classes. The UDP classes are SlpUnicastClient, SlpSharedUnicastClient and SlpMulticastClient. The SlpUdpClient class implements the basic functionality for receiving messages over the network. It also has the default implementation of all accessible methods for the UDP classes. In many cases this is simply throwing an exception saying that the method is not implemented. For example, the SlpMulticastClient class can not be used to send messages.

- **Class SlpUnicastClient**
  Class for handling unicast traffic. That is, traffic that is sent to a valid local address. Messages to multicast groups are not received by this class. This class

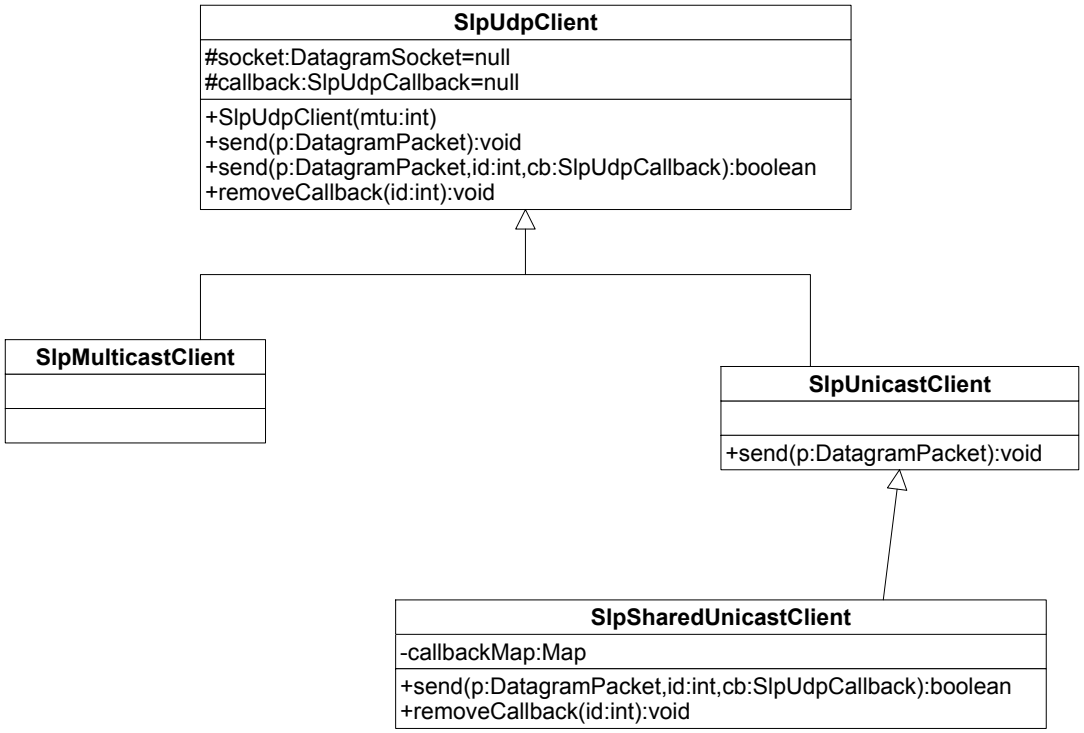

Figure 3.2: UDP Network Classes in SLP Library (Class Diagram)

is also able to send data to a remote address.

- **Class SlpSharedUnicastClient**
  This class is a subclass of the SlpUnicastClient class. It can be used by multiple threads simultaneously to send SLP messages. A special send method enables senders to register the XID of their message with the class. The XID is a unique identifier included with every SLP request. When a reply is received, the class checks the XID of the reply and invokes the callback method of the correct object based on this. A reply always has the same XID as the request causing it to be sent.

- **Class SLPTcpClient**
  The SLPTcpClient class is used when the library has to deal with messages that can not fit inside a single Datagram packet. The UA use this to resend a request if the reply overflows. This way it can get all the possible replies. The SA use it to send registrations to the DA if the registration message is too big for a single Datagram.

- **Class SLPTcpServer**
  This class is used to listen for incoming TCP requests. When a request is received, it accepts the connection and pass the request on to the DA or SA running

the server. Replies are written back over the established connection.

SLP use UDP whenever possible. It will only fall back to using TCP if the messages to transfer are too big for a single datagram. See also the note in Section 3.9 about the use of TCP in this implementation.

3.5 Package org.smartfrog.services.comm.slp.util

The util package contains a collection of classes that are used by various parts of the library. There are generally no real connection between these classes. Each of the classes found here are presented below:

- **Class ParseTree**
  Implements a parse tree for the search filter given in service requests. The class builds a tree from the String given as search filter, and can then match this against a set of attributes.

- **Class SLPDefaults**
  This class contains a number of constants defining the default values for the configurable properties of the SLP library. It also has a method to return a Java Property object with all the default values.

- **Class SLPInputStream**
  The SLPInputStream class is built around a DataInputStream. It contains the methods required to read the datatypes found in the SLP messages from this stream. The class is used by the message classes when they need to read their data from a stream. I.e. When a message is received from the network.

- **Class SLPOutputStream**
  The SLPOutputStream class is built around a DataOutputStream. It contains the methods required to write the datatypes used by the SLP message classes to this stream.

- **Class SLPUtil**
  This class contains a number of static methods used by various parts of the library.

- **Class ServiceAttributeEnumeration**

- **Class ServiceTypeEnumeration**

- **Class ServiceURLEnumeration**
  The Enumeration classes are implementations of the ServiceLocationEnumeration interface found in the org.smartfrog.services.comm.slp package. The enumerations contains ServiceLocationAttributes, ServiceTypes and ServiceURLs

respectively. The URLs in the ServiceURLEnumeration are sorted by decreasing lifetime so the one with the longest lifetime is always returned first.

3.6 Package org.smartfrog.services.comm.slp.agents

This package holds the implementation of the SLP User Agent, Service Agent and Directory Agent and a number of classes related to the implementation of these. Classes found here are:

- **Class DAInfo**
  Holds information about a DA. Used by the Service Agent and User Agent to store information about the DAs they discover.

- **Class DirectoryAgent**
  Implements an SLP Directory Agent. The Directory Agent is implemented according to the specification of SLP version 2 in RFC-2608[9]. The agent listens for requests from user agents, and receives registrations and deregistrations of services from service agents. It will periodically send a DAAdvert message to the SLP multicast address to let other agents know it's there.

- **Class SARegistrationThread**
  Implements a thread used by the SA for handling the registration and de-registration of services with a DA. When a new service registration is received, it will tell this thread to register the service with all known directory agents. As this is done in a separate thread it will not interfere with the SA's ability to receive registrations. Deregistrations are handled in the same way. This thread is also responsible for refreshing the registrations of permanent services with the directory agents.

- **Class SLPAgent**
  Super class for the Service and User Agent classes. Implements the common functionality for these two.

- **Class SLPDatabase**
  Implements the Database used to store service registrations. This is used by the SA and DA for storing the registrations they receive. The database implementation uses a linked list for storing advertisements.

- **Interface SLPMessageCallbacks**
  Defines the methods used to handle incoming messages. This interface defines two methods: handleNonReplyMessage and handleReplyMessage. These are called upon receiving a request or a reply to a request.

- **Class SLPMessageSender**
  Used by the UA to send requests, and the SA to send registrations to the DA.

The sendSLPMessage method of the class will send a given message and wait for replies. If required, it will open a TCP connection and resend the request over TCP. Multiple SLPMessageSender objects can share a SharedUdpClient object, thus transmitting the message over the same socket. This avoids opening a lot of ports on the computer if multiple requests are sent simultaneously. The SLPMessageSender can not be shared by multiple threads. Each thread must create their own instance of this class.

- **Class SLPRequestHandlers**
  A collection of static methods used by the SA and DA to handle incoming requests. This is to reduce the amount of duplicate code in the SA and DA. I can not create a super class for the DA and SA since the SA is already a subclass of SLPAgent. Java does not allow multiple inheritance.

- **ServiceAgent**
  Implementation of an SLP Service Agent. The service agent implements the Advertiser interface, and the methods defined there are used to register and deregister the services advertised by the service agent. The agent will automatically discover directory agents on the network and register its services with them.

- **UserAgent**
  Implementation of an SLP UserAgent. The user agent implements the locator interface, and the methods defined there are used to perform searches for services, service types and service attributes through SLP. The user agent will automatically discover any directory agents in the network. When it sends a request, it will automatically select directory agents that supports the scopes in which to search and unicast the request to those directory agents. Depending on the scopes used it may be necessary to send the request to more than one directory agent to cover all scopes. If the known directory agents do not cover all the scopes in the request, the request will be multicast to the SLP multicast address.

3.7 Putting It All Together

This section will show how the different parts of the library are used to create a working implementation of the various SLP agents. I will use the Service Agent as an example as that uses most of the library. The user agent and directory agent are using a similar design, but may use different parts of the library to implement the things they need. Figure 3.3 shows a class diagram for the ServiceAgent.

The diagram shown here does not include every class used by the ServiceAgent, but shows the main architecture of the library. Figure A.2 in Appendix A shows a use case diagram for the service agent. I will try to explain what happens internally in the library for the different use cases. The sequence diagrams found in Appendix A will also help showing this.

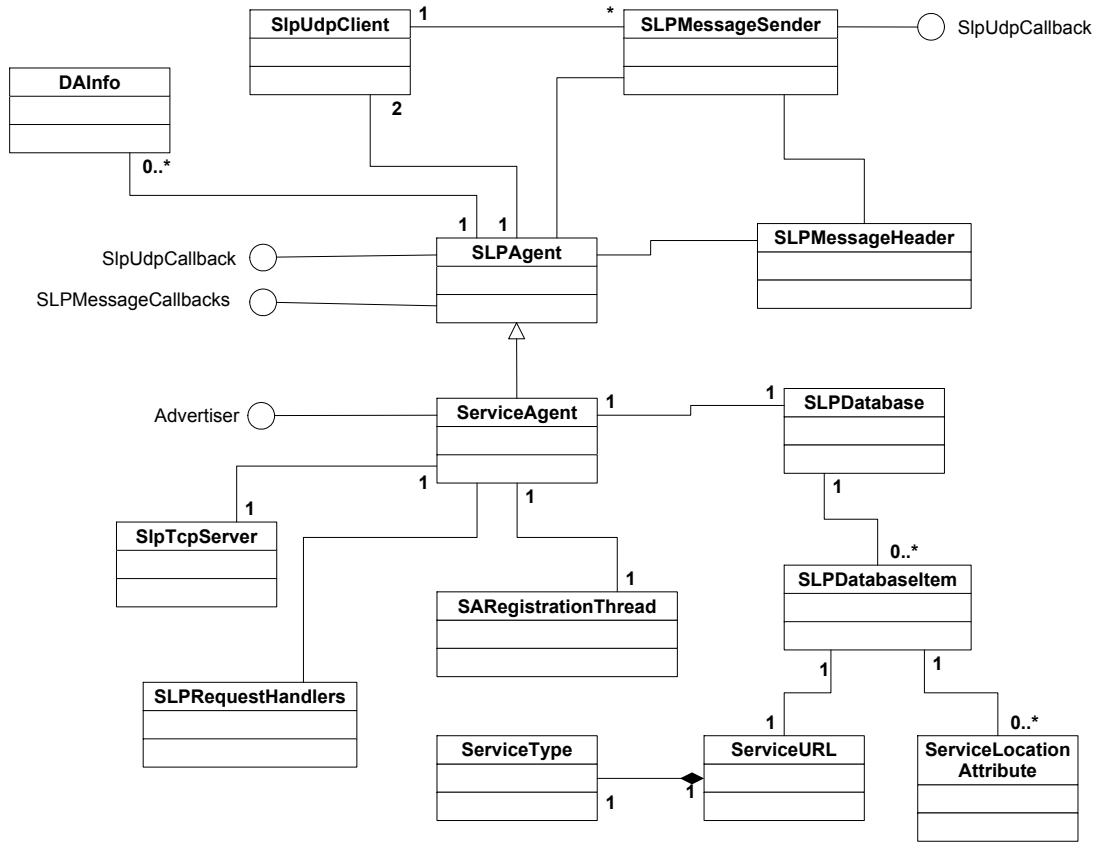

Figure 3.3: Class Diagram for ServiceAgent

The ServiceAgent is used by service providers to register and deregister services advertised through SLP. When a provider has a service to advertise, it registers its service by a call to the ”register” method implemented by the ServiceAgent. The implementation of this method creates a new SLPDatabaseEntry containing information about the new service. The item is then added to the SLPDatabase used by the ServiceAgent. The item is also passed on to the SARegistrationThread to be registered with any Directory Agents known by the ServiceAgent. The registration thread will send a registration message to each known DA registering the service. When a service is deregistered, the SLPDatabaseEntry containing that service is removed from the SLPDatabase, and the SARegistrationThread is told to deregister the service. The registration thread will then send a deregister message to each of the known Directory Agents.

In order to help clients to find the service they are looking for, the ServiceAgent knows how to answer service request, service type request and service attribute request messages. The ServiceAgent gets the requests by listening on the SLP multicast address. This is handled my an SlpMulticastClient object (SlpMulticastClient is a subclass of SlpUdpClient). When a datagram is received by the SlpUdpClient, the method

udpReceived is called on the ServiceAgent object. This method creates an SLPInputStream that can be used to read the bytes included in the received datagram. It also extracts information about the type of message received. After this it calls handleNonReplyMessage providing the stream and type as parameters. Based on the type of message, the appropriate method to handle the received request is called. The ServiceAgent can also receive requests through TCP. The SlpTcpServer class is used for this. When a TCP connection is established, it will create an SLPInputStream to read data from, and call the handleNonReplyMessage method. Thus the request is handled in the same way whether it is UDP or TCP. Handling a request includes searching for an appropriate entry in the database and create a reply if possible. The SLPDatabase class have methods that can be called for finding services of a given type, the different types of services, and the attributes of services. If matching services, service types or attributes (depending on type of request) are found in the database, a reply containing the result is created. If no match is found, or an error occurs, a reply is only created if the request was received on the unicast address.

The ServiceAgent also needs to discover Directory Agents running on the network. To do this it sends service requests looking for the service:directory-agent service. When a message that should trigger a reply is sent, the SLPMessageSender class is used. This takes a message to send and waits for replies. Whenever a reply is received, it will call the handleReplyMessage method in the ServiceAgent or UserAgent that used it for sending the request.

Appendix A has use case and sequence diagrams for the main functionality of the User Agent, Service Agent and Directory Agent implementations. These should give an idea on how the library works. The handling of received messages is pretty much identical for all the agents. All messages that are expected to generate a reply are sent using the SLPMessageSender class. This is true for both the UserAgent and ServiceAgent. The DirectoryAgent never sends messages that expects a reply, so it can send the messages directly using the SlpUdpClient bound to the unicast address.

Appendix E contains the user guide for the SLP library. This will show how to use both the core SLP library and the SmartFrog components written for the library. In the next section I will give a description of how the components are implemented and attached to the rest of the SLP library.

3.8 SmartFrog Components

Three SmartFrog components have been written to allow SLP to be used easily within the SmartFrog framework. These three components are SFSlpAdvertiser, SFSlpLocator and SFSlpDA. The advertiser and locator components use the standard methods provided by the Advertiser and Locator interfaces to access the core SLP library. This is done in the same way as any other program would use the library. The SLP user guide in Appendix E has information on how this is done. In addition to the components, a special class for deploying SmartFrog components, SFSlpDeployerImpl, has

been written. The implementation of the components and deployer class will be described in the next sections, with most detail on the SFSlpLocator and SFSlpAdvertiser components.

3.8.1 SFSlpLocator

The SFSlpLocator component is used to locate things advertised through SLP. Other components use this to find a SmartFrog component or some other object like a String, Integer or SmartFrog Reference. To get the result of a service discovery a component use a link to the ”result” attribute of the locator component. i.e.

```
someObject LAZY locator:result;
```

The SFSlpLocator component has a special implementation of the sfResolve method described in Section 2.1. Whenever the ”result” attribute is to be resolved, this triggers the service discovery. The actual discovery is done in a separate thread, and can be repeated periodically. If a result is available when sfResolve is called, that result will be returned. If the service discovery is not completed, the method will wait until the results are ready and then return the discovered object. Pseudo code for the sfResolve code is given below.

```
public Object sfResolve(Reference r, int index) {
    if(r.elementAt(index).toString().equals("HERE result") ) {
        wait for service discovery to complete;
        select a discovered object to return;
    }
    else {
        return super.sfResolve(r, index);
    }
}
```

The if-statement checks that the ReferencePart at the current index is a reference to the ”result” attribute. This means that the result of the service discovery is requested. The ”returnEnumeration” attribute of the SFSlpLocator can be set to ”true” to indicate that the ServiceLocationEnumeration object returned from the service discovery is to be returned as is. The default operation is to select the first service in the result, and create the Object to return from this. Depending on the format of the service URL, different objects are returned. The URLs in the enumeration are sorted by decreasing life time, thus always having the one with the longest life time as the first element.

If the attribute to resolve is not the ”result” attribute, the standard implementation of sfResolve is used to find the correct Object to return. The full SmartFrog description file for the SFSlpLocator is given in Appendix B.

3.8.2 SFSlpAdvertiser

The SFSlpAdvertiser component is used to advertise things through SLP. It has attributes to define the service type, lifetime and service attributes for the thing it advertises. The attribute ”toAdvertise” has the value to advertise. This may be a simple

value like a String or Integer, or a link to some other attribute. If toAdvertise is a link, one has the choice of advertising the resolution of the link or the link itself. When the component starts, it builds a service URL to advertise the given object. The format of this URL depends on the type of object to advertise. The basic format is

```
serviceType://hostname/path
```

When objects other than String or InetAddress are advertised the hostname part of the URL is usually empty, giving the URL

```
serviceType:///someObject
```

The someObject part is given in plain text for objects that are easily represented as a String. This includes Numbers, Boolean and String. Other objects are serialised to a byte array and the BASE64 encoded String created from that array is added to the URL. The object can then be recreated by the SFSlpLocator when it receives the URL. The full SmartFrog description of the SFSlpAdvertiser component is given in Appendix B.

3.8.3 SFSlpDA and SFSlpDeployerImpl

The SFSlpDA component is a component for running a Directory Agent on a host. The component does nothing more than creating an instance of the DirectoryAgent class on startup, and makes sure the DirectoryAgent is terminated when the component terminates. The full source code and SmartFrog description for this class is given in Appendix B.

The SFSlpDeployerImpl class is used in place of the default deployer class if one wants to deploy components in a ProcessCompound advertised using SLP. The class use an SLP Locator to look for the advertised ProcessCompound. When the results are collected, the first result is used (if multiple results are available). When the process compound is found, the sfProcessName and sfProcessHost attributes are added to the description of the component to deploy, and the standard SmartFrog deployer class is told to deploy the component. The standard class will use the sfProcessName and sfProcessHost to locate the correct ProcessCompound. A typical usage is:

```
sfConfig extends Compound {
    sfDeployerClass "org.smartfrog.services.comm.slp.SFSlpDeployerImpl";
    slpConfig extends SFSlpConfiguration {
        serviceType "service:pc"; // serivce type of the advertised PC.
        // any other SLP configuration
    }
    // normal components
}
```

The ”slpConfig” component description must be included, and have the attribute ”serviceType”. If not, SLP will not know the type of service to look for. Other possible configuration options for SLP is given in the SLP manual in Appendix E.

3.9 Non-Standard Implementation

Some parts of the library differ slightly from the behaviour defined in the standard. This was necessary in order to implement the library the way it is done. I will explain the reason for my choice for each point.

- **Service Agent does not accept unicast datagrams**
  In order to allow multiple service agents to coexist on the same host, they do not listen for unicast messages on the default SLP port. The service agent can only receive requests that are multicast. User agents should use multicast when communicating with service agents, so this should not be a problem. If a user agent tries to be clever and unicast its requests to one specific service agent, that will fail.

- **TCP connections**
  The service agents do not listen for TCP on the default SLP port. Instead they listen on their own special port. This is again to allow multiple service agents to coexist on the same host. If a user agent expects the SA to listen on the default SLP port, it will not be able to connect to it. TCP is used if a reply overflows. That is, all possible replies could not fit inside a single datagram. The user agent can then connect using TCP to get all the replies. The user agent in this implementation tries to make the TCP connection to the port from which it received the reply, which will work. As long as the user agent does this, it will work. In any case, the user agent should be able to handle the failure to connect and return the services included in the original reply.

A way to solve the problem of having multiple SAs running on the same host is to create a SA Server daemon that SAs can register their services with over the loopback interface. This will then represent all the SAs on the host. This means that the SA Server must be running for the SAs to work. I did not want the SAs to depend on any other components being present. The changes described above allowed this.

The Directory Agent will accept unicast datagrams and TCP connections on the default SLP port, so these issues are only related to the Service Agent.

Chapter 4

Testing the Library

During the work on the SLP library, a number of tests were run in order to check that the library worked correctly. Some of these tests were simple tests where I just checked that a message was received an understood by my library. To make sure that my messages had the correct format, I tried to send requests to other SLP implementations. Two open source implementations of SLP were used in these tests, OpenSLP[14] and mSLP[18]. OpenSLP is a C implementation of the Service Location Protocol. mSLP is implemented in Java, and is not a fully standard implementation, although the format of the messages sent using it should be correct.

mSLP has a graphical user interface where one can write the service URL, service type and attributes of the service to register or discover. This was very useful during testing, as it also enabled me to write invalid requests and check that my implementation were able to handle those. For example some errors related to my URL handling were found this way. A malformed URL could in some cases cause my SLP agents to crash, which is not a good thing.

In addition to using the existing SLP implementation to check that my library was able to work together with those, I wrote some simple test programs which I used to test other parts of the library. The various tests will be further explained in the next sections.

4.1 Testing With OpenSLP

OpenSLP consists of a shared library and a daemon process, slpd. The shared library has the required methods for registering/deregistering services and perform service discovery. In other words, the functionality of a SA and UA. The daemon can be used as an SA Server or Directory Agent. The shared library connects to the SA server over the loopback device when services are to be registered or deregistered. On my system I ran slpd as a Directory Agent. This way I could test that my SA registered and deregistered services correctly and that the UA sent requests and handled replies correctly.

4.1.1 Testing the Service Agent

The Service Agent is used to advertise services. In order to test that my agent was able to register services with the Directory Agent, I created a small program that allowed me to enter service URLs to register by hand. I could then see in the log file of OpenSLP that the registrations were correctly received by the DA. When I started testing the User Agent, I used this program without a DA running to check that the replies sent by the SA was the same as the UA got from the DA when that was running. Some problems were discovered on the first tries, but those were mostly related to minor errors in the message classes resulting in slightly wrong format of the sent data. The program I used to test the SA is described in Section 4.3.1.

4.1.2 Testing the User Agent

The User Agent is used to request services on behalf of a client. In order to test this, I wrote a simple program that allowed me to write the service type to search for and a search filter. After completing the discovery, all results were printed on screen. When sending a request to the OpenSLP DA, I could again see in the log file of OpenSLP that the message was received and parsed correctly. Services were registered with the DA either using mSLP or using my test program for the SA. When using my SA test program, I also made some requests without the DA running. These were then sent on the multicast address, and replied to by the SA if any matching service were available.

4.2 Testing with mSLP

mSLP is, as written in Section 2.4, an open source implementation of the service location protocol written in Java. The implementation is not fully standard. They have instead focused on making an extension to the protocol. mSLP has two parts. The Directory Agent and a combined Service and User agent. The directory agent will work as a standard DA if configured to. In my tests, I mainly used the SA/UA part. This way I could check that my own DA implementation returned the expected replies to requests.

4.2.1 Testing the Directory Agent

The directory agent must be able to accept requests for services and registration/deregistration of services. For testing that my DA worked as expected, I used the UA/SA provided with mSLP to send registrations of services and service requests to my DA. My DA was running with all debug options enabled, so it would say what was going on. Using this combination of mSLP and my own DA, I could check that registrations were received and stored correctly, and that it responded correctly to requests. I also tried sending invalid URLs when registering services to make sure that my DA handled

those correctly. That is, discarded the registration and set an error code indicating a parse error in the reply to the SA.

4.3 Special Test Programs

As mentioned in the previous sections, some special test programs were written to test my implementation of the service agent and user agent. These are very simple programs. I will describe these programs in the next two sections. I also had a simple test application for the SmartFrog components I created. This will also be described below.

In addition to these simple test programs, I created a test to check how well my library handled multiple concurrent requests. This test had a simple SA program that advertised one service. Another program was then started that did multiple requests for that service in parallel through one UA.

Chapter 5 will show how the SLP library was used for some components controlling the Portable Batch System, PBS. That also proved to be a good test of the library. PBS is a system for running batch jobs. The created components was able to start the different daemons required by PBS, and dynamically add execution nodes to the system.

4.3.1 Service Agent Test Program

The source code to this program can be found on the CD in the directory tests/ServerTest. The program is very simple, and basically allowed me to enter a URL to register with the service agent. The program uses the standard methods for creating and using the service agent. These methods are explained in the SLP manual found in appendix E.

When started, the program presents a graphical user interface where one can enter a service URL and the attributes for that service. Before one can register services, a ServiceAgent object has to be created. This is done by selecting ”Create SA” from the File menu. Pressing the ”Register” button will register the given service with the SA. A user agent can then be used to search for that service. If the SA know of one or more directory agents, it will automatically register the service with the known DAs. The GUI also display a list of the registered service URLs. Selecting a URL from the list and pressing ”Deregister” will deregister the service represented by that URL.

4.3.2 User Agent Test Program

The source code to this program can be found on the CD in the directory tests/ClientTest. The program is basically a copy of the ServerTest program with some minor modifications. Instead of using a service agent to register services, this will use a user agent and try to locate a service.

When the program starts, the GUI allows one to enter a service type and a search filter for the service to look for. Pressing the ”Search” button will search for that service. The results of the discovery are printed inside the window when discovery is completed. Before starting to search for services, one has to create the user agent by selecting ”Create UA” from the menu.

4.3.3 Testing Concurrent Requests

An instance of my UserAgent class can be shared by multiple threads. All threads can use the UA for sending requests in parallel. To test that this feature worked, and nothing unexpected happened when doing this, I wrote a simple test program that would create a number of threads that all sent a request for a single service. The result of the request were written to a file for each thread. I could then compare the contents of these files to verify that all threads found the same service, as they should.

To advertise the service, a simple program was written that would create a ServiceAgent object, and register one service with this. This would then be the service discovered by the user agent.

The initial tests revealed a minor problem with how multiple threads were handled when sending a request. This was, however, easily fixed. Each request has an ID, which should be unique. When a reply is received, this ID is used to find the correct object to notify. In my implementation I did not check that this randomly selected ID was in fact unique. If a large number requests were sent in parallel, two requests could have the same ID, causing replies to get lost. This was fixed by checking that the ID was unique when creating a request. After this was fixed, the library worked as expected.

The source code for these programs can be found in Appendix C. It is also included on the CD in the tests/threads directory.

4.3.4 Testing SmartFrog Components

One of the goals of the SmartFrog components for SLP was that components should be able to use it without the Java code having to be modified. The first test of the components where simply trying a slightly modified Hello World example. The example that comes with SmartFrog has one application where a Printer component is used by a Generator component to output text on screen. The Generator has a LAZY link to the printer in its description. In my test, I created two applications. One started the Printer component and advertised that using the SFSlpAdvertiser component. The other application started the Generator component, and used the SFSlpLocator component to find the Printer. This test were successful, so I was very happy with that. The SmartFrog descriptions for the two applications are given in Figure 4.1.

Other attributes were also tested. This was done by adding some debug output to the advertiser component to check that it advertised the attribute the way it was intended to do. Similarly some debug output were added to the Locator component to

Printer

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/examples/helloworld/printer.sf"
#include "org/smartfrog/services/comm/slp/components.sf"
sfConfig extends Compound {
    printer extends Printer;
    adv extends SFSlpAdvertiser {
        serviceType "service:sf-prim:printer";
        toAdvertise LAZY printer;
    }
}
```

Generator

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/examples/helloworld/generator.sf"
#include "org/smartfrog/services/comm/slp/components.sf"
sfConfig extends Compound {
    loc extends SFSlpLocator {
        serviceType "service:sf-prim:printer";
    }
    generator extends Generator {
        printer LAZY loc:result;
    }
}
```

Figure 4.1: Test of SmartFrog Components

check that it returned the correct type of object. I did not write any special components for this, but simply used the Hello World example only changing the value of the toAdvertise attribute. This would of course generate an exception when the Generator didn't get a Printer back after calling sfResolve, but I could see from the debug output that the SLP components worked as expected.

4.3.5 Testing the SLP Deployer Class

The SLP deployer class, SFSlpDeployerImpl, was tested in a similar way to the SmartFrog components. I created a simple application which just contained an SFSlpAdvertiser advertising the ProcessCompound in which it was started. I then deployed the Printer application using the SLP deployer class. When the Generator was started, I could then see that the Printer was in the correct ProcessCompound, as it started to write text in the window of that compound. The application advertising the process compound is given in Figure 4.2.

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/services/comm/slp/components.sf"
sfConfig extends Compound {
    sfProcessName "myProcess";
    adv extends SFSlpAdvertiser {
        serviceType "service:process-compound";
        toAdvertise LAZY sfProcess;
    }
}
```

Figure 4.2: Advertising the ProcessCompound

The description of the Printer application is the same as shown in Figure 4.1, but has some extra attributes added to the sfConfig description. these attributes tells it to use the SLP deployer class instead of the standard one, and provides the SLP configuration:

```
sfDeployerClass "org.smartfrog.services.comm.slp.SFSlpDeployerImpl";
slpConfig extends SFSlpConfiguration {
    serviceType "service:process-compound";
}
```

Chapter 5

PBS Components

This chapter will show how a set of components to control the Portable Batch System (PBS) was made. The first section will give a quick introduction to PBS, and present the goals of the work with the SmartFrog components. I will then show how the implementation was done. The work done on the PBS components was done in co-operation with Andreas Unterkircher at CERN.

5.1 Introduction

The Portable Batch System is a system for running batch jobs. A number of execution nodes are available for running the jobs. The jobs to run are sent to the PBS server which will start the jobs on the appropriate node(s). PBS is made up of a set of daemons and tools to manage these daemons. The PBS daemons are: pbs_server, pbs_mom and pbs_sched. More on these in the next sections. Tools to manage the daemons and get information about the running system includes qmgr and pbsnodes. There are other utilities available for getting the status of jobs and so on, but these were not used in this implementation.

PBS is available in two versions. The PBS professional version and OpenPBS. OpenPBS is an open source version of PBS. On our test system, we were running OpenPBS version 2.3.16. PBS was installed on a number of nodes in a Linux/IA64 cluster. We wanted to use SmartFrog to start and stop the required daemons on these nodes. More information on PBS can be found on the OpenPBS homepage[13] and in the PBS Administrator Guide[20]

The next sections will give a brief overview of the PBS daemons and tools used by the SmartFrog components created. The information given is based on how it works on our systems.

5.1.1 Pbs_Server

The PBS server is the centre of a running PBS system. All other daemons and tools communicate with the server in some way over the network. Among the tasks of the server is to create or receive batch jobs. The server is also responsible for sending the jobs to the pbs_mom daemons for execution.

The server is started by running the startup script in /etc/init.d/pbs_server. The running daemon can be configured by using the qmgr command. By using this one can for example add or delete an execution host from the server's list of such hosts. The server will only send jobs to hosts that are in this list.

5.1.2 Pbs_Mom

The pbs_mom daemon, also referred to as just mom, is responsible for executing jobs. It must be running on each host that should be able to run batch jobs. The jobs are sent to the pbs_mom daemon by the server. From time to time, the server will also send a status query to which the mom has to reply. If the mom fails to reply, the server will assume it is down, and don't send any jobs to it.

For the mom to accept jobs and queries from the server, it needs to know on which host the server runs. By default only localhost may connect to the mom. When the server is running on a different host, the hostname must be given in the configuration file for the mom daemon. Similarly, to allows the PBS scheduler to contact the mom, the hostname of the host running the scheduler must be given in the mom configuration file. On our systems, this file is located in /var/spool/pbs/mom_priv/config. The mom daemon is started by running the startup script /etc/init.d/pbs_mom.

5.1.3 Pbs_Sched

The scheduler controls the order in which jobs are executed and on which host they are executed. The scheduler communicate with the mom daemons to get information about which resources are available and other status information. The scheduler also communicate with the server to get information about the jobs that are ready for execution. The scheduler is started by running /etc/init.d/pbs_sched.

5.1.4 Tools

A selection of tools, or commands, is provided with OpenPBS. These can be used to configure the system, and to get information about things like jobs, job queues and the status of the PBS daemons. The tools used for the SmartFrog components are qmgr and pbsnodes.

- **qmgr**
  qmgr can be used for updating the configuration of the server and get status

information from the server. In our SmartFrog components, we used this command to add hosts to the the list of available execution hosts (hosts running the pbs_mom daemon). The command was also used to delete nodes from this list when they were no longer available.

- **pbsnodes**
  pbsnodes can be used to get information about the execution hosts known by the server. When run with the ”-a” option, the command will list all execution hosts known by the server. We used this to make sure the host we wanted to add was not already known by the server when a new node was to be added by using SLP discovery.

5.1.5 Goal of SmartFrog Components

The SmartFrog components controlling PBS should be able to start and stop the different PBS daemons on a node. By using SLP, we should be able to have a number of nodes running that are not originally part of the PBS system. That is, they are not known by the server and will not receive jobs. If an extra node needs to be inserted into the system, one should be able to use SLP to find one of these spare nodes and add it. A special SmartFrog component could be used for this. The idea is that one does not need a specific node to be added to the system. One just need one of the available nodes. Its location is not important.

It should also be possible to start a node and have it registered with the server immediately. We therefore have these two options:

- A node sends a message to the server when it starts telling the server to add it to the system. The node will then be able to receive jobs.
- A node starts up and is advertised through SLP. When a new node is to be added to the system, a special SmartFrog component is started which does a search for spare nodes using SLP. After the search, one of the discovered nodes are registered with the server.

Figure 5.1 shows how a running PBS system could look. An arrow from the server to the execution node indicates that the server knows about the node and can send jobs to it. The other nodes are advertised using SLP and can be added later.

5.2 Implementation of SmartFrog Components

The original SmartFrog components for controlling PBS were written by Andreas Unterkircher. These did not use SLP, thus only the first method was supported. I then took the job of adding SLP support to this. The original components did not change in order to support SLP. A special SLP component for advertising the PBS node was written. This component knows how to correctly advertise the execution node.

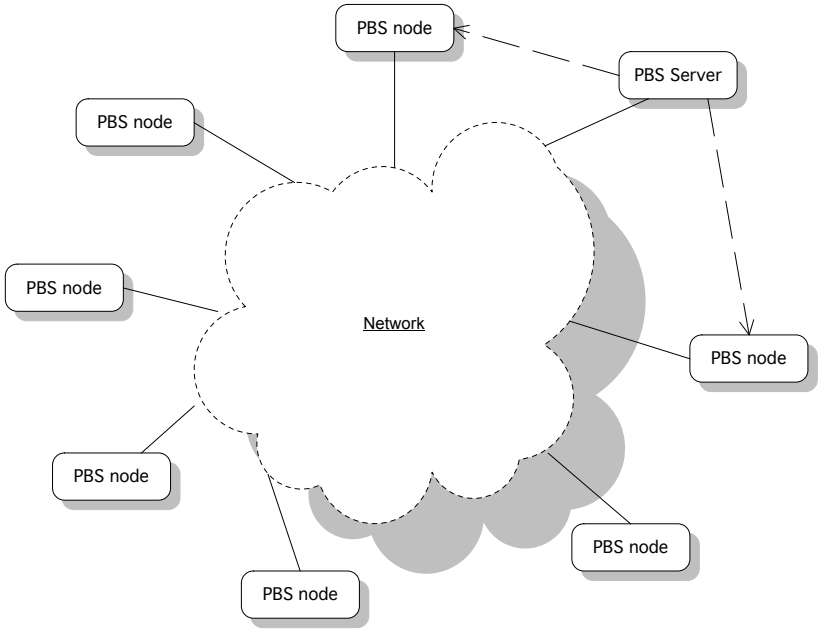

Figure 5.1: Example of a Running PBS System.

In order to discover nodes and register them with the server, a component was written that searched for these advertised nodes and registered one of the found nodes with the server. Registering the node was done by sending an event message to the server. This is the same event that a node would send to register itself.

Next follows a description on each of the components written to create the full PBS system supporting both of the options given above.

5.2.1 PBS Server

The PBS server component, PbsServer, is used to start and stop the pbs_server daemon on a host. The component is able to receive events using the built-in event mechanisms of SmartFrog. An event is a single String indicating that something has happened. Two events are understood by the PbsServer component:

- **pbs node up**
  When received, the node given in the event is registered with the running pbs_server daemon. The required information on the node is encoded into the event string.

- **pbs node down**
  When received, the node given in the event is no longer able to accept jobs, and is deleted from the list of nodes available to the pbs_server daemon.

Upon receiving any of these events, the component uses the ”qmgr” tool to modify the configuration of the running server daemon by adding or removing a node as requested

by the event. Any other events are ignored.

5.2.2 PBS Node

A PBS node is a host capable of executing jobs. The PbsNode component is used to start and stop the pbs_mom daemon that must be running on such hosts. The component is able to tell the PBS server component when the mom daemon is running. It does this by sending an event. For the server to receive this event, the ”sendTo” attribute in the description of the PbsNode component must point to the server. In the case where the node is only to be advertised through SLP, the ”sendTo” attribute is not set, and the event is simply lost. When the component terminates, an event telling the server that the daemon is not running is sent. The events sent are ”pbs node up” on startup, and ”pbs node down” on termination. The format of these events are given below.

- <hostname>:pbs node up:ntype=<ntype>:np=<nprocs>
- <hostname>:pbs node down

In the events, <hostname> is the hostname for the node running the mom daemon. <ntype> is the node type, and <nprocs> is the number of processors in the node. Further explanation of the node type and nprocs attributes can be found in the OpenPBS Administrator Guide[20].

5.2.3 PBS Advertiser (SLP)

The PbsAdvertiser component is a component that can advertise nodes ready to be inserted into the PBS system through SLP. The component is a subclass of the SFSlpAdvertiser component. It overrides the createServiceURL method in order to build the service URL used for advertising the PBS node. The format of the URL used to advertise the node is:

```
service:pbs_node://<hostname>/pbs_node?ntype=<type>&np=<nprocs>
```

<hostname> is the name of the host running the pbs_mom daemon. <np> is the number of CPUs in the node, and <type> is the node type. More information on the ”nprocs” and ”type” attributes of a node can be found in the PBS Administrator Guide[20].

5.2.4 PBS Locator (SLP)

A special component was written to discover advertised nodes and register a discovered node with the server. This component, PbsNodeLocator, is deployed whenever a new node is to be inserted into the system. The component uses SLP to search for advertised nodes and adds one of them to the server. The component checks that the

server does not already know about the discovered node before adding it. If no new nodes are discovered, the component will print a message saying so. The standard SFSlpLocator component is used by the PbsNodeLocator to do the service discovery. The way the component adds a node to the server is to send an ”pbs node up” event to the server with the attributes given in the service URL for the node.

Chapter 6

Results

This chapter presents the final results of my work. The chapter is divided into three parts: The SLP library, the SmartFrog components and classes for SLP and the PBS components.

6.1 Service Location Protocol Library

The main result of my work is the Java implementation of the service location protocol and the SmartFrog components using that. This section will look at the core SLP library without the SmartFrog components. Next section will look at the components created.

My implementation of SLP closely follows the standard specified in RFC-2608[9] and RFC-2614[8]. Exceptions are described in Section 3.9. I have implemented all the mandatory features of the service location protocol for the User Agent, Service Agent and Directory Agent. In addition to the mandatory features, which only includes the ability to do service discovery, I have also implemented support for service type and service attribute discovery. This means that my library supports all the types of requests defined in the SLP specification.

The API presented to external programs that want to use the library is a subset of the API defined in RFC-2614. I have not implemented all classes defined in that specification because some are only for accessing optional features that are not present in my implementation. I have implemented all parts that are relevant to the functionality present in the current version of my library. Additional parts of the API can be supported as new features are added to the library.

The library has been tested in various ways as described in Chapter 4. The current version of my library is fully capable of interacting with other implementations of the protocol. This makes me confident that the library is in fact implemented according to the given standard.

The library has been made available as open source under the GNU Lesser General Public License (LGPL)[21], and the code can be found in the SmartFrog CVS reposi-

tory on http://www.sourceforge.net/projects/smartfrog. The directory core/components/slp in CVS contains the source code for my SLP library, including the SmartFrog components. The source code is also included on the CD attached to this thesis in the slplib/src directory.

The SLP Manual included on the CD and in Appendix E shows how to use the library to advertise and locate services on a network.

6.2 SmartFrog SLP Components

The SmartFrog components for SLP enables the use of the SLP library within the SmartFrog framework. SmartFrog components and attributes can be advertised through SLP by adding the SFSlpAdvertiser component to the description. The advertiser component is configured with service type, attributes, lifetime and the thing to advertise. The ”toAdvertise” attribute decides what is to be advertised. The value of this may be a simple value or a reference to some other component or attribute in the SmartFrog description. If the value of the ”toAdvertise” attribute is a reference, one can set the ”advertiseReference” attribute to ”true” to advertise the Reference object. If that attribute is ”false”, the object pointed to is advertised.

The SFSlpLocator component is used to search for advertised services. It is configured with a service type to look for and an optional search filter. Components that want to use the result of the discovery must have a reference to the ”result” attribute of the locator. When this reference is resolved, the discovered object is returned.

Chapter 4 showed some tests that were done in order to test these components. The result of the tests were very positive. Advertising and locating objects in the SmartFrog framework works well. By using the functionality provided here, SmartFrog applications can find advertised components belonging to other applications without the writer having to know which host the other application runs on. One only has to know the service type of the advertised service. The service type for a given service will stay the same even if the application is moved to a different host. If the components of the application were referenced by using a HOST reference, that reference would have to be updated if the application were started on a different host. Additionally, there may be several identical services running. In many cases it is not important which of these to use. SLP will select one from anywhere in the network.

When SmartFrog applications are deployed, they need to be deployed in a ProcessCompound. This is a special SmartFrog component that can start applications. A process compound can be advertised through SLP by using the standard SFSlpAdvertiser component. To advertise a process compound, one sets the ”toAdvertise” attribute to be a LAZY reference to the ”sfProcess” attribute. E.g.

```
toAdvertise LAZY sfProcess;
```

As with all advertisements, a number of attributes can be registered for the service. If one wants to deploy an application in a ProcessCompound with some special attributes,

one can use the special SLP deployer class. This does a search for advertised Process-Compound components, and returns one of the discovered ProcessCompounds. If no match is found, the standard deployer class is used instead.

The use of the components and deployer class is documented in the SLP manual. The examples shown in Chapter 4 will also give an idea on how these components are used.

6.3 PBS Components

The components for controlling a PBS system that was made during the work on this thesis have been tested successfully on a system with 6 computers. Figure 6.1 shows the setup used for testing this. The 5 computers running the PBS daemons are IA64 systems running Linux. The system description was started from a normal IA32 PC also running Linux. Only the PbsServer and PbsNode components were tested, as those were the only ones implemented at that time. Later, a component for starting the PBS scheduler has been created. This has not been tested in a full system, but the component is able to start and stop the scheduler.

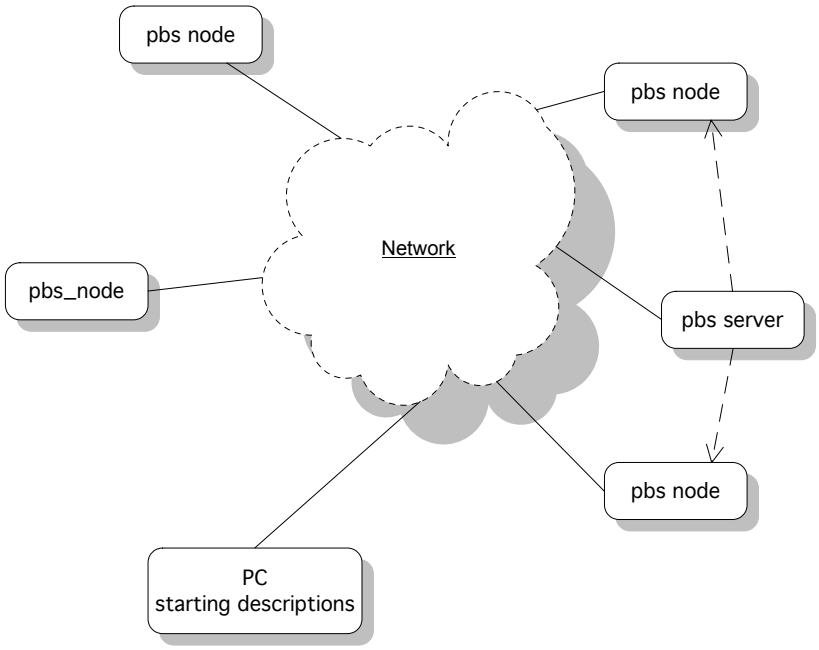

Figure 6.1: Test Set-up for the SmartFrog PBS Components

The components are quite straight forward implementations, only having the ability to start and stop the PBS daemons by using the normal startup scripts for PBS. The PbsNode components can be advertised through SLP for later inclusion into a running

PBS system, or they can be registered with the PBS server on startup. The system shown above has two advertised nodes, and two that are registered with the server on startup. The SmartFrog description file for the system is given in Appendix D. The descriptions and Java implementations for the PBS components are included on the CD in the ”pbs” directory.

Chapter 7

Conclusions and Future Work

SLP provides an alternative way for SmartFrog components of finding other components and services in a SmartFrog system. By using SLP, components from one SmartFrog application can easily obtain references to running components belonging to other applications. A component can also use SLP to locate any other advertised service.

The possibility to use SLP offers a great advantage over the HOST reference, which could also be used to get a reference to components belonging to a different application. The HOST reference requires that you know a lot about the application within which the interesting component is running. If the structure of the application changes or it is moved to a different host, the reference needs to be updated, or the wanted component will not be found. With SLP, the component can be moved around within the application and the application can be moved to another host without any changes being required in other applications. As long as the component is advertised, it will be found. Thus SLP offers more flexibility for the designer of a SmartFrog system.

The implementation of the Service Location Protocol described in this thesis has been successfully tested in a number of ways as described in Chapter 4. The library in its current form is working, and fully useable through SmartFrog components and API calls.

The implementation of the PBS showed one use of SLP, where a number of PBS execution nodes were originally not able to receive jobs as they were not known by the server. SLP made it easy to dynamically add spare nodes to the PBS system. In our implementation we manually started a component that did a search for spare nodes and added one to the system. It would also be possible to have the system automatically add or remove nodes as they become available or is shut down. By periodically using SLP to search for execution nodes, the list of discovered nodes could be compared to the nodes known by the PBS server. Any new nodes could then be added to the system. A node known by the system but not discovered using SLP could be removed.

7.1 Future Work

The implemented SLP library supports all required parts of the Service Location Protocol. However, there are still some optional features that are not currently supported. Future work will include implementing these missing features.

The need of additional SmartFrog components using SLP should be investigated. One thing that might be useful in some cases is a service browser. This would be a component with a graphical user interface displaying all the services it is able to discover. This way an administrator can check which services are available before deploying a SmartFrog application that needs a specific service. Other possibilities includes a component that can send a SmartFrog event message whenever a given service is found. Components can then wait for this event, and then perform the appropriate actions when the service becomes available.

The PBS components implemented for this thesis were very simple, and did not have a lot of error handling. This should be improved in future versions of the components. The current implementation does not check that the node discovered using SLP does actually exist. A way of verifying this should be added to avoid adding nodes that do not exist to the PBS system.

Bibliography

[1] Paul Anderson. ”The Complete Guide to LCFG”.
http://www.lcfg.org/doc/guide.pdf.

[2] Paul Anderson, Patrick Goldsack, and Jim Paterson. ”SmartFrog meets LCFG - Autonomous Reconfiguration with Central Policy Control”. *In the Proceedings of the LISA 2002 System Administration Conference*, 2003.
http://www.lcfg.org/doc/lisa03.pdf.

[3] Stuart Chesire, Bernard Aboba, and Erik Guttman. ”Dynamic Configuration of IPv4 Link-local Addresses”.
http://files.zeroconf.org/draft-ietf-zeroconf-ipv4-linklocal.txt, 2004.

[4] Stuart Chesire and Marc Krochmal. ”DNS-Based Service Discovery”.
http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt, 2004.

[5] Stuart Chesire and Marc Krochmal. ”Multicast DNS”.
http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt, 2004.

[6] Hewlett-Packard Development Company. ”A Brief Description of the Dynamic Web Server Demonstrator”.
http://www.hpl.hp.com/research/smartfrog/papers/sfDynamicWebServerExample.pdf, 2004.

[7] Hewlett-Packard Development Company. ”The SmartFrog Reference Manual v3.02”.
http://www.hpl.hp.com/research/smartfrog/papers/sfReference.pdf, 2004.

[8] E. Guttman and J. Kempf. ”RFC-2614 An API for Service Location”.
http://www.ietf.org/rfc/rfc2614.txt, 1999.

[9] E. Guttman, C. Perkins, J. Veizades, and M. Day. ”RFC-2608 Service Location Protocol v2”.
http://www.ietf.org/rfc/rfc2608.txt, 1999.

[10] Apple Computers Homepage.
http://www.apple.com.

[11] CERN GridCafe Homepage.
http://www.gridcafe.org.

[12] GridWeaver Project Homepage.
http://www.gridweaver.org.

[13] OpenPBS Homepage.
http://www.openpbs.org.

[14] OpenSLP Homepage.
http://www.openslp.org.

[15] Rendezvous Homepage.
http://developer.apple.com/rendezvous/.

[16] SourceForge Homepage.
http://www.sourceforge.net.

[17] T. Howes. "RFC 2254 - The String Representation of LDAP Search Filters". *http://www.faqs.org/rfcs/rfc2254.html*, 1234.

[18] mSLP Homepage.
http://mslp.sourceforge.net.

[19] Todd Poynor. "Automating Infrastructure Composition for Internet Services". *In the Proceedings of the LISA 2001 15th System Administration Conference*, 2002.
http://www.usenix.org/events/lisa2001/tech/full_papers/poynor/poynor_html.

[20] Veridian Systems. *"Portable Batch System v2.3 Administrator Guide"*, 2000.

[21] GNU Lesser General Public License v2.1.
http://www.gnu.org/licenses/lgpl.html.

Appendix A

UML Use Case and Sequence Diagrams for SLP Library

This appendix has UML use case diagrams and sequence diagrams for different parts of the system.

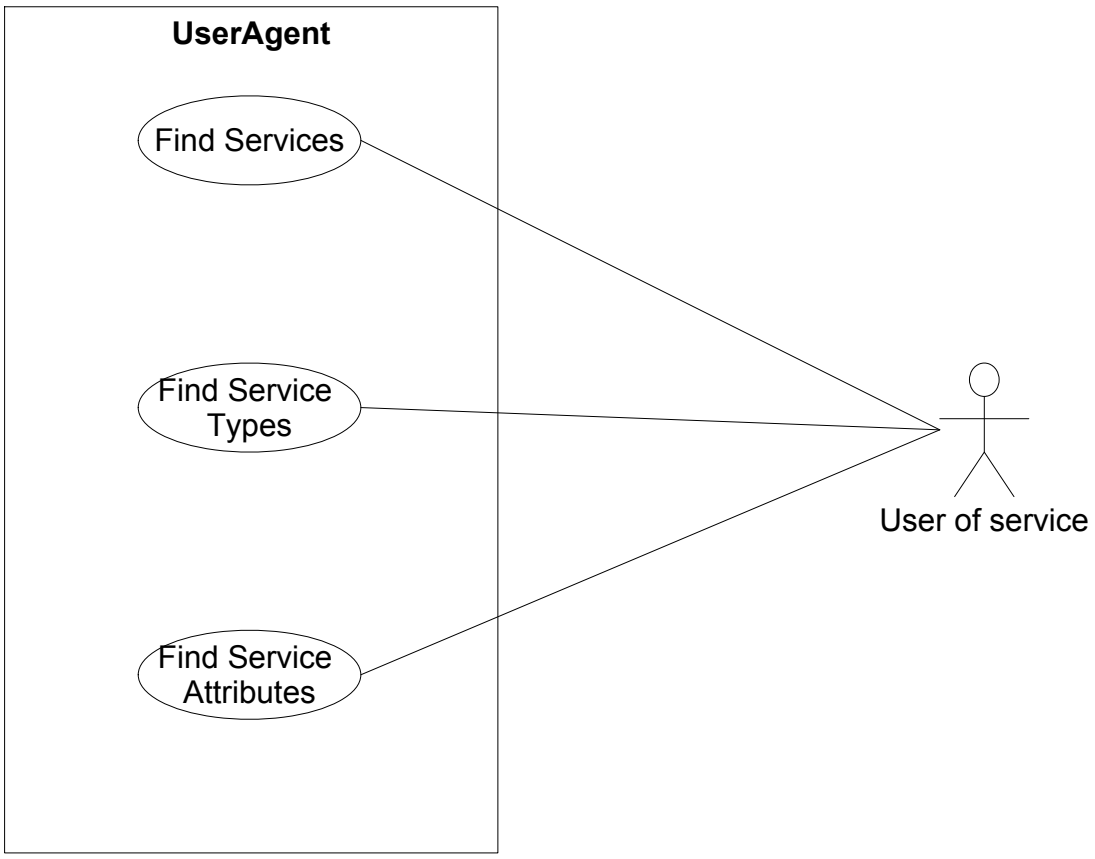

Figure A.1: Use Case Diagram for SLP User Agent (UA)

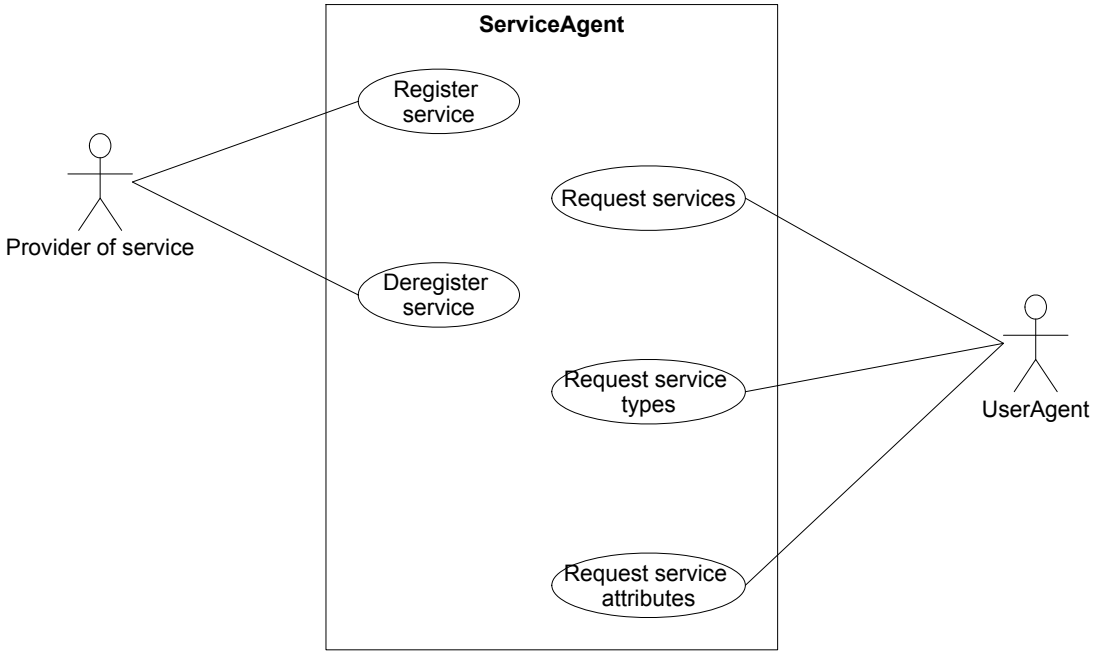

Figure A.2: Use Case Diagram for SLP Service Agent (SA)

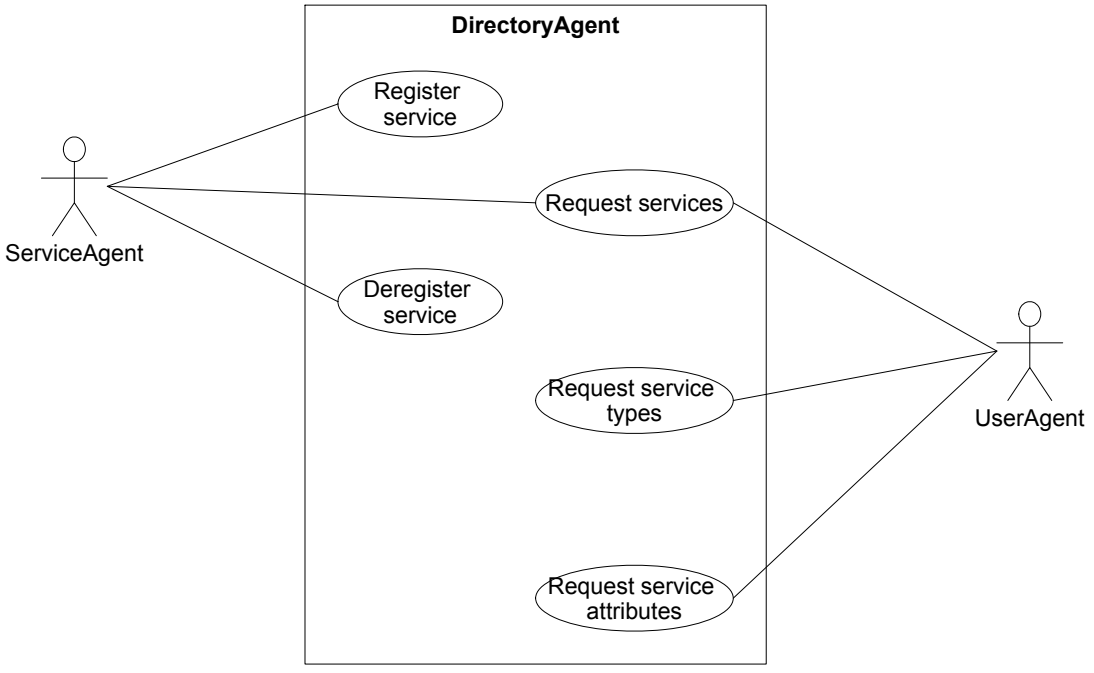

Figure A.3: Use Case Diagram for SLP Directory Agent (DA)

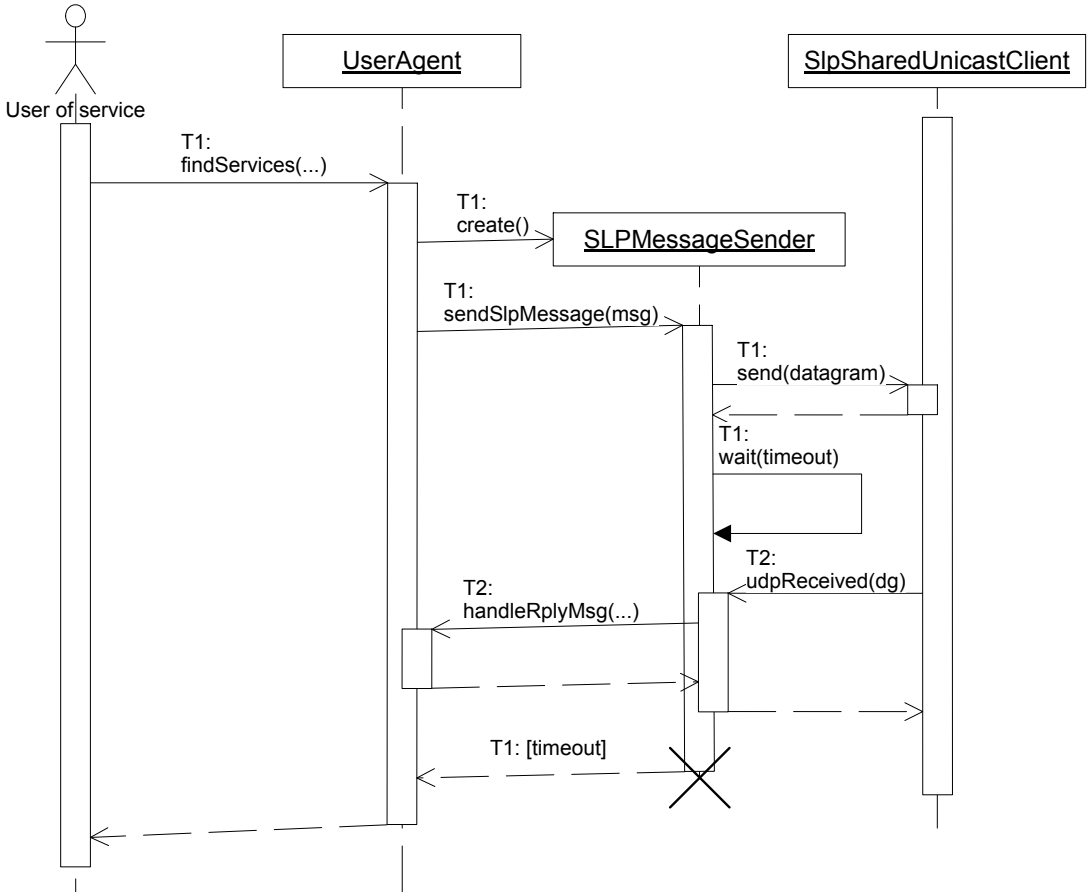

Figure A.4: Sequence Diagram for the UA's Find Services Use Case

Two threads are involved in this operation. Thread1 (marked T1) is used for sending the request over the network. Thread2 (marked T2) is the thread listening for incoming messages on the network. After a message has been sent, this will notify the SLPMessageSender when replies are received

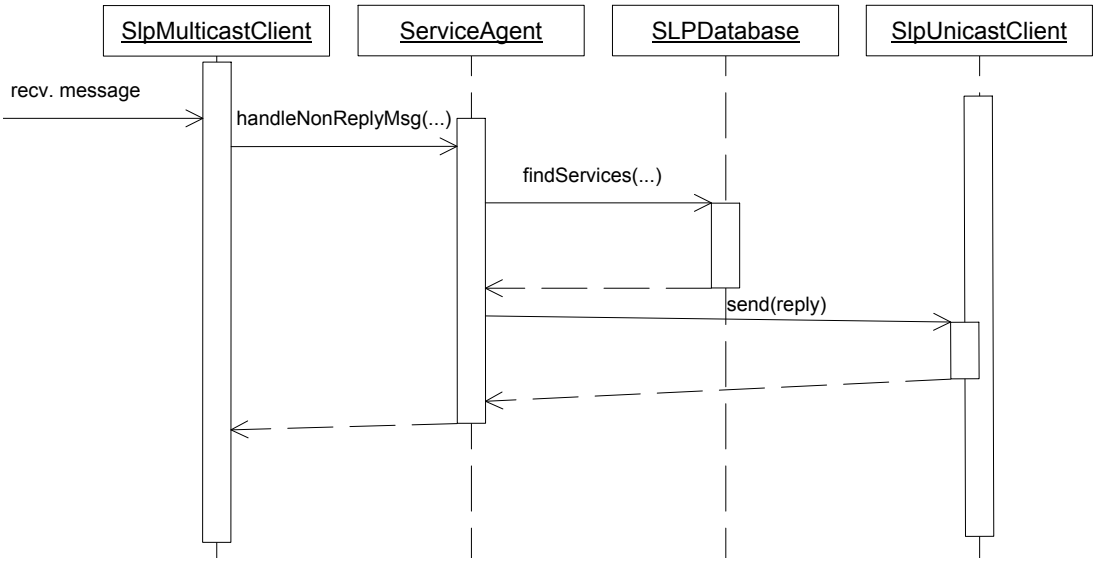

Figure A.5: Sequence Diagram for the SA's Request Services Use Case
The request service usecase for the Service Agent. The thread listening on the multicast address receives a message. This is the message sent by a UA during the findServices use case.

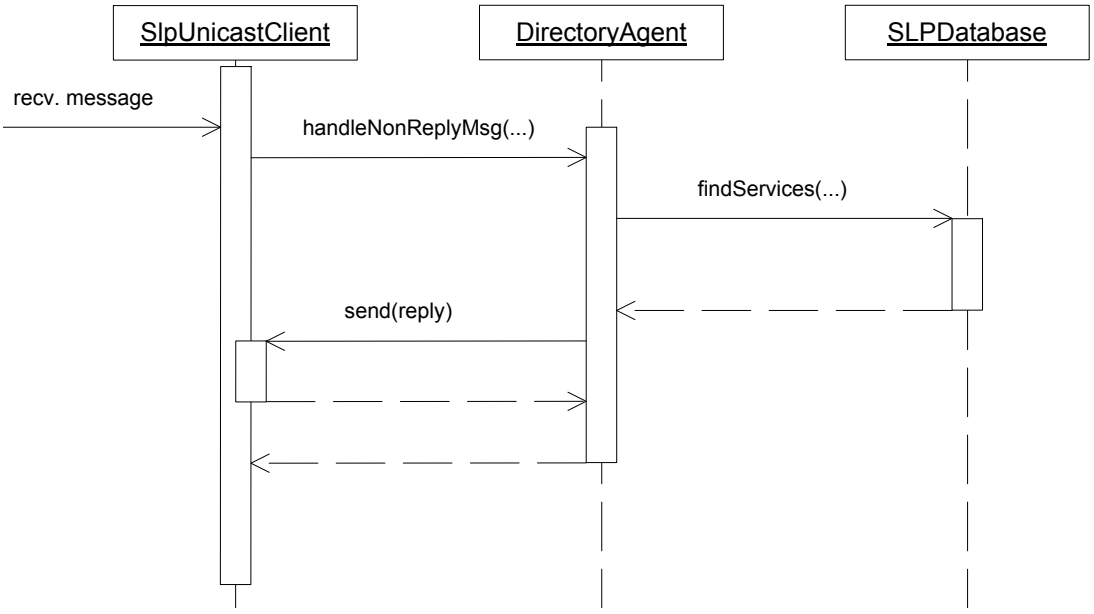

Figure A.6: Sequence Diagram for the DA's Request Services Use Case
The DA receives request on the unicast listener. Apart from that, it works in the same way as the SA.

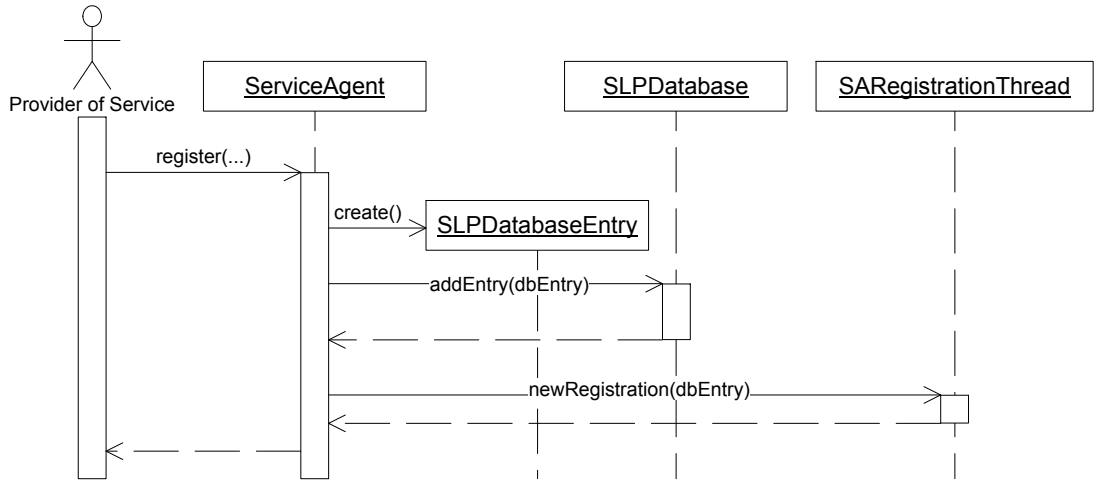

Figure A.7: Sequence Diagram for the SA's Register Service Use Case

Shows a simplified sequence diagram for the register service use case for the Service Agent. After the SARegistrationThread has been notified of the new registration it will send the registration to any DA known by the SA. This is not shown in the diagram.

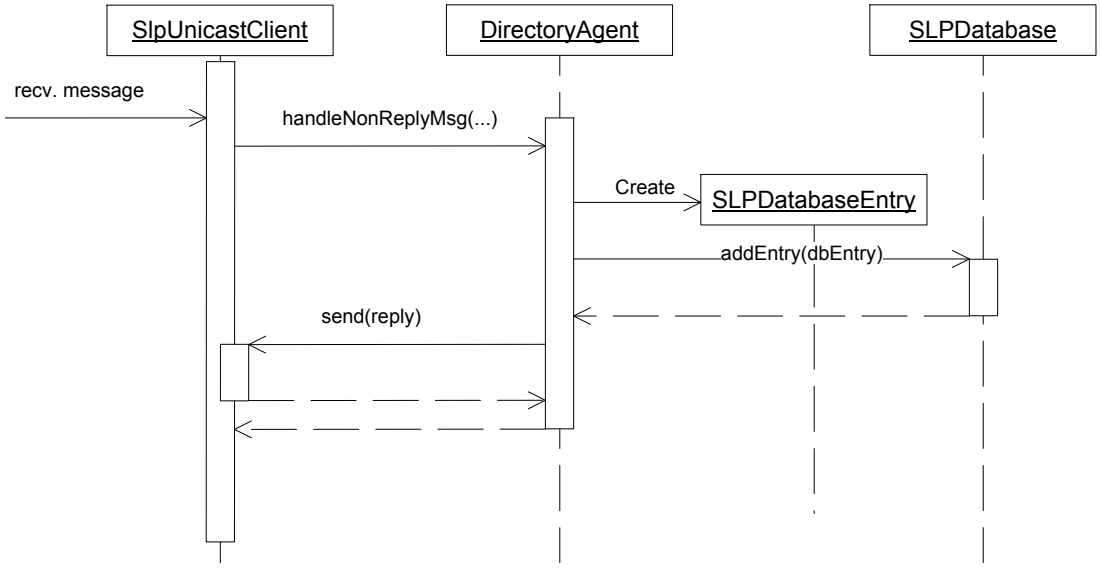

Figure A.8: Sequence Diagram for the DA's Register Service Use Case

When the DA receives a SLP SrvReg message, it will add the given service to its service database. A reply is sent to tell if the registration went well.

Appendix B

Component Descriptions and Source Code

This appendix has the SmartFrog description files for all the SmartFrog components implemented for the SLP library. For the Directory Agent component, the full source code is included as well.

B.1 SLP Configuration

```
SFSlpConfiguration extends {
    slp_config_mc_max 15000; // 15 seconds maximum wait for multicast req.
    slp_config_rnd_wait 1000; // 1 second
    slp_config_retry 2000; // 2 seconds
    slp_config_retry_max 15000; // 15 seconds. maximum wait for unicast req.
    slp_config_da_beat 10800000; // 3 hours
    slp_config_da_find 900000; // 900 seconds
    slp_config_daAddresses ""; // comma-separated list of ip-addresses/hostnames of DAs
    slp_config_scope_list "default"; // comma-separated list of scope names.
    slp_config_mtu 1400; // mtu for slp messages
    slp_config_port 427; // default slp port (used for requests)
    slp_config_locale "en"; // locale to use for the agent
    slp_config_mc_addr "239.255.255.253"; // multicast address to use for slp.
    slp_config_interface ""; // IP-address of the network interface to use with SLP.
    slp_config_debug "false"; // turn debug output on/off
    slp_config_log_errors "false"; // turn on/off logging of errors
    slp_config_log_msg "false"; // turn on/off logging of sent/received messages
    slp_config_logfile ""; // name of log file ("" -> stdout)
}
```

B.2 SFSlpLocator

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/services/comm/slp/sf/SFSlpConfiguration.sf"

SFSlpLocator extends Prim {
    // implementation
    sfClass "org.smartfrog.services.comm.slp.SFSlpLocatorImpl";
```

```
    // slp configuration
    slp_config_mc_max ATTRIB SFSlpConfiguration:slp_config_mc_max;
    slp_config_rnd_wait ATTRIB SFSlpConfiguration:slp_config_rnd_wait;
    slp_config_retry ATTRIB SFSlpConfiguration:slp_config_retry;
    slp_config_retry_max ATTRIB SFSlpConfiguration:slp_config_retry_max;
    slp_config_da_find ATTRIB SFSlpConfiguration:slp_config_da_find;
    slp_config_daAddresses ATTRIB SFSlpConfiguration:slp_config_daAddresses;
    slp_config_scope_list ATTRIB SFSlpConfiguration:slp_config_scope_list;
    slp_config_mtu ATTRIB SFSlpConfiguration:slp_config_mtu;
    slp_config_port ATTRIB SFSlpConfiguration:slp_config_port;
    slp_config_locale ATTRIB SFSlpConfiguration:slp_config_locale;
    slp_config_mc_addr ATTRIB SFSlpConfiguration:slp_config_mc_addr;
    slp_config_interface ATTRIB SFSlpConfiguration:slp_config_interface;
    slp_config_debug ATTRIB SFSlpConfiguration:slp_config_debug;
    slp_config_log_errors ATTRIB SFSlpConfiguration:slp_config_log_errors;
    slp_config_log_msg ATTRIB SFSlpConfiguration:slp_config_log_msg;
    slp_config_logfile ATTRIB SFSlpConfiguration:slp_config_logfile;

    // locator component configuration
    locator_discovery_delay 0; // The number of millisceonds to wait before
                               // the first service discovery attempt.
    locator_discovery_interval 0; // retry discovery at regular intervals.
                                  // 0 = do not retry.

    // service type to look for
    serviceType ""; //e.g. service:http
    searchFilter ""; // Search filter to limit the number of results
    searchScopes []; // scopes to search in. If empty, the scopes returned
                     // from ServiceLocationManager.findScopes() are used.

    // The result of the service discovery
    //result

    // control what is being returned by the locator. The default is to return the
    // first element that is discovered.
    returnEnumeration false; // set to true to return the unmodified ServiceLocationEnumeration
                             // returned by the service discovery.

}
```

B.3 SFSlpAdvertiser

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/services/comm/slp/sf/SFSlpConfiguration.sf"

SFSlpAdvertiser extends Prim {
    // implementation
    sfClass "org.smartfrog.services.comm.slp.SFSlpAdvertiserImpl";

    // slp configuration
    slp_config_mc_max ATTRIB SFSlpConfiguration:slp_config_mc_max;
    slp_config_rnd_wait ATTRIB SFSlpConfiguration:slp_config_rnd_wait;
    slp_config_retry ATTRIB SFSlpConfiguration:slp_config_retry;
    slp_config_retry_max ATTRIB SFSlpConfiguration:slp_config_retry_max;
    slp_config_da_find ATTRIB SFSlpConfiguration:slp_config_da_find;
    slp_config_daAddresses ATTRIB SFSlpConfiguration:slp_config_daAddresses;
    slp_config_scope_list ATTRIB SFSlpConfiguration:slp_config_scope_list;
    slp_config_mtu ATTRIB SFSlpConfiguration:slp_config_mtu;
    slp_config_port ATTRIB SFSlpConfiguration:slp_config_port;
    slp_config_locale ATTRIB SFSlpConfiguration:slp_config_locale;
```

```
    slp_config_mc_addr ATTRIB SFSlpConfiguration:slp_config_mc_addr;
    slp_config_interface ATTRIB SFSlpConfiguration:slp_config_interface;
    slp_config_debug ATTRIB SFSlpConfiguration:slp_config_debug;
    slp_config_log_errors ATTRIB SFSlpConfiguration:slp_config_log_errors;
    slp_config_log_msg ATTRIB SFSlpConfiguration:slp_config_log_msg;
    slp_config_logfile ATTRIB SFSlpConfiguration:slp_config_logfile;

    // service to advertise
    serviceType ""; // e.g. service:http
    toAdvertise ""; // the thing to advertise
    serviceLifetime -1; // a positive integer, or -1 to indicate a permanent lifetime
    serviceAttributes []; // Vector of service attributes
                          // example: [["s1a1", "s1a1v1", "s1a1v2"], ["s1a2", "s1a2v1"]];

    advertiseReference false;
}
```

B.4 SFSlpDA

SmartFrog Description

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/services/comm/slp/sf/SFSlpConfiguration.sf"

SFSlpDA extends Prim {
    // implementation
    sfClass "org.smartfrog.services.comm.slp.SFSlpDAImpl";

    // slp configuration
    slp_config_scope_list ATTRIB SFSlpConfiguration:slp_config_scope_list;
    slp_config_mtu ATTRIB SFSlpConfiguration:slp_config_mtu;
    slp_config_port ATTRIB SFSlpConfiguration:slp_config_port;
    slp_config_mc_addr ATTRIB SFSlpConfiguration:slp_config_mc_addr;
    slp_config_interface ATTRIB SFSlpConfiguration:slp_config_interface;
    slp_config_debug ATTRIB SFSlpConfiguration:slp_config_debug;
    slp_config_log_errors ATTRIB SFSlpConfiguration:slp_config_log_errors;
    slp_config_log_msg ATTRIB SFSlpConfiguration:slp_config_log_msg;
    slp_config_logfile ATTRIB SFSlpConfiguration:slp_config_logfile;

}

sfConfig extends SFSlpDA {

}
```

Implementation Source Code

```
package org.smartfrog.services.comm.slp;

import org.smartfrog.sfcore.prim.PrimImpl;
import org.smartfrog.sfcore.prim.Prim;
import org.smartfrog.sfcore.prim.TerminationRecord;
import org.smartfrog.sfcore.common.SmartFrogException;

import java.rmi.RemoteException;
import java.util.Properties;

import org.smartfrog.services.comm.slp.agents.DirectoryAgent;

/**
```

```
    SmartFrog component for the Directory Agent.
*/
public class SFSlpDAImpl extends PrimImpl implements Prim, SFSlpDA {
    private DirectoryAgent da;

    public SFSlpDAImpl() throws RemoteException {

    }

    public void sfDeploy() throws SmartFrogException, RemoteException {
        super.sfDeploy();
        // get properties
        Properties properties = new Properties();
        String s = (String)sfResolve("slp_config_interface");
        if(!s.equals("")) properties.setProperty("net.slp.interface", s);
        properties.setProperty("net.slp.useScopes",
                               sfResolve("slp_config_scope_list").toString());
        properties.setProperty("net.slp.mtu",
                               sfResolve("slp_config_mtu").toString());
        properties.setProperty("net.slp.port",
                               sfResolve("slp_config_port").toString());
        properties.setProperty("net.slp.multicastAddress",
                               sfResolve("slp_config_mc_addr").toString());
        properties.setProperty("net.slp.debug",
                               sfResolve("slp_config_debug").toString());
        properties.setProperty("net.slp.logErrors",
                               sfResolve("slp_config_log_errors").toString());
        properties.setProperty("net.slp.logMsg",
                               sfResolve("slp_config_log_msg").toString());
        properties.setProperty("net.slp.logfile",
                               sfResolve("slp_config_logfile").toString());

        // create DA.
        try {
            da = new DirectoryAgent(properties);
        }catch(ServiceLocationException sle) {
            throw (SmartFrogException)SmartFrogException.forward(sle);
        }
    }

    // make sure DA stops on sfTerminate...
    public void sfTerminateWith(TerminationRecord tr) {
        da.killDA();
        da = null;
        super.sfTerminateWith(tr);
    }
}
```

Appendix C

Test Programs

This appendix contains the test programs used to test that the SLP library was able to handle multiple requests in parallel.

C.1 Advertising the Service

```
//
//  AdvertiserTest.java
//  AdvertiserTest
//
//  Created by Glenn Hisdal on Mon May 10 2004.
//  Copyright (c) 2004 __MyCompanyName__. All rights reserved.
//
import java.util.*;
import org.smartfrog.services.comm.slp.*;

public class AdvertiserTest {
    private static final String SERVICE_TYPE ="service:testservice";
    public static void main (String args[]) {
        try {
            Properties p = new Properties();
            p.put("net.slp.port", "4427");
            ServiceLocationManager.setProperties(p);
            Advertiser advertiser =
                ServiceLocationManager.getAdvertiser(new Locale("en"));
            ServiceURL url = new ServiceURL(SERVICE_TYPE+"://at.some.server",
                                            ServiceURL.LIFETIME_PERMANENT);
            advertiser.register(url, new Vector());

            while(true) {
                Thread.sleep(10000);
            }
        }catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

C.2 Locating the Service

```
//
//  LocatorTest.java
```

```
//  LocatorTest
//
//  Created by Glenn Hisdal on Mon May 10 2004.
//  Copyright (c) 2004 __MyCompanyName__. All rights reserved.
//
import java.util.*;
import java.io.*;
import org.smartfrog.services.comm.slp.*;

public class LocatorTest implements Runnable {
    private static final String SERVICE_TYPE = "service:testservice";
    private int nextId = 1;
    private Locator locator;
    private ServiceType serviceType;
    private static int numThreads;

    public LocatorTest() throws ServiceLocationException {
        Properties p = new Properties();
        p.put("net.slp.port", "4427");
        ServiceLocationManager.setProperties(p);
        locator = ServiceLocationManager.getLocator(new Locale("en"));
        serviceType = new ServiceType(SERVICE_TYPE);
    }

    public static void main (String args[]) {
        try {
            numThreads = 10;
            if(args.length != 0) numThreads = Integer.parseInt(args[0]);

            LocatorTest test = new LocatorTest();

            // create locator threads...
            for(int i=0; i<numThreads; i++) {
                Thread t = new Thread(test);
                t.start();
            }
        }catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    public synchronized int getNextId() {
        return nextId++;
    }

    public void run() {
        int tid = getNextId();
        try {
            Vector scopes = new Vector();
            scopes.add("default");
            ServiceLocationEnumeration srv = locator.findServices(serviceType,
                                                                  scopes, "");

            PrintWriter file = new PrintWriter(new FileWriter("thread-"
                                                              +tid+".txt"));
            while(srv.hasMoreElements()) {
                file.println(srv.nextElement().toString());
            }
            file.close();
        }catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Appendix D

PBS System

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/services/comm/slp/components.sf"
#include "pbsNode.sf"
#include "pbsServer.sf"

/*
Starts a pbs server and a number of nodes.
Some nodes are registered with the server on startup, while others are
advertised through SLP. The advertised nodes can then be registered with
the server at a later time...

The components will read the pbs configuration found in
pbsConfig.sf
*/

sfConfig extends Compound {
    // server
    server extends pbsServer;

    // registered nodes...
    node1 extends pbsNode {
        sfProcessHost "hostname1";
        sendTo:foo LAZY server;
    }

    node2 extends pbsNode {
        sfProcessHost "hostname2";
        sendTo:foo LAZY server;
    }

    // advertised nodes
    node3 extends pbsAdvertisedNode {
        sfProcessHost "hostname3";
    }

    node4 extends pbsAdvertisedNode {
        sfProcessHost "hostname4";
    }
}
```

Appendix E

SLP User Guide

This is the user guide for the SLP library. A copy of this document can also be found on the attached CD-ROM under the name slpManual.pdf.

E.1 Introduction

This aim of this manual is to show how the Service Location Protocol library can be used. It is split into two parts: SLP API classes and SLP SmartFrog components. The API classes implements the standard Java API for SLP as defined in RFC-2614. These can be used to add SLP support to any Java program. The SLP SmartFrog components are special SmartFrog components that allow the SLP library to be used easily within a SmartFrog application.

The Service Location Protocol is a protocol for advertising and locating services over a network. The main parts of the protocol are the Service Agent (Advertiser) and the User Agent (Locator). The Advertiser is responsible for advertising the existence of one or more services. The Locator is used to search for advertised services in the network.

E.2 SLP API Classes

This section explains how to use the features of the SLP library directly within your own source code, without going through the SmartFrog components. In fact, you don't have to be running SmartFrog at all. The API presented follows the standard suggested in RFC-2614, so it may already be familiar. This document will go through the typical usage of SLP in a program. More details on the available classes and methods can be found in the javadoc files. The important classes for using the library are in the org.smartfrog.services.comm.slp package.

E.2.1 ServiceLocationManager

The Service Location Manager manages the access to the SLP framework. One can use this to obtain an advertiser or locator for use in a program. Multiple programs within the same JVM can share a locator or advertiser object. The Service Location Manager will only create a new locator or advertiser if none of the existing ones have the correct properties. The following methods are useful for dealing with the service location manager:

- **static Vector findScopes()**
  Returns the names of all scopes known by the Locators and Advertisers created by this service location manager.
- **static Advertiser getAdvertiser(Locale loc)**
  Returns an advertiser object for the specified language. This can be used to advertise a service. If an advertiser can not be obtained, an exception is thrown indicating the cause of the failure.
- **static Locator getLocator(Locale loc)**
  Returns a locator object for the specified language. This can be used to discover services. If a locator can not be obtained, an exception is thrown indicating the cause of the failure.
- **static void setProperties(Properties p)**
  Sets the properties to use for Locators and Advertisers returned by future calls to getLocator or getAdvertiser. This method is an extension to the standard, and is only needed if the default settings can not be used. The possible properties are given in Section E.2.4.

E.2.2 Advertiser

The Advertiser (Service Agent) is used for advertising services. The Advertiser interface defines the methods needed in order to register a service to be advertised or deregister a service that is no longer available. As well as the methods for registering

and re-registering services, the interface defines methods to add or delete attributes for a registered services. These methods are not implemented in the current version of this library. Using them will result in a ServiceLocationException being thrown with the error code set to NOT_IMPLEMENTED.

- **void register(ServiceURL url, Vector attributes)**
  Registers a service and starts advertising it. The URL is a standard service URL as defined in RFC-2608. The URL is typically of the form "service:my_type://location". The attributes is a Vector of ServiceLocationAttribute objects. See also the javadoc files for information on how the ServiceURL and ServiceLocationAttribute classes work. If the registration is not successful, a ServiceLocationException is thrown indication the cause of the failure.

- **void deregister(ServiceURL url)**
  Deregisters a service. After this, the service can no longer be found using SLP discovery. The service is deregistered in every scope and language it was registered in. If the operation fails, an exception is thrown.

E.2.3 Locator

The Locator (User Agent) is used for locating advertised services. The Locator interface has methods for finding services, service types and service attributes. For each type of discovery, a ServiceLocationEnumeration object holding all results of the request is returned. The standard Java Enumeration methods can be used to get the results from the enumeration.

- **ServiceLocationEnumeration findServices(ServiceType serviceType, Vector scopes, String searchFilter)**
  Tries to find the services with the given service type. The scope vector allows you to limit the search to a given set of scopes. To further limit the number of results, a search filter can be given. The search filter specifies attributes and their values that needs to be present for the discovered service. The filter is an LDAPv3 search filter. The following operators are supported by the current implementation:

  - | - or operator. At least one of the given attributes need to be present with the correct value.
  - & - and operator. All given attributes must be present with the correct value.
  - = - equals operator. At least one of the values for the given attribute must equal the value given in the search filter.
  - $<$ - less than operator. At least one of the values for the given attribute must be less than, or equal to the value given in the search filter.

  - > greater than operator. At least one of the values for the given attribute must be greater than, or equal to the value given in the search filter.

  A filter may look like: (&(attr1=3)(attr4<8)).
  The ServiceLocationEnumeration returned by this operation contains zero or more ServiceURL objects giving the location of the discovered services. If the operation fails, an exception is thrown.

- **ServiceLocationEnumeration findServiceTypes(String namingAuthority, Vector scopes)**
  Finds all the service types available in the given scope. The namingAuthority string may be the name of a naming authority, an empty string or null. If a naming authority is given, only service types registered with that naming authority are returned. If an empty string is provided, only service types registered with the default naming authority are returned. Providing a null pointer will result in all available service types being returned. The ServiceLocationEnumeration returned by the operation contains zero or more ServiceType objects. One for each service type found. If the operation fails, an exception is thrown.

- **ServiceLocationEnumeration findAttributes(ServiceURL URL, Vector scopes, Vector attributeIds)**
  Finds all attributes registered with the given service. The attributeIds vector can be used to only return a subset of the registered attributes. If not empty, only attributes given in the attributeIds vector are returned. The returned ServiceLocationEnumeration contains one ServiceLocationAttribute object for each discovered attribute. If the operation fails, an exception is thrown.

- **ServiceLocationEnumeration findAttributes(ServiceType serviceType, Vector scopes, Vector attributeIds) throws ServiceLocationException**
  This is the same as the above method except that it will find the attributes of all services of the given type.

E.2.4 Configurable Properties

The following properties can be set for the SLP agents (UA, SA, DA). Normally a Java Property object containing the desired properties are created and passed to the service location manager before obtaining the Advertiser or Locator. The DA, as a standalone application, is currently not configurable. If the DA is started as a SmartFrog component, it will get its properties from the SmartFrog description in the same way as any other agent.

- **net.slp.multicastMaximumWait**
  Sets the time to wait for replies to a multicast request. During this time, the original request may be resent a number of times. Default is 15 seconds

- **net.slp.randomWaitBound**
  Sets the maximum value for the various random waits used in the library. Default is 1 second.

- **net.slp.initialTimeout**
  The time before a request is retransmitted for the first time. Default is 2 seconds. The time to wait is doubled each time a message is retransmitted.

- **net.slp.unicastMaximumWait**
  Maximum time to wait for a reply to a unicast request. During this time the message can be resent a number of times. Default is 15 seconds.

- **net.slp.DAHeartBeat**
  The time between each multicast DAAdvert sent by the Directory Agent to advertise its existence. Default is 3 hours.

- **net.slp.DAActiveDiscoveryInterval**
  The time between each attempt by a User Agent or Service Agent to actively discover a Directory Agent. The default is 900 seconds.

- **net.slp.useScopes**
  Sets the scopes to use by the agent. The scopes are separated with a comma character. Default is ”default”.

- **net.slp.DAAddresses**
  Comma-separated list of IP-addresses or hostnames giving the location of Directory Agents. This is useful if a DA can not be discovered by multicast. The default is ”” (no predefined DA).

- **net.slp.passiveDADetection**
  A boolean enabling or disabling passive DA detection. If disabled, the UAs and SAs will not listen for DAAdverts on the multicast address. Default is true.

- **net.slp.MTU**
  Sets the MTU for SLP messages (maximum message size). Default is 1400 bytes.

- **net.slp.port**
  Sets the default port for SLP messages. All requests are sent to this port. The default is 427.

- **net.slp.uaport**
  The port used for sending messages from the UA. When set to 0, the system will select a free port. If, for some reason, a special port has to be used then set this attribute. Default is 0.

- **net.slp.saport**
  The port used for sending messages from the SA. This is also the port on which the SA will be able to receive TCP requests. Default is 0 (system selects).

- **net.slp.locale**
  The language supported by the agent. Each UA/SA supports one language. To register a service in multiple languages, one Advertiser for each language is required. This attribute is set automatically by the service location manager when the getAdvertiser and getLocator methods are called, as these takes the language as a parameter. The default language is English (en).

- **net.slp.multicastAddress**
  The multicast address used for SLP messages. Default is 239.255.255.253.

- **net.slp.interface**
  The IP-address of the network interface to use for SLP messages. Use this if a host has more than one network interface, and you do not want to use the default.

- **net.slp.debug**
  Enable/disable debug output from the SLP library. Default is false.

- **net.slp.logErrors**
  Enable/Disable logging of errors in the SLP library. Default is false.

- **net.slp.logMsg**
  Enable/disable logging of messages sent and received in the SLP library. Default is false.

- **net.slp.logfile**
  The name of the file to write debug, errors and message logs to. If an empty string, all output goes to stdout. Default is "" (stdout).

E.3 SLP SmartFrog Components

This section will explain the use of the SmartFrog components for SLP and the special SmartFrog deployer class that uses SLP to find a process compound in which to deploy a component. There are two components, the SFSlpAdvertiser and the SFSlpLocator. The deployer class is the class SFSlpDeployerImpl found in the org.smartfrog.services.comm.slp package. The next sections will show how the SLP components can be used in SmartFrog.

E.3.1 SFSlpAdvertiser

The SFSlpAdvertiser component is used to advertise things through SLP. The ”toAdvertise” attribute points to what you want to advertise. The ”toAdvertise” attribute can be a reference to another attribute, including other components, or it can be just a simple value. You also need to set some other attributes in order to tell the component which type of service it is advertising, and how long the advertisement should last. The important attributes are given in the list below. In addition, it is possible to override the default settings for the SLP library. The attributes defining the SLP configuration are given in Section E.3.4.

- **serviceType**
  Sets the service type for the advertised service. The service type must be in the format specified in RFC-2608. An example is ”service:sf-prim:printer”. This could be used to advertise the Printer component in the Hello World example provided with SmartFrog. This attribute must be given. Failing to provide this attribute will make the component fail in sfDeploy().

- **serviceAttributes**
  This attribute should be set to a Vector of attributes that are valid for the advertised service. An attribute is represented by a Vector where the first element is the name of the attribute and the remaining elements are the values for that attribute. The default is an empty Vector. Example:
  serviceAttribute [[”a1Name”, ”a1Value1”, ”a1Value2”], [”a2Name”, ”a2Value2”]];

- **serviceLifetime**
  The life time of the service. That is, for how long should the service be advertised. This is given as a positive Integer giving the number of seconds for which the service is advertised. Alternatively, a life time of -1 can be given. This makes the advertisement permanent. I.e. it is advertised as long as the advertiser component is running. The default is -1.

- **toAdvertise**
  Gives the thing to advertise. This can be a simple value or a reference to another attribute. The example below shows how the SFSlpAdvertiser component

is used to advertise the Printer component in the Hello World example. This attribute must be given. Failing to provide this attribute will cause the component to fail in sfDeploy().

- **advertiseReference**
  When the toAdvertise attribute is a Reference, like in the example below, setting this attribute to ”true” will cause the Reference to be advertised instead of the resolution of the Reference. The default is ”false”.

Example of how the SLP advertiser can be used to advertise another component is given in Figure E.1.

```
#include "org/smartfrog/examples/helloworld/printer.sf"
#include "org/smartfrog/services/comm/slp/components.sf"

sfConfig extends Compound {
    p extends Printer;
    adv extends SFSlpAdvertiser {
        serviceType "service:sf-prim:printer";
        toAdvertise LAZY p;
    }
}
```

Figure E.1: Advertising a Component using SLP

E.3.2 SFSlpLocator

The SFSlpLocator component is used to locate advertised services. Other component use a reference to the ”result” attribute of the SFSlpLocator in order to get the result of the search. To be able to search, the component needs to know which type of service to look for. It may also be given a search filter in order to make sure certain attributes are present in the discovered service. The important attributes are given below. As for the advertiser component, the default SLP configuration can be overridden.

- **serviceType**
  The service type for the service to discover. This attribute has to be given, or the component will fail on startup. The service type should match the service type of an advertised service.

- **searchFilter**
  A String giving a search filter to use for the discovery. This search filter must be given in the format shown in Section E.2.3. Specifying this attribute is optional. The default is to use an empty String.

- **discoveryInterval**
  The search done by the component may be repeated periodically. This attribute sets the time (in milliseconds) between each search. A value of zero, means that the search is only performed once, and not repeated.

- **discoveryDelay**
  Sets the delay (in milliseconds) before the first attempt at finding the service. This is to allow the user agent to have time to discover directory agents before it starts searching for services. The default value is zero.

- **returnEnumeration**
  This is a boolean attribute controlling what is returned from the locator when the result attribute is requested. The default is to return the first of the discovered Objects. If this attribute is set to ”true”, the unmodified ServiceLocationEnumeration object is returned. The component requesting the service must then be able to get the information it needs from this. The default is ”false”.

- **result**
  This is not an attribute that can be set by the user. It is the attribute name one would use in references to obtain the result of the service discovery. The value of the attribute may differ for each time sfResolve is called, as new services may be found.

Figure E.2 shows how the SFSlpLocator can be used to locate another component.

```
#include "org/smartfrog/examples/helloworld/generator.sf"
#include "org/smartfrog/services/comm/slp/components.sf"

sfConfig extends Compound {
    g extends Generator {
        printer LAZY loc:result;
    }
    loc extends SFSlpLocator {
        serviceType "service:sf-prim:printer";
    }
}
```

Figure E.2: Locating a Component using SLP

E.3.3 SFSlpDeployerImpl

The SLP deployer class, org.smartfrog.services.comm.slp.SFSlpDeployerImpl, is used to deploy SmartFrog components into a process compound advertised using SLP. If no

PC is found during discovery, the standard deployer class is used instead. A process is advertised by setting the toAdvertise attribute in the advertiser to be a reference to sfProcess. An example is given in Figure E.3.

```
sfConfig extends Compound {
    adv extends SFSlpAdvertiser {
        serviceType "service:sf-pc:whatever";
        toAdvertise LAZY sfProcess;
    }
}
```

Figure E.3: Advertising a SmartFrog Process

In order to use SLP, the deployer class needs to know the service type to look for. It may also need some special SLP configuration. This is done by adding the attribute slpConfig to the description. This should be a component description having at least one attribute. The serviceType attribute. Figure E.4 shows a simple example.

```
sfConfig extends Compound {
    sfDeployerClass "org.smartfrog.services.comm.slp.SFSlpDeployerImpl";
    slpConfig extends SFSlpConfiguration {
        serviceType "service:sf-pc";
    }
    ...
}
```

Figure E.4: Using the SFSlpDeployerImpl Class

E.3.4 SmartFrog SLP Configuration

This section gives an overview of the possible SmartFrog attributes for configuring the SLP agents used by the SmartFrog components. See also Section E.2.4 for an explanation on each of the properties. The default values for these attributes are in the org/smartfrog/services/comm/slp/sf/SFSlpConfiguration.sf file.

- **slp_config_mc_max** Sets the vaule for the net.slp.multicastMaximumWait property.
- **slp_config_rnd_wait** Sets the value for the net.slp.randomWaitBound property.
- **slp_config_retry** Sets the value for the net.slp.initialTimeout property.
- **slp_config_retry_max** Sets the vaue for the net.slp.unicastMaximumWait property.
- **slp_config_da_beat** Sets the value for the net.slp.DAHeartBeat property.
- **slp_config_da_find** Sets the value for the net.slp.DAActiveDiscoveryInterval property.
- **slp_config_daAddresses** Sets the value for the net.slp.DAAddresses property.
- **slp_config_scope_list** Sets the value for the net.slp.useScopes property.
- **slp_config_mtu** Sets the value for the net.slp.MTU property.
- **slp_config_port** Sets the value for the net.slp.port property.
- **slp_config_locale** Sets the value for the net.slp.locale property.
- **slp_config_mc_addr** Sets the value for the net.slp.multicastAddress property.
- **slp_config_interface** Sets the value for the net.slp.interface property.
- **slp_config_debug** Sets the value for the net.slp.debug property.
- **slp_config_log_errors** Sets the value for the net.slp.logErrors property.
- **slp_config_log_msg** Sets the value for the net.slp.logMsg property.
- **slp_config_logfile** Sets the value for the net.slp.logfile property.