A Service Discovery Framework for a Peer-to-Peer Network

by

Siddhartha Goyal

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

February 10, 2003

Author [illegible]
Department of Electrical Engineering and Computer Science
February 10, 2003

Certified by [illegible]
Dr. Larry Rudolph
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Service Discovery framework for a Peer-to-Peer Network

by

Siddhartha Goyal

Submitted to the
Department of Electrical Engineering and Computer Science

February 10, 2003

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

As mobile networks become ubiquitous, and devices like PDAs and cell phones become commonplace, today's traditional computing paradigms are no longer relevant. Instead, we now live in a world where computing power is pervasive, and the means to access that power is as simple as the click of a button or a voice command. However, in order to make use of that power users need a means to locate it. Specifically, users need a means to easily query for services within a network without having to necessarily know where those services are located or even how to query for those services beforehand. The purpose of this thesis is to present a service discovery framework for an ad-hoc network of peers that allows users to discover in realtime *how* to query for a particular service and particular instances of a service using the query mechanisms presented to them.

Thesis Supervisor: Dr. Larry Rudolph
Title: Principle Research Scientist

Acknowledgments

First off I would like to give a hearty thanks to my supervisor, Dr. Larry Rudolph, for guiding me through this thesis and helping me navigate the pitfalls associated with such a project.

I would also like to thank Quinton Zondervan for his vision and support as this thesis progressed.

I would like to thank all my friends who were there for me during these times as I struggled to make this thesis a reality.

Finally I would like to thank my parents, Arvind and Nalini Goyal, and my sister, Vandana Goyal for their love, support, and encouragement not only during the writing of this thesis, but also throughout my life. I wouldn't be where I am today without the three of you!

Contents

LIST OF FIGURES

Chapter 1 Introduction

Imagine a world where we could walk into a city with only a cell phone, and use that cell phone to locate things like ATMs that satisfy our withdrawal needs, a cab that will take us to a certain location, or a restaurant that serves a particular type of food. While this vision may seem to be a pipe dream, all the technologies needed to make such a vision a reality exist today. Wireless networks are powerful enough to allow for the transfer of data almost anywhere, and p2p protocols, like the JXTA protocol suite, make it possible to connect devices together, independent of platform or network topology. In addition, with service discovery protocols, like the one in the INS system, it is even possible to locate things like a restaurant or an ATM using descriptive attributes, instead of IP addresses. What is not possible, however, is the ability to use such discovery techniques without *a priori* knowledge of those techniques.

For instance, if we want to use a cell phone to locate a cab, an ATM, and a restaurant, the cell phone must have some pre-existing knowledge of *how* to query for the cab, ATM, or restaurant. Otherwise, the cell phone would have to be upgraded with software that would have the ability to formulate the proper queries for the particular resource or service. While this may be ok if a user wants to use their cell phone to locate a few different types of things, it does not scale well. What would be ideal, is to have some agent on the cell phone that not only would have the ability to use some general framework for locating completely different services, but also have the ability to *learn* new means of querying for a service. Furthermore, it would be ideal for this agent to have the ability to communicate to other agents these new mechanisms for locating a service.

Using the service discovery framework presented in this thesis, it is possible to accomplish those goals. The service discovery framework in this thesis provides a generalized framework for querying for any type of service, a framework for describing new methods for querying an existing service, and a means to inform others about those methods for querying for a service.

1.1 The Problem

Many service discovery protocols provide very flexible means for describing a service. Thus, it is possible to use any number of existing protocols to describe complex queries, and allow a person to use their cell phone to query for an ATM, a restaurant, and a cab. However, the common problem that all these protocols have, is that there is no way to *learn* a new means for querying for a service.

For instance, the service discovery protocol that is part of the JINI framework allows service providers to describe themselves in any way they see fit. However, it does not provide a means for consumers to learn about the attributes that describe a service. Thus, in order to locate a new service, it is necessary to use software upgrades to query for that service.

On the other hand, there is the JXTA protocol suite. JXTA provides a means for advertising over the network the existence of any resource or service. Thus, using JXTA, it is possible to locate a new means for querying for a service, using its advertisement mechanism. However, JXTA does not provide an explicit framework for actually querying for a particular service.

What would be ideal is to have the best of both worlds. In other words, we want to have the ability to advertise the existence of new service attributes using something like JXTA, while having the ability to specify complex queries, as is possible in JINI.

1.2 The Solution

The service discovery framework presented in this thesis provides that ideal middle ground. Using the basic concept of JXTA advertisements, it is possible to specify a service discovery framework that allows applications to dynamically locate new means of querying for a service, and also to execute complex queries for a particular service. This is all accomplished through the creation and advertisement of service templates. Service templates are simple XML structures that describe the attributes that fully describe a partic ular service. These templates provide a means for registering new attributes with a particular service, while the JXTA framework provides a means for informing other devices about those attributes, and making use of those attributes to specify complex queries for a service.

1.3 The Roadmap

The rest of this paper is laid out as follows. Chapter 2 describes some related research in this field. Chapter 3 describes the design of the service discovery framework. Chapter 4 describes the implementation of this design, as well as a sample application that makes use of the framework. Chapter 5 concludes this thesis with a summary, and possible directions for future research.

Chapter 2 Background Work

The field of pervasive computing is a fast growing field, with many commercial and research projects ongoing concurrently. Having an understanding of existing research and commercial projects will provide a clear context in which to place the framework presented in this thesis. It should be noted that the first project presented in this chapter was used as a basis for implementing the framework described in the third chapter.

2.1 Project JXTA

Project JXTA started out as a small development team formed by Sun in response to the perceived need for a set of peer-to-peer networking protocols that would allow application developers to write any sort of peer-to-peer application, without having to worry about the details of p2p communication [2].

At its very heart, the JXTA v1.0 specification defines a set of six protocols that are the building blocks of all p2p communication. These protocols are as follows -

- The Peer Discovery Protocol – This protocol allows peers to discover what services other peers offer on a particular network.
- The Peer Resolver Protocol – This protocol allows peers to send and process requests.

- The Rendevous Protocol – This protocol handles the propagation of messages between peers.
- The Peer Information Protocol – This protocol allows peers to obtain status information from other peers in the network.
- The Pipe Binding Protocol – This protocol provides a mechanism to create a virtual communication channel from one peer to another.
- The Endpoint Routing Protocol – This protocol provides routing services from a source peer to a destination peer.

Each JXTA protocol defines a set of XML messages that address one area of p2p networking. The use of XML was a conscious choice from the start because it allows the entire JXTA framework to be language, operating system, and transport independent. Every protocol conversation involves a local peer and a remote peer. The local peer generates messages and sends them to a remote peer, while the remote peer is responsible for receiving and processing those messages.

Figure 2-1 depicts the JXTA protocol stack, and how each protocol makes use of the others to accomplish its tasks. Although protocols such as the Peer Resolver Protocol rely on other protocols in the protocol stack, each protocol is partially independent of others that are at lower levels in the stack. That is because some peers might have pre-configured services built into their operating systems or some other modules that might make the implementation of one the JXTA protocol layers redundant. For instance, if a peer already has a pre-configured set of routing peers that it knows about, then there is no need to implement the Endpoint Routing Protocol. Thus, the protocols at the higher levels in the stack simply need some implementation of the services that are provided by

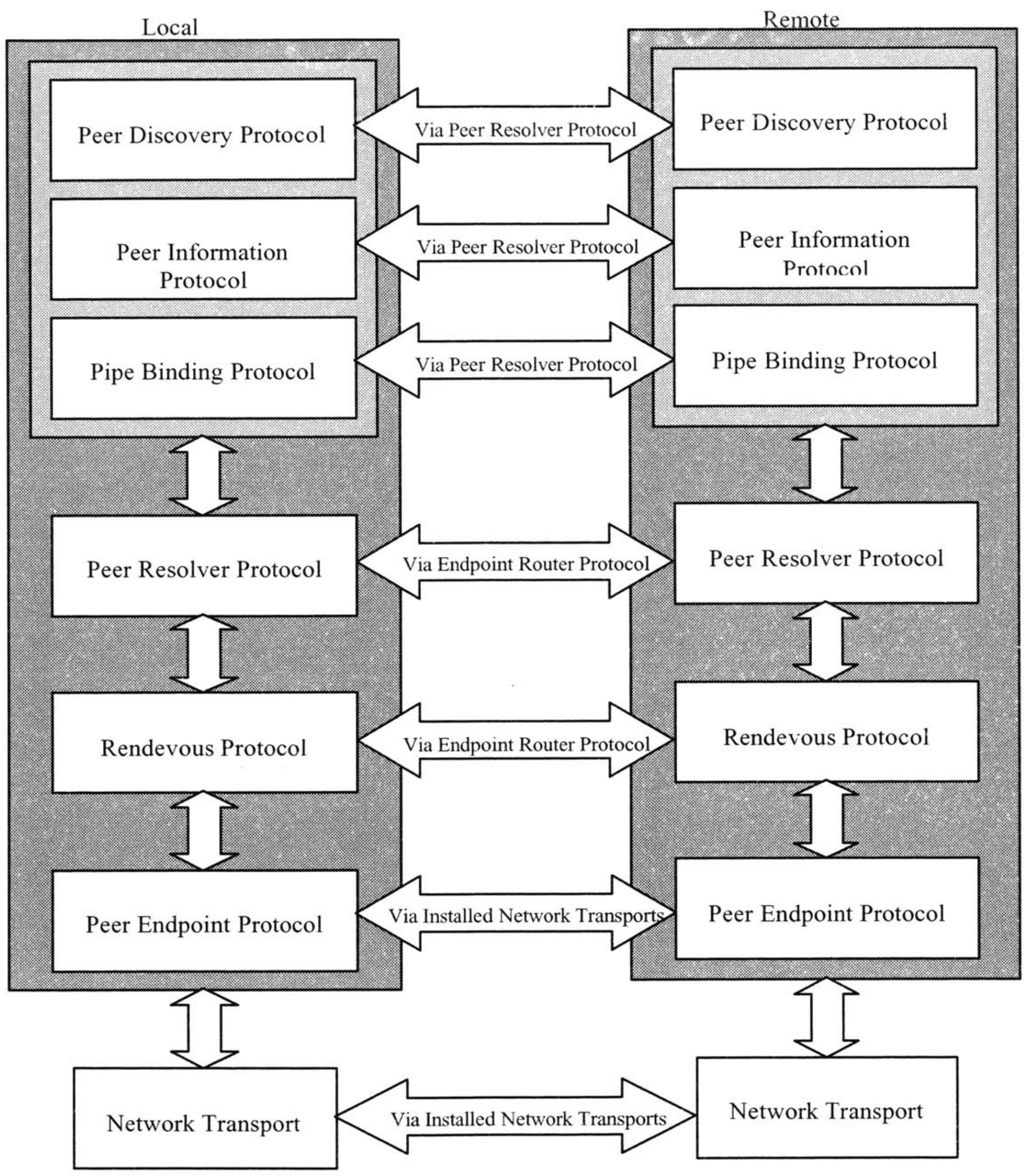


Figure 2-1: The JXTA protocol stack.

the protocols in the lower levels of the stack.

In the context of this thesis and the overall framework discussed in the opening chapter, JXTA provides a framework that allows –

- Peers to form ad-hoc networks without peers having to specify a fixed location.
- Peers to enter and leave the network without the network being disrupted.
- Peers to publish and make use of services within a network.

JXTA also provides a means to discover networks that offer particular services and peers within a particular network. However, this framework is a low-level framework that is based on a simple name-value pair specification. It is desirable that there be a service discovery framework that is more expressive, so as to allow users to specify queries in a more human-like manner.

2.2 INS

The Intentional Naming System, or INS, is a service discovery system designed for mobile and dynamics networks of computers [3]. INS is built on two main concepts, name-specifiers and INRs.

2.2.1 Name-specifiers

Name-specifiers are a hierarchy of attribute-value pairs used by client applications to indicate the intended destination of a request. These specifiers are essentially a replacement for static IP addresses, and allow clients to make requests for services without having to know where exactly those services are located at any given time.

The hierarchy of attribute-value pairs can be easily visualized as a tree, where a node anywhere in the tree (an attribute-value pair) is dependant on the node above it. Thus, a child only has meaning in the context of its parent. Figure 2-2 illustrates an example name-specifier.

```
[city = washington [building = whitehouse
                   [wing = west
                   [room = oval-office]]]]
[service = camera [data-type = picture
                   [format = jpg]]
                  [resolution = 640 x 480]]
[accessibility = public]
```

Figure 2-2: A sample INS name-specifier object.

The nested pairs indicate a containment hierarchy. Therefore, in figure 2-2, the attribute-value pair "building=whitehouse" is a child of "city=washington", and likewise "wing=west" is a child of "building=whitehouse". A single node can also have more than one child, as is the case for the "service=camera" node. This simple fixed structure makes it easy to build standard processing tools for search queries, but is powerful enough to allow any search query to be satisfied.

2.2.2 INRs

In the INS system a client will send a message, like the one in figure 2-2, out onto the network to locate a service with the characteristics specified. These messages are sent to network nodes known as Intentional Name Resolvers, or INRs. INRs act much like DNS servers [8]. They form an "application-level overlay network to exchange service descriptions and construct a local cache based on these advertisements [3]." When an INR receives a message from a client, it does one of two things based on the type of client request. If the early-binding flag is included in the request, the INR simply returns to the client a list of IP addresses where the specified service is located. This is similar to the existing DNS system, and "is useful when services are relatively static [3]." If the client has chosen late-binding, the INR forwards the request directly onto one or more nodes (depending on whether the client has specified intentional anycast or multicast) that match the request query. Late-binding is useful in situations where services are dynamic, and not necessarily fixed at one node.

INRs know the location of various services by listening to advertisements of the existence of those services. Those advertisements are sent periodically over the network on a well-known port that all INRs listen on. When a service advertisement is received, the INR adds the service advertisement to its database of service advertisements, and disseminates that advertisement to other INRs in the network. This brings a degree of fault-tolerance to the network.

In the context of this thesis, the INS system is interesting because of name-specifiers. These simple, yet powerful structures provide a better means for service description and service discovery than the core JXTA service discovery framework provides. In addition, their fixed nature allows applications to use a single service discovery API for searching for any type of service.

2.3 Java JINI and RMI

Sun's Remote Method Invocation, or RMI, was a project initiated to make remote method calls over a network transparent to an application [9]. RMI is built on three main concepts – the RMI registry, service providers, and clients. In the RMI architecture, the registry is a repository for well-known services available over the network. Service providers populate the repository in the following manner. A stub describes each service. The stub is responsible for marshalling calls to the service and unmarshalling responses. This stub, along with instance specific data, is sent to the registry along with a service specified name. When a client is looking for a service, it queries the registry for the service by name, and the stub, along with the instance data, is then returned to client.

The JINI framework builds on top of the RMI framework by providing a more powerful framework for service discovery, and also a framework for distributed events [10]. Whereas in RMI a service is described simply by name, JINI allows services to be described by an arbitrary set of attributes, which are defined by the service itself. Clients looking for a service find the service by querying for specific attributes instead of the service name. The distributed event framework is interesting in the context of this thesis because it breaks the traditional client/server request/response paradigm. This framework allows services to push data to clients, instead of requiring clients to poll for data from services, thus allowing for a peer-to-peer topology.

JINI's service discovery protocol is not as structured as the INS system, and thus allows a service to describe itself in any way it sees fit. Of course, while this gives services considerable flexibility in describing themselves, it makes it difficult to build client applications using a single service discovery framework. In fact, for each service that a client is interested in, it is quite likely that the client will need a specialized module for service queries.

2.4 RDF

The Resource Description Framework, or RDF, is a framework developed by the World Web Consortium to represent "information about resources in the World Wide Web [4]." This information can be about resources that can be retrieved on the web, such as web pages or documents, or about resources that can be identified on the web, such as "items available from online facilities (e.g., information about specifications, prices, and availability), or the description of a Web user's preferences for information delivery [4]." The basic concept behind RDF is that every resource on the web has properties, which in turn have values. Furthermore, it is possible to make statements about those resources, and specify properties and values for those resources. These statements are built using subjects, predicates, and objects. In RDF, the subject of a statement is the part of the sentence that identifies the resource being talked about. The predicate of the statement is the part of the statement that identifies the property of the resource being talked about. The object of the statement is the part of the statement that identifies the value of the property described.

While the main goal of RDF is to provide a uniform framework for resource description, another goal is to also build a machine-processable language. In order for that to happen, it is necessary that the language has a system by which subjects, predicates, and objects can be identified uniquely, and also that the language allows these statements to be exchanged between different platforms. To that end, RDF is built upon URIs [5] and XML [7]. URIs allow for the unique description for anything in a statement, and XML gives a powerful meta-language for representing and exchanging statements. Furthermore, in RDF URIs are used as a means for identifying any resource. Based on this, it is now possible to describe the actual framework. Take the following statement -

```
http://www.example.org/index.html has a creator whose
value is John Smith
```

In RDF the statement is broken down as follows –

- subject - `http://www.example.org/index.html`
- predicate - `http://purl.org/dc/elements/1.1/creator`
- object - `http://www.example.org/staffid/85740`

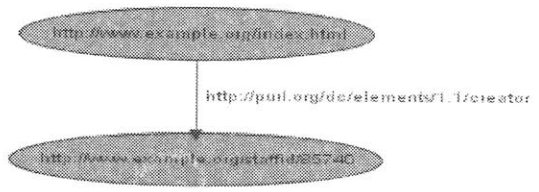

http://www.example.org/index.html
http://purl.org/dc/elements/1.1/creator
http://www.example.org/staffid/85740

Figure 2-3: Graph of a RDF statement.

The string literals “creator” and “John Smith” have been replaced by the URI references, “ `http://purl.org/dc/elements/1.1/creator`” and “`http://www.example.org/staffid/85740`” respectively. The reason for this will be explained shortly, but first the means by which RDF represents statements as graphs of nodes and arcs will be described.

In figure 2-3, the subject is represented by a node, and is labeled with an URIref. The object is also represented by a node, and is labeled with an URIref. The predicate is represented as an arc, also labeled with an URIref, directed from the subject to object. To make additional statements about the same subject all that is required is the addition of more nodes and arcs to the graph. For instance, take the following statements about the same web page.

```
http://www.example.org/index.html has a creation-date whose value is
August 16, 1999
http://www.example.org/index.html has a language whose value is English
```

The previous graph would be expanded, as in figure 2-4, to include two additional arc-node pairs to describe the "creation-date" and "language" properties.
Again the string literals "creation-date" and "language", or predicates, have been replaced by URIrefs. However, the two objects have remained as string literals. Notice that these string literals are enclosed in boxes as opposed to ellipses. In an RDF graph, string literals are enclosed in boxes and URIrefs in ellipses.

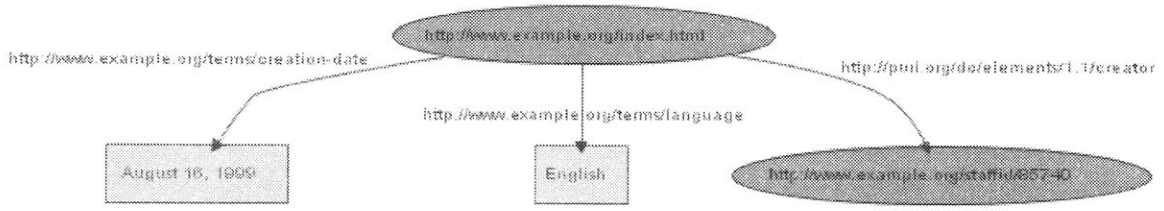

http://www.example.org/index.html
http://www.example.org/terms/creation-date
http://purl.org/dc/elements/1.1/creator
http://www.example.org/terms/language
August 16, 1999
English
http://www.example.org/staffid/85740

Figure 2-4: An RDF graph with string literals as well as URI references.

So why exactly is it advantageous to replace string literals with URIrefs? For one, URIrefs are unambiguous, and allow machine processors to uniquely identify a resource. Secondly, since URIrefs represent resources in RDF, it is possible to build complex statements using URIrefs for object values. For instance, using an URIref for the object "John Smith" allows for the possibility of making additional statements about "John Smith", and for the expansion of the graph in figure 2-4. For a detailed discussion of how this is accomplished see [4].

The graphical representation aids in the understanding of the relationships between various parts of a statement, but obviously it is not possible to send graphs between different platforms to represent these relationships. That is where XML comes in. RDF statements are described as XML documents. Figure 2-5 depicts the XML document representation for the original statement shown earlier. The "Description" element indicates the subject of the statement, with the "about" attribute giving the value of the subject. The "creator" element indicates the predicate of the statement, and its value is the object. In the example in figure 2-5, the value of the element is a text node, which is meant to represent a string literal for the object. However, if the object were an URIref instead, the value of the element would be another "Description" element. In this manner it is possible to build documents describing complex statements.

The value of RDF in the context of this thesis is great because it provides a rich XML based framework for describing resources. Although RDF is meant for describing web resources, the parallels between RDF and the framework described in this thesis are obvious, as the framework presented in this thesis is used for describing and searching for services.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1">

 <rdf:Description rdf:about="http://www.example.org/index.html">
       <dc:creator>John Smith</ex:creator>
 </rdf:Description>
```

Figure 2-5: A RDF statement in XML.

Chapter 3 Design

The purpose of this chapter is to provide a detailed description of the framework that was introduced in the opening chapter. In particular, this chapter will describe how peers get together to form ad-hoc groups that offer a particular service, how peers locate and join those groups, how peers locate services within a particular group, and how a peer publishes its own services to a group.

3.1 Service Advertisement and Discovery

In the previous chapter, many different systems that incorporated service discovery and service publishing mechanisms were described. In addition, one framework was described, RDF, which focused entirely on how to describe meta-information about resources. In each of the systems that offered service discovery and service publication services, nodes within the system could query for services or register services based on the idea of describing a service using attribute-value pairs. In INS, the attribute-value pairs are constrained to a fixed structure, yet it is possible to build powerful naming structures. In JINI, the attribute-value pairs are not constrained to any fixed structure, thus allowing for the description of basically anything. A common thread in each of these systems is that the service provider has the freedom to describe the service in any way he or she sees fit.

While this makes sense, since a service provider knows exactly how to describe itself, it makes it necessary for consumers of these services to have pre-existing knowledge of *how* to query for that same service. This makes it impossible to create a system that is truly dynamic. Every consumer of a particular service has to be pre-configured with the attributes of that service so that the consumer can search for that service. Likewise, every service provider has to be pre-configured with the same set of attributes so that it can advertise its own services.

Thus, in these systems, it is not possible to publish the existence of a service using different attributes on the fly without having a network administrator reconfigure every node in the system to have knowledge of the new service attributes. In addition, if a service is modified in any way, it is likely that the way in which the service is described will change as well. Again in order to make use of the service, a network administrator must intervene to update every node with the new parameters of the service.

What is desirable is a framework that offers service providers and consumers the following capabilities -

- A means for consumers to discover in real time *how* to search for a particular service within the group.
- A means for new service providers within a group to discover in real time *how* to advertise their services.
- A means for service providers within a group to publish new ways of locating a service.

3.1.1 Hailing a cab

To give more motivation as to why these three properties are important, take the example of hailing a cab. When a person hails a cab they often ask for a cab that is close to a certain location. However, this is not the only method of hailing a cab. In some situations a cab may not offer service to the final destination, and therefore it is necessary to look for cabs based on a desired final destination. In yet another situation, suppose that a person has no cash in their wallet, and needs a cab that accepts payment by credit card. Clearly what is evident is that there is no single "right" way to hail a cab. In addition to the methods previously described, there may be other methods of hailing a cab. Because of the difficulty in imagining the *all* the means to hail a cab, it is nearly impossible to build a system that allows users to hail cabs, without requiring there to be some way to update software every time a new means of service discovery is published. If a new means of service discovery were to be introduced into the system, it would require the following actions to be taken by some software provider –

1) The upgrade of all consumers (those peers in the system looking for a cab) with modules to allow them to use the new discovery technique.
2) The upgrade of all service providers (the cabs in this case) with modules that would allow them to advertise their services using the new service attributes.

3.2 PeerGroups

Before discussing the details of how to locate a particular service instance, and how to register templates for discovering a particular service, it is necessary to define the concept of a peergroup. A peergroup is a logical grouping of a number of peers that is based on some common characteristic. This characteristic could be something like a location, a set of services that each peer is interested in, or some platform commonality. For instance, in a home we might create peergroups based on different rooms within the house. There might be a family room peergroup, a bedroom peergroup, and a kitchen peergroup. In this situation, each peergroup would probably offer a number of different services as opposed to a single service. The advantage of the peergroup concept is that it creates a scoping mechanism for interactions amongst peers. This reduces the amount of traffic sent over the network, and it also provides boundaries for service discovery and service registration, as both operations are executed in the context of a peergroup.

3.3 Service Discovery Framework

At a basic level, searching for a service simply involves describing the attributes for a service that we are looking for. However, by examining some common examples searching for a service involves specifying a lot of information. Take the example of looking for a cab that can pick us up at 77 Massachusetts Avenue in Cambridge. When we say we want a cab at a certain location, we are specifying three pieces of information.

1) We are saying that we want a cab service as opposed to some other service.
2) We are saying we want to search for a cab based on a location.
3) We are saying that the location search is based on a street address (as opposed to something like a landmark).

These are the components that make up search queries. We specify the type of service we are looking for, how we wish to search for that service (e.g. location based search), and something specific about that type of search (e.g. a specific street address). In a more general case, it is possible that a search will be more complex, and contain more than one criterion for searching.

Given these components, it is now possible to come up with a framework for describing these queries in some machine-readable format. One such method is through the use of INS name-specifiers. In INS, a search for a cab could be specified as in figure 3-1.

```
[service=cab [searchType=location
        [locationType=streetAddress
         [street = 77 Massachusetts Avenue]
         [city = Cambridge]
 [state=MA]]]]
```

Figure 3-1: An INS name-specifier for a cab service.

XML can be used to describe the same query. Figure 3-2 depicts an XML representation of the same query. While both representations are equally valid for query representation, the framework presented in this thesis uses XML to represent service queries, as well as registration requests, because XML provides a slightly more powerful syntax than INS does. It also allows for queries to be processed by readily available processing tools.

```
<search xmlns="urn:search:peer"
        service="urn:service:cab">
   <query queryType="urn:queryType: proximity"
          dataType="urn:dataType:streetAddress">
      <street>77 Massachusetts Avenue</street>
      <city>Cambridge</city>
      <state>MA</state>
   </query>
</search>
```

Figure 3-2: An XML query for a cab service.

The following sections define each element within a query in more detail.

3.3.1 The "search" element

The "search" element is the container element for an actual query. It has two attributes. The first attribute defines the default namespace for the elements contained within the element. The second attribute, the "service" attribute, defines the name of the service that the query applies to. In general, the "service" attribute takes on the form – urn:service:<serviceType>.

The serviceType defines the actual service. In the previous example, since a cab was the object of the search, the string "cab" is in the serviceType field. If the serviceType were an ATM machine, then the field would have "ATM" in the serviceType field.

3.3.2 The "query" element

The "query" element serves as a container for the actual query predicate. It is always a child of the search element. In general, there can be more than one query element in a search. This indicates that a search is based on multiple query predicates. Each query element has two required attributes.

The first attribute is named "queryType", and defines the specific type of query. The attribute takes on the form of the URN [6]– urn:queryType:<queryType>. The queryType field can take on one of three possible values –

1) proximity – This value means that the peers should be located based on their proximity to the location specified in the query predicate.
2) name – This value means that peers should be located based on their name. The name does not have to be unique, so this could possibly return multiple matches.
3) serviceAttribute – This value means that peers should be located based on a specific service attribute of that peer. If this is specified as the "queryType", then the "query" element must also have an additional attribute that is named "attribute". This attribute defines the name of service attribute. The attribute value is another URN – "urn:attribute:<attrName>". The attrName field is the name of the service attribute. As an example, suppose that a user specified a final destination in their query for a cab. That type of query would be represented by the query shown in figure 3-3.

The second attribute that is required for a "query" element is the "dataType" attribute. The "dataType" attribute defines the data type of the fields contained by the "query" element. This attribute is another URN of the general form – urn:dataType:<dataType>. The dataType field can take on one of the following values –

- string – a simple string data type.
- integer – a simple integer data type.
- time – a compound data type for defining a time.
- date – a compound data type for defining a date.
- streetAddress – a compound data type for defining a street address.

The child elements of the "query" element depend on the value of the "dataType" attribute. In the case of the simple types "string" and "integer", the child elements of the "query" element are either "string" or "integer" elements. In the case of compound data types, there are multiple child elements for each data type.

```
<search xmins="urn:search:peer"
          service="urn:service:cab">
    <query queryType="urn:queryType:serviceAttribute"
            dataType="urn:dataType:streetAddress"
            attribute="urn:attribute:destination">
        <street>77 Massachusetts Avenue</street>
        <city>Cambridge</city>
        <state>MA</state>
        <zip>02139</zip>
    </query>
</search>
```

Figure 3-3: A query based on a service attribute.

3.3.2.1 The “time” data type

In the event that the data type of the query is specified as “time”, then the “query” element has two child elements – “hour” and “minute” – that define the hour and minute of the time.

3.3.2.2 The “date” data type

In the event that the data type of the query is specified as “date”, then the “query” element has three child elements – “day”, “month”, “year” – that define the day, month, and year of the date.

3.3.2.3 The “streetAddress” data type

In the event that the data type of the query is specified as “streetAddress”, then the “query” element has three child elements – “street”, “city”, “state” – that define the street, city, and state of the address.

3.3.3 Query Formulation

A peer formulates a query for a particular service based on service registration templates (section 3.4). Using an API described in chapter 4, agents can generate queries for a particular service instance, without having *a priori* knowledge of how to query for a particular service. In that way, it is possible to have an intelligent agent on peers that can adapt to different situations, without the necessity for constant software upgrades.

3.4 Service Registration Framework

This section describes how to register new service discovery mechanisms within a particular network. In the previous section, the components of a peer discovery message were described –

1) The service we would like to find.
2) The means of querying the service (e.g. through service attributes, proximity, or a name).
3) The actual query predicate.

Service registration messages are a means by which service providers can describe as to how they would like to be queried. Service providers simply send messages that indicate they are willing to accept certain query types. In the cab example, the cab service provider would send a message, like the one in figure 3-4, to register a specific query type on the network. The following sections define each element within a service registration message.

```
<register xmlns="urn:search:peer:">
   <serviceName>cab</serviceName>
   <serviceDesc>This is a cab service.</serviceDesc>
   <queryTemplates>
      <template type="proximity">
         <location dataType="urn:dataType:streetAddress"/>
         <location dataType="urn:dataType:landmark"/>
      </template>
      <template type="serviceAttribute"
               dataType="urn:dataType:streetAddress">
         <attrName>destination</attrName>
         <attrDesc>The destination where we'd like to go</attrDesc>
      </template>
   </queryTemplates>
</register>
```

Figure 3-4: A service registration template.

3.4.1 The "register" element

This element is the container element for the registration message. This element also has a "namespace" attribute, which defines the default namespace for all elements contained within this element.

3.4.2 The "serviceName" and "serviceDesc" elements

The "serviceName" element simply indicates a shorthand name for the service being registered. This is the name that will be used to generate the full URN for the "service" attribute of the "query" element described previously. The "serviceDesc" element provides a longer description of the service. This can be used to describe the service to a user if they are confused about what the service provides.

3.4.3 The “queryTemplates” element

This element serves as a container for each query template that the service is registering. A “queryTemplates” element can contain multiple query templates.

3.4.4 The “template” element

This element is a container for an actual query template, or a set of query templates. This element has one required attribute, named “type”. The attribute can have one of two values –

- proximity – In this case, the template is for searches based on proximity to a certain location. If the “type” attribute has a value of proximity, then the children of the template are “location” elements. Each “location” element has a “dataType” attribute that defines the data type supported for a proximity query. For instance, if the dataType is “urn:dataType:streetAddress”, then the proximity search should be based on a street address. All “location” elements are lumped under a single “template” element, so as to ease in readability and machine processing.
- serviceAttribute – In this case, the template is for searches based on a specific service attribute. If the “type” attribute has this value, then the “template” element must also have a “dataType” attribute, which defines the data type of the

service attribute. The "dataType" attribute has a value that is a URN of the form defined previously (see section 3.3.2). A "template" element of this type must also have two child elements – "attrName" and "attrDesc". The "attrName" defines the name of the service attribute. The "attrDesc" defines a description of the attribute.

3.4.5 Service Registration

Service providers register a service query template using the public API described in the next chapter. When service providers create a template and register it using the public API, the following events occur –

1) The service query template is registered into the service provider's own service table. This table is a container for all the query templates for a particular service. There can be multiple services, and multiple templates for each service in the table. The reason for this is that in a particular network there may be multiple services offered. For instance, in a home network there may be a music service, a movie service, and a video service.
2) The service query template is sent over the network to all other peers in the network. When a peer receives a template it adds it to its own table of service templates, if it already has not done so.

3.5 Intelligent Agents

Using service templates and service queries, it is possible to build intelligent agents that can adapt to different situations. For example, in a cell phone there could be an agent that would have the ability to handle requests for a cab, and an ATM, using the same framework. It would require no software upgrades, and would dynamically adjust to different situations. This is the ultimate vision for this framework, and although this idea is not investigated fully in this thesis, it is a step in the direction of making pervasive computing a reality.

Chapter 4 Implementation Details

This chapter describes some of the implementation details of the framework described in the previous chapter. In addition, a sample application that was implemented using this framework is described.

4.1 JXTA

The implementation of the framework was done on top of the JXTA peer-to-peer connection framework. There are a few reasons for this –

- It provides an infrastructure for propagating messages to peers within a network.
- It provides a service for creating a virtual connection between two peers within a network.
- It provides a framework for creating peergroups.

Sending messages, creating connections between peers, and creating peergroups is all accomplished through the sending and receiving of Advertisements. "Advertisements are one of the basic building blocks of JXTA [1]," and provide a mechanism for describing any resource or service in a JXTA network. When a peer in the JXTA framework is looking for a resource or service, it simply looks for an advertisement describing that resource or service. Advertisements are nothing more than XML documents, and thus

```
<?xml version="1.0"?>

<!DOCTYPE jxta:MIA>

<jxta:MIA xmlns:jxta="http://jxta.org">
   <MSID>
      urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000010306
   </MSID>
   <Comp>
      <Efmt>
         JDK1.4
      </Efmt>
      <Bind>
         V1.0 Ref Impl
      </Bind>
   </Comp>
   <Code>
      net.jxta.impl.peergroup.StdPeerGroup
   </Code>
   <PURI>
      http://www.jxta.org/download/jxta.jar
   </PURI>
   <Prov>
      sun.com
   </Prov>
   <Desc>
      General Purpose Peer Group Implementation
   </Desc>
   <Parm
      <Svc>
         <jxta:MIA xmlns:jxta="http://jxta.org">
            <MSID>
      urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000030106
            </MSID>
            <Code>
      net.jxta.impl.discovery.DiscoveryServiceImpl
            </Code>
            <PURI>
         http://www.jxta.org/download/jxta.jar
            </PURI>
            <Prov>
               sun.com
            </Prov>
            <Desc>
   Reference Implementation of the DiscoveryService service
            </Desc>
         </jxta:MIA>
      </Svc>
   </Parm>
</jxta:MIA>
```

Figure 4-1: A sample JXTA Advertisement.

provide a platform independent representation of any sort of object. Figure 4-1 depicts a sample advertisement document. In the implementation of the service discovery framework, advertisements are used for describing service discovery queries and service registration templates. These two advertisement types are transparent to application developers, as they are used internally to exchange queries and templates. From a development standpoint, all developers need to worry about is how to create queries and query templates.

4.2 Queries and Query Templates

In the previous chapter, two messages types were defined – queries and query templates. Peers can send queries to other peers to locate an instance of a particular service, and peers can also send, or register, query templates within a peergroup, which describe how to formulate queries for a particular service. Three simple classes define these two messages.

The ServiceQuery class is the base class for all queries and query templates. It contains the following pieces of information –

- The type of the query or query template. Class instances can either represent proximity or service attribute query or query templates. Future extensions to this scheme could include a name-based query.
- The data type of the query or query template. For possible data types see section 3.3.2. In this prototype implementation there is support for only the string data type.
- The fields in the query or query template.

The ProximityQuery and AttributeQuery classes represent queries or query templates that are based on proximity and service attributes respectively. These classes are implementations of the abstract base class ServiceQuery. The ProximityQuery class contains no extra fields, as all the information necessary in that query type is contained in the ServiceQuery class. The AttributeQuery class extends the ServiceQuery class by adding attribute name and attribute description fields.

4.3 QueryReader and QueryWriter classes

The QueryReader and QueryWriter classes are used for writing and reading ServiceQuery objects to and from XML streams. They operate as shown in figure 4-2.

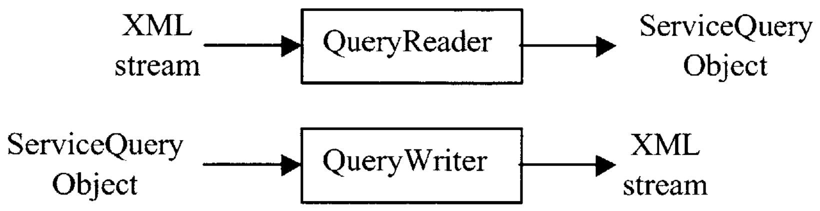


Figure 4-2: QueryReader and QueryWriter.

These two classes are used inside the framework when a query needs to be written or read from a stream. An application developer does not make use of these classes to develop agents or applications that make use of the framework.

4.4 ServiceTemplate class

The ServiceTemplate class contains the following pieces of information –

- The name of the service that the query templates are meant for.
- The description of the service that the query templates are meant for.
- One or more query templates that indicate how a user can query for the particular service.

Query templates are simply represented by ServiceQuery instances. The only difference to a developer between a query and a query template is the context the ServiceQuery is used in. When a ServiceQuery instance is contained within a ServiceTemplate object, the field values for that ServiceQuery instance are ignored by other components in the system.

4.5 TemplateReader and TemplateWriter classes

The TemplateReader and TemplateWriter classes are used much like the QueryReader and QueryWriter classes. They are used to read and write ServiceTemplates to and from XML streams. They operate as shown in figure 4-3.

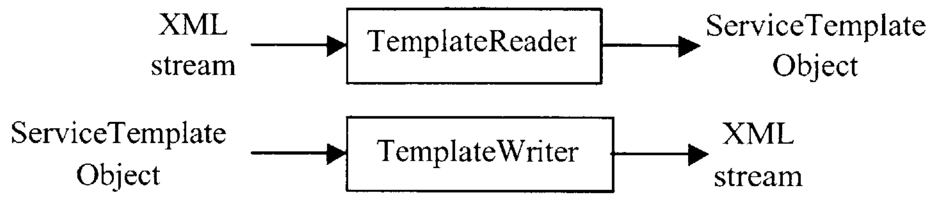


Figure 4-3: TemplateReader and TemplateWriter.

Again these two classes are internal classes, and are not used by application developers directly. The framework makes use of these classes when reading and writing templates.

4.6 ServiceTable class

This class is used to store the names of the services that are registered with a particular peer, and the methods for querying for those services. This table is maintained by the internal system. Whenever a new service template is discovered through the JXTA framework, the table is updated with the new service template. The table has a public API that allows applications to locate service names that are discovered, and also to locate the queries that can be used to locate service instances. Abstractly, the table can be pictured as in figure 4-4.

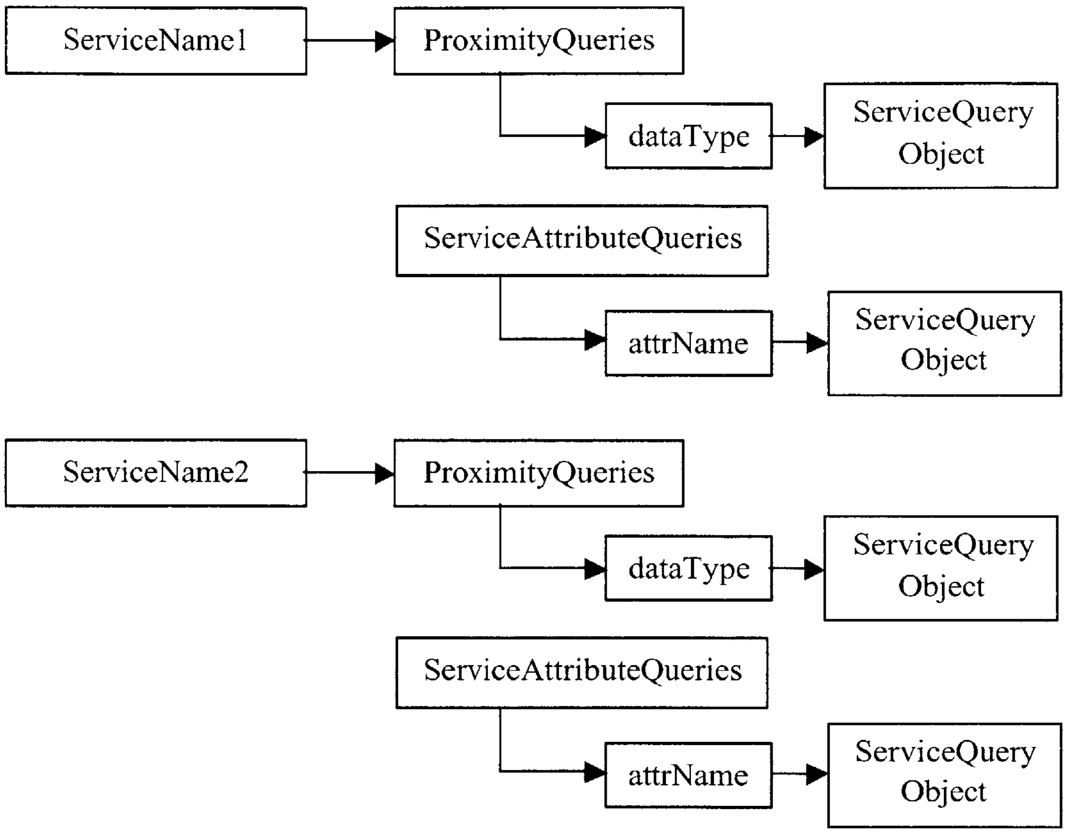


Figure 4-4: A depiction of a ServiceTable object.

4.7 PeerGroupConnection

This class is the main class that all applications use to locate peergroups, join a peergroup, send service queries, and register new templates. It contains a simple API that allows for all of these operations to occur. Two methods in this class are asynchronous. The first is the *locatePeerGroup()* method. Applications must register a listener with this method that receives notifications when new peergroups are found. What an application does when a new peergroup is found is beyond the scope of the framework. It can display a notification to the user, update a list that shows peergroups, etc. The second method is the *getServiceTemplates()* method. Applications should call this method after they join a peergroup. This method also requires a listener that is notified when new service templates are located. The default listener is the ServiceTable class.

4.8 Public API

The public API for developers to develop applications that make use of the search framework include the ServiceQuery, ServiceTemplate, and ServiceTable classes. The ServiceQuery is used for creating search queries and search templates for a particular service. The ServiceTemplate class is used to register a set of ServiceQuery objects with a particular service. The ServiceTable class is used to locate the type of queries that a particular service accepts. Finally, the PeerGroupConnection class is used to locate a peergroup, connect to a peergroup, and to make use of the search framework.

4.9 Sample Application

One scenario where the service discovery framework presented is useful is in a city, where users often need to locate something like a cab, a parking meter, or an ATM. While we normally want to locate these things using a location-based query, there are times when we want to add more attributes to our query. For instance, when we are looking for an ATM, we don't just look for an ATM that is nearest to us, but we also may want to locate an ATM that belongs to a particular financial institution, accepts deposits, or is capable of dispensing stamps. This type of search is perfect for the service discovery framework, as it allows users to specify queries based on attributes.

To demonstrate the viability of this service discovery framework, a simple ATM locator application was built. This application allows users to locate ATMs based on various attributes. The application demonstrates the major pieces of functionality that are present in the service discovery framework –

- PeerGroup Creation – While it would be ideal for every ATM, regardless of financial institution, to belong to one single network, it is likely that each institution would have separate networks of ATMs. In the context of this framework, this equates to creating peer groups of ATMs. In the application, when an ATM comes online, it joins an existing group of ATMs, or creates a new group if it cannot locate the group it is looking for.
- Query Template Registration – Each ATM network may have different service characteristics, and when a user is searching for an ATM in a particular network, they need the ability to figure out dynamically how to query for an ATM. The application allows ATM peers to register query templates within the network.
- Query Dissemination – Once a user has created a query, the JXTA framework disseminates that query throughout the network. ATMs that have attributes that match the service request respond to the peer using a published pipe.

The following sections describe in detail how the ATM locator application works.

4.9.1 ATM Peers and the Locator Peer

Using the ATM locator application involves running not only the application that is used to query for an ATM, but also running multiple instances of ATM peers. ATM peers are the peers that are capable of creating peer groups, registering service templates within a peer group, and responding to requests from the ATM locator application. The ATM locator application is the application that is capable of joining a specific ATM peer group, and querying for an ATM using the attributes described in the service templates advertised within that peer group.

4.9.2 ATM Peers

Bringing an ATM online involves a three-step process. The first step in the process involves the ATM peer reading an XML file that specifies the attributes used to describe the particular ATM, and also the peer group to which the ATM belongs to. This XML file is a service template file that the ATM also publishes within the peer group. Figure 4-5 shows the service template for describing ATMs within a peer group of FleetBank ATMs.

The second part of the initialization process involves using the administration depicted in figure 4-6 to specify values for the attributes for the ATM, and also its street location.

The last step in the initialization process involves the ATM joining or creating the peer group specified in its initialization file. The ATM peer creates the peer group if it cannot locate the peer group on the network after ten tries. If it does locate the peer group, it joins that peer group.

```
<register xmlns="urn:search:peer">
    <serviceName>Fleet ATM Network</serviceName>
    <serviceDesc>Network of Fleet ATMs </serviceDesc>
    <queryTemplates>
        <template type="serviceAttribute" dataType="urn:dataType:string">
            <attrName>city</attrName>
            <attrDesc>The city this ATM is located in</attrDesc>
        </template>
        <template type="serviceAttribute" dataType="urn:dataType:string">
            <attrName>dispensesStamps</attrName>
            <attrDesc>Determines whether this ATM dispenses stamps</attrDesc>
        </template>
    </queryTemplates>
</register>
```

Figure 4-5: XML service template for an ATM peer.

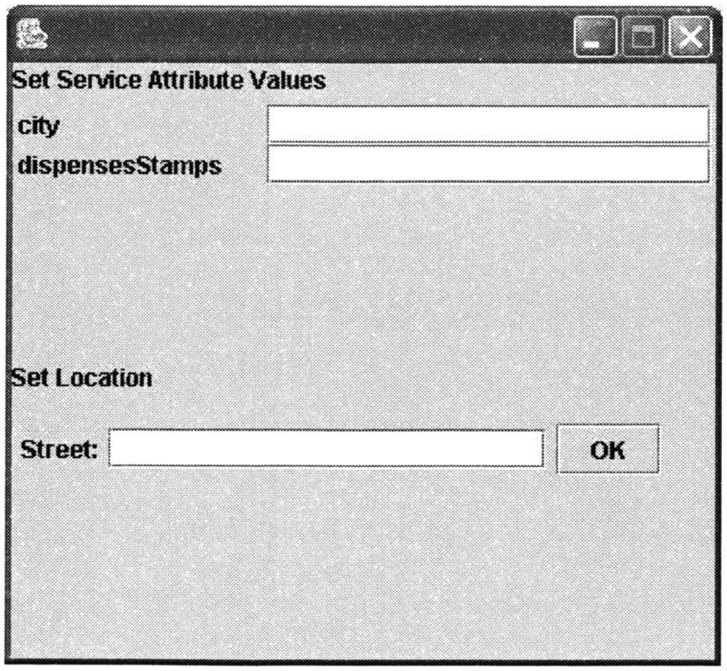


Figure 4-6: ATM initialization screen.

This three-step process brings the ATM online, and has it prepared to process requests. Once online, the ATM listens for queries within the peer group. Upon receiving a query, the ATM processes the query by matching the values of the attributes in the query with its own values for the same attributes. If all the values match, the ATM responds to the peer making the request with its own street address.

4.9.3 The Locator Application

The ATM locator application allows users to join a specific group of ATMs, and query for ATMs within that group. When the application is first started, the user is presented with a screen that allows the user to see the peer groups that are online, and to join a specific peer group. The user must click on the "Locate Groups" button to start the peer group discovery process. When a peer group is discovered, that group is added to the list of groups known to the locator peer. Clicking the "Join Group" button causes the peer to join the highlighted peer group. The second screen in the application is used to display to the user the list of peers that responded to a query request. To make a query for a specific ATM, a user clicks on "Locate ATM providers" button on that screen. To leave the peer group, the user clicks on the "Leave Group" button. The last screen in the application is used to specify the terms of a search query. When a user clicks "OK" button a query of the form specified in chapter 3 is sent out in the peer group. Figures 4-7 through 4-9 show the three screens in the ATM locator application.

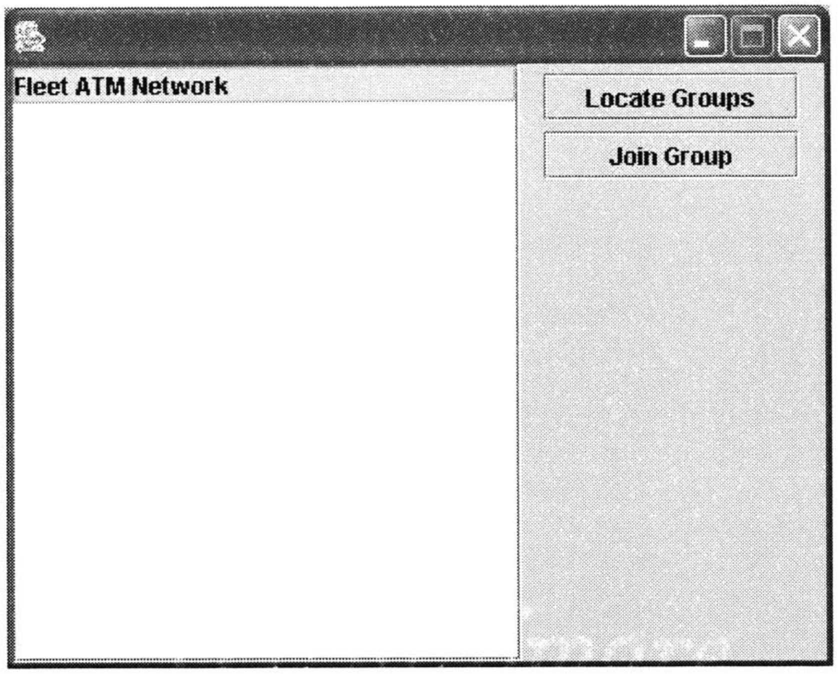


Figure 4-7: The first screen in the ATM locator application.

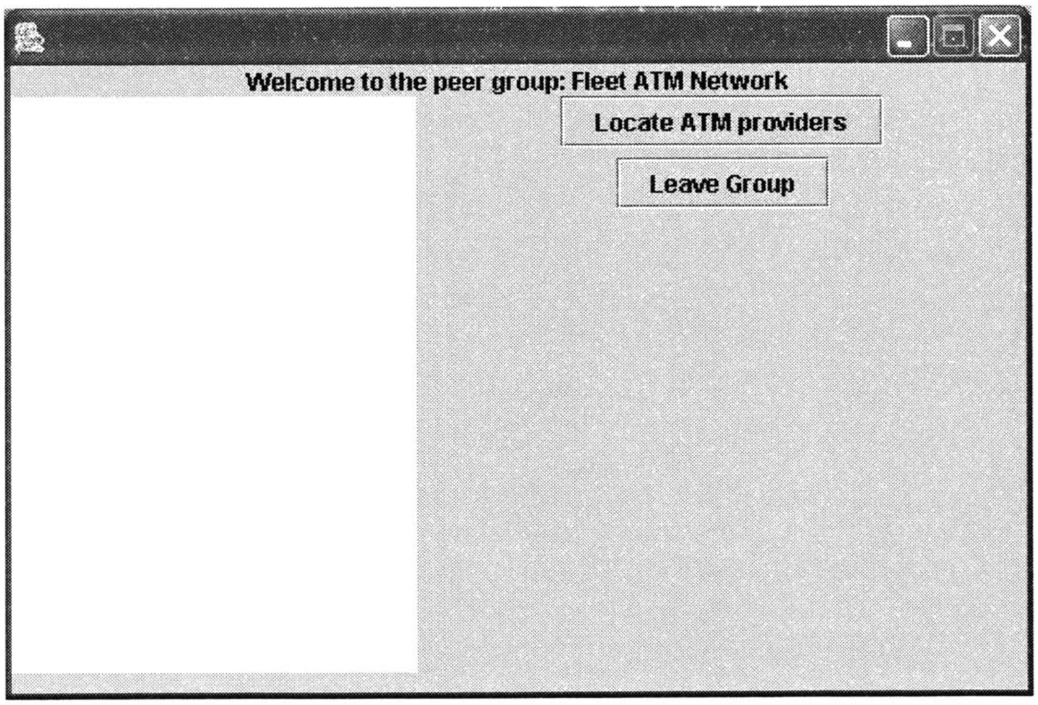
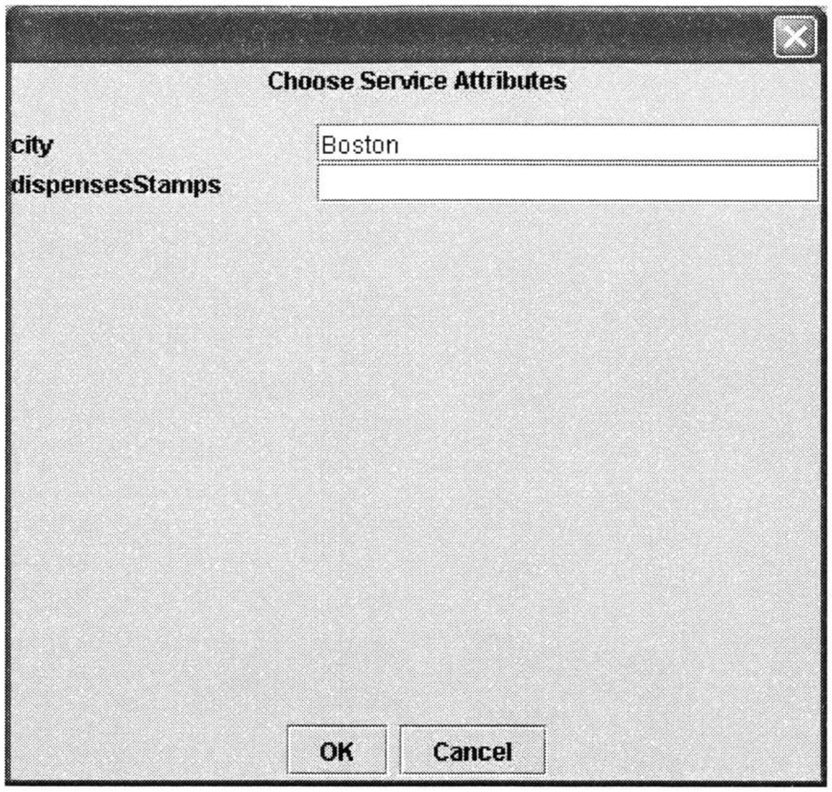


Figure 4-8: The second screen in the ATM locator application.


Choose Service Attributes

city: Boston

dispensesStamps:

Figure 4-9: The third screen in the ATM locator application.

Chapter 5 Conclusion

In the previous chapters, the design and implementation of a framework that allows peers to dynamically register and discover different methods for service discovery was described. In this chapter, some directions for future research in this, and related fields, will be briefly discussed.

5.1 Human Centered Interfaces

The framework proposed in this thesis is a low level framework for service discovery and registration. In the real world, we want to be able to say a query like, "I want a cab at 77 Massachusetts Avenue in Cambridge." Saying this should translate down into the type of queries discussed in the third chapter. In addition, in the event that new types of service discovery templates are located, it should be possible to inform a non-technical user how to use those templates. Ideally, there should be some sort of workflow process that explains to a user how to make use of a new discovery technique. All of this work involves a lot of different disciplines, one of which is natural language processing. Taking sentences spoken in a conversational manner, and breaking them down into pieces that a machine can understand is a big problem. The Spoken Language Systems

Group at MIT's Laboratory for Computer Science is currently conducting research in this area.

5.2 Service Instantiation

Once a specific service, or service instance, is discovered there is the need for a framework for invoking that service. For instance, suppose we use a device equipped with the service discovery framework for finding open parking meters within a certain location. Once we have found a parking meter, we would like to be able to pay that parking meter electronically. In order to do this, we must go through the following process –

1) We must tell the parking meter how long we want to stay for.
2) The parking meter must tell us how much money that will cost.
3) Our device must be able to contact our financial institution to initiate a transaction to pay the parking meter.
4) Our device must send to our financial institution a security certificate that authenticates us as a valid user of the account we are withdrawing funds from
5) Once authenticated our device must send to the financial institution a payment request that includes the amount we are paying, who we are paying, and some service specific parameters. In this case, our device would have to send to the financial institution some piece of information that uniquely identifies the parking meter we are paying for.

6) The financial institution would then send to the parking meter company the payment, and the parameters that tell the company which parking is being paid for.
7) The parking meter company would okay the payment, and our financial institution would send us a confirmation saying that a payment had been made.

While this is a fairly straightforward workflow for this type of application, we must remember that this application is not likely to come pre-installed on every peer. That would be cumbersome, as it would be necessary to install applications for every type of service we would want to make use of. What is desirable is the ability to dynamically load the necessary components on a device, after establishing a connection with the parking meter. It is likely that the device we are using has some component that has the ability to contact our financial institution, but beyond that the device might not have the components to accomplish the negotiation process with the parking meter (i.e. the components that ask a user how much time they would like to stay for, tell a user how much that will cost, and add to the payment slip the parking meter identifier).

In order to support this specific service, and other services, the following questions must be answered –

1) How is a peer able to dynamically load software modules from another peer, and how are those modules intelligently configured to run on the platform they will execute on. For instance, if we are using a cell phone to pay the parking meter, then there may be some speech-based interface to our service. Whereas if we are using something like a PDA, there may be some GUI interface to our service.

2) How is possible to securely conduct transactions in an ad-hoc network of peers? There needs to be some authentication framework that allows one peer to authenticate another. This authentication mechanism must be flexible enough to support multiple authentication types, such as a simple password based system or a more complex certificate based system. In addition, there must be support for data encryption when passing around sensitive data, such as bank account information.

As is evident there are many pieces that must be put together to provide a system that is easy to use and intelligent enough to adapt to different situations. The purpose of this thesis was to provide insight into one of those pieces that will allow us to live in a world where computing is truly pervasive, and as easy to use as the air we breathe.

References

[1] Scott Oaks, Bernard Traversat, Li Gong, "JXTA In a Nutshell," 2002

[2] Brendon J. Wilson, "JXTA," 2002

[3] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilly, "The Design and Implementation of an Intentional Naming System," *17th ACM SOSP*, Dec. 1999

[4] Frank Minola, Eric Miller, Brian McBride, "RDF Primer," http://www.w3.org/TR/rdf-primer/, 2001-2002

[5] T. Berners-Lee, R. Fielding, L. Masinter, "RFC: 2396 Uniform Resource Identifiers (URI): Generic Syntax," http://www.ietf.org/rfc/rfc2396.txt, 1998

[6] R. Moats, "RFC: 2141 URN Syntax", http://www.ietf.org/rfc/rfc2141.txt, May 1997

[7] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, "Extensible Markup Language (XML) 1.0 (Second Edition)," http://www.w3.org/TR/REC-xml, October 2000

[8] P. Mockapetris, "RFC: 1035 DOMAIN NAMES – IMPLEMENTATION AND SPECIFICATION," http://www.ietf.org/rfc/rfc1035.txt, November 1987

[9] Sun Microsystems Inc., "Java Remote Method Invocation Specification," 2002

[10] Sun Microsystems Inc., "Jini Architecture Specification, Version 1.2," 2001