A Framework for Semantic Service Discovery

Xiaowei Yang (MIT LCS, yxw@lcs.mit.edu)

June 4, 2001

Abstract

Locating a network service or device on demand is a challenging task for enabling mobile and pervasive computing. There exists quite a number of network service discovery protocols and architectures. Most existing work uses an attribute-based description scheme to characterize a service. However, the discovery process in existing work is primarily done by bit-by-bit exact matching, type matching, string comparison, or integer comparison for matching a query with a service description. To the best of our knowledge, no existing work allows both a service description with arbitrary complex attribute types and a set of meaningful comparison operations based on the semantics of those attributes. The semantic of an attribute is largely ignored in the discovery process. In this paper, we present a framework for discovering a network service based on the semantics of a service representation.

1 Introduction

Locating a network service [1] or a device on demand is a challenging task for enabling mobile and pervasive computing [31]. Unlike traditional computing environments, the available network services are under constant and unpredictable change while a user with a handheld device is roaming around. Thus, it is difficult to statically configure a device with network services in different computing environments. Moreover, as the number of networked services and devices are increasing rapidly, automatic service discovery also eases the system administrator's task. There exists a variety of network protocols and architectures that enable an application to discover a network service with little manual configuration [2, 18, 28, 10, 8, 9, 32, 13].

Existing work on service discovery shares many similarities. Most of them support an attribute-based discovery as well as a simple name lookup to locate a service. There are usually only a set of primitive attribute types, such as string, integer, and floating point, to characterize a service. Thus, the service discovery process is primarily done by type matching, string comparison or integer comparison. When there are complex attribute types, such as in Jini [13], the service discovery is done by exactly matching attribute values in a query with those in a registration. For representing real world objects such as network services, it is necessary to have more complex data structures to capture richer semantics. Moreover, a user may not be able to specify the exact values of interested attributes. Thus, approximate matchings are desirable. To the best of our knowledge, no existing work allows both a service description with arbitrary complex attribute types and a set of meaningful comparison operations based on the semantics of those attributes. For example, in the existing work, when a user issues a query with an attribute-value pair "location=fifth floor", the query is not able

[1] We define a network service to be a facility that connects to the network and can provide some service based on a client's request. In the rest of the paper, we do not differentiate the term "service" with "device" unless explicitly specified.

to match with a service description that has an attribute-value pair "location=5th floor", though "fifth floor" and "5th floor" are semantic equivalent. Another example would be if a service has an attribute that specifies its 3-dimension GPS location, it is difficult for a client to pose a query with the exact value of the attribute. A client usually wants to find a service that is close to it. The notation of physical proximity is best captured by comparing the distance between two locations, instead of syntactically comparing two location expressions, or comparing the values independently at each dimension.

Our main contributions in this paper are: 1. We articulated the conceptual issues involved in service discovery protocols. Structured service representations are not sufficient to define data semantics. The semantics of data is conveyed by how data are interpreted. In the context of service discovery, the semantics of data that is relevant to the discovery process can be captured by the interpretation of the equivalence relation, approximate equivalence relation, and partial order relation over the data type. 2. Having identified semantics essential to the discovery process, we present a framework for discovering a network service based on the semantics of its representation. The framework can also be applied to discovering general distributed objects.

This paper is organized as follows. In Section 2, we briefly describe the service discovery architecture. In Section 3 and 4, we introduce the unique design concepts and design details in our framework. In Section 5, we sketch our prototype implementation in Java [19, 3] and XML [6]. In Section 6, we present the related work. In Section 7, we summarize our work and address some future directions of this work.

2 Service Discovery Architecture

Most existing service discovery work shares a similar architecture. The architecture in our framework is based on the Service Location Protocol [18, 17, 16, 20]. The unique points in our framework lie in the way services are described, queries are formatted, and queries and descriptions are matched. We keep the structure and the operations defined in SLP.

A detailed introduction to SLP can be found in [33]. Here is a brief overview. There are three components in the system, a User Agent (UA), a Service Agent (SA) and a Directory Agent (DA). The directory agent announces its presence through periodic scoped-multicasting [23] on a well-known channel. A user agent or a service agent discovers the address of the directory server in three ways: 1. passive listening to the service announcement; 2. actively multicasting a discovery message for a directory agent; 3. statically configured with the address of directory agents. If there is no DA present in the local network, SAs and UAs discover each other through multicasting.

When an SA discovers a DA, it registers with the DA by sending a service registration message. When a UA requests a service, it contacts a known DA by sending a service query message. Service descriptions are kept as soft states in a directory agent. Services periodically refresh their registrations in a directory agent to keep the states cached there up-to-state.

DAs are served as a centralized repository to provide weak consistency of distributed states in a system. Though it is possible to federate DAs and scale up the service discovery, we focus on service discovery in a Local Area Network (LAN) in our framework.

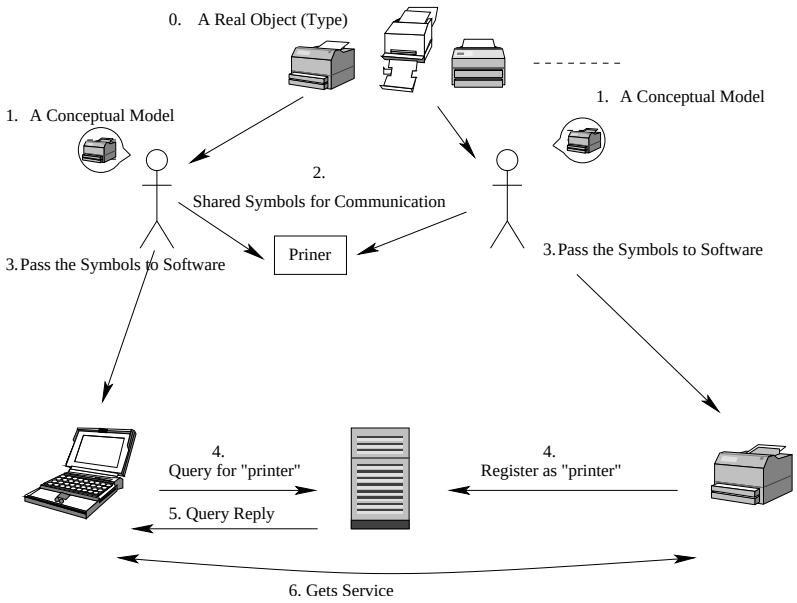

Figure 1: How Ontologies are Shared Between Clients and Services

3 Design Concepts

3.1 Ontological Level Issues

In the mobile and pervasive computing environment, there are two problems needed to be solved. First, how does a cient discover a service. Second, how does a client use a service? For a client to be able to use a service, it expects to get a reply pointing to the "right" service after it poses a query for that service. For example, the client who issues a query for "printer" should get a reply that points to a printer service. This requires that both the service and the client agree on the meaning of "printer". As computers can do nothing but symbol computing, this agreement is actually one between humans who create the query and those who create the service description. Figure 1 shows what happens in this process. First a human forms a conceptual model about a real world object or an object type, which is a set of objects that share common features. Second, to enable communication, shared symbols are created among humans to represent the conceptual model. We call this set of shared symbols with agreed meanings a shared ontology. The ontology reflects the shared conceptual model of the service, which includes what a service is capable of doing (e.g. the functional interface of the service), the terms in which the service is described (e.g., the data types for describing the service) and the meanings of the terms (e.g., what they stand for and what operations are allowed on them). This shared ontology is passed to software through the efforts of programmers or software users. Therefore, the software produced will show behaviors consistent with human's conceptual model. For example, a printer service does the printing job. Thus, the directory agent that does the string (a form of symbol encoding) matching is able to return the desired service to a client. Therefore, for the service discovery to work, we require that in our framework, the client of a service and the service share a common ontology on the service representation. The service representation is ultimately shared between humans who create the service description and who create the query.

In the following section, we discuss the logical structure of a service representation.

3.2 The Logical Structure of a Service Representation

A representation of an object is needed for a client to compose a query to acquire information about the object. In systems such as DNS [25] and [27], an object is represented by a unique identifier. The identifier may optionally contain some information related to the conceptual model of the object, such as the information of organizational hierarchy in DNS.

In most of the existing service discovery architectures, more features of a service are represented by a set of attribute-value pairs. The logical structure of the representation can be viewed as a one-level tree. Figure 2 shows an example. The root node is the service type. The child nodes of the root is the set of attributes. The structure of the tree is determined by the definition of the service type. The types of attributes are primarily primitive types that are understood by a directory agent. In a service registration, a service object is instantiated by supplying values for each attributes. This model is limited because it does not have enough complex data structures to capture arbitrary service models.

In our framework, a service is modeled by an arbitrary-depth tree structure. Figure 3 shows an example. For a complex type such as *GPSLocation*, a simple floating point comparison at each dimension may not make sense. A natural way of using this information is to match it with a service within some distance from the client. Thus, a complex type such as the *GPSLocation* may require a customized matching operation that captures the semantics of the type.

3.3 The Sharing of a Service Representation

In our framework, the service is represented by a complex data structure, which is difficult to share it through memorization. However, to compose a structured query that matches a service representation, the sharing of the representation is necessary. We imagine this sharing can be achieved in the following steps:

1. An organization, for example, a device vendor, publishes the definition of a service type. The definition explains the meanings of data types used in the type definition and operations on how to match or compare them.

2. Programmers who write the service software instantiate a service description according to the service type definition. The service software advertises itself using this description.

3. Programmers who write the client software create structured queries according to the type definition. A user can create a query by using a user interface rendered according to the structure of the service representation.

We expect that different device manufactures or software vendors come up with service definitions in different application domains. Those definitions that reflect the best conceptual models are likely to be widely adopted. Thus, those definitions will become the de facto standards.

3.4 Semantic Aware Query-Representation Matching

As we have mentioned, the existence of a directory agent keeps a weakly consistent view of states in a distributed system. It reduces the network traffic for service discovery. A directory agent is expected to cache a variety of service representations. Thus it is difficult for the directory agent to know the semantics of all data types for heterogeneous service representations. Moreover, it is impossible for the directory agent to predicate what data types will be used in the future. However, as the functionality of a directory agent is to match a query against service representations, unlike clients and services, the

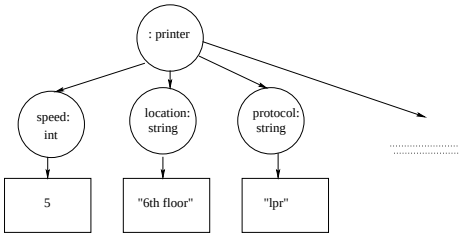

Figure 2: An Example of a Service Representation of Primitive Types

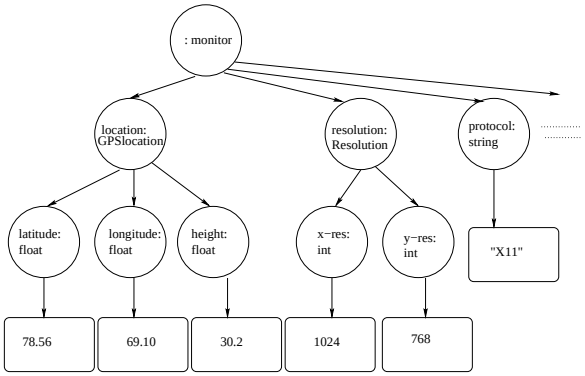

Figure 3: An Example of a Service Representation of Complex Types

directory agent does not have to know the full semantics of the representations. In general, there are some intrinsic data relations associated with a given data type. For example, 60 minutes and an hour are semantic equivalent, though they have quite different syntax. To enable semantic service discovery, a directory agent should understand how to match two values given any data type, which may include:

1. whether two values are equivalent (the equivalence relation);
2. whether two values are approximate equivalent according to a user's criterion (the approximately equivalence relation);
3. whether one value is "less" than another value (the partial order relation).

In the existing infrastructures, a directory agent either knows merely the data relations over primitive data types such as real number, string, boolean, or knows how to do bit-by-bit exact matching or type matching for complicated data types. To solve this problem, we introduce mobile code in our framework. A service representation should also specify the operations for matching or comparing new data types used in its representation. The operations are specified by code. On the service registration, the directory agent uploads the code associated with new data types. When processing a query that contains the new data types, the directory agent calls the code to compare the values in the query and the values in a service registration. Therefore, the directory agent is able to support semantic service discovery for arbitrary data types, including those unknown types when the directory agent software is built.

4 Design Details

4.1 Service Registration

When a service agent discovers a directory agent, it sends a registration message. A service registration provides the following information:

- A unique identifier globally identifying the service object;
- The service type information, which include the URL to download the type definition and the mobile code that implements customized matching operations for new data types;
- The data structure that represents the service object.

A type identifier is quantified within a global unique namespace. Usually, a global unique namespace can be generated from an existing global unique identifier – the URL. For example, the complete type identifier for *GPSLocation* can be *cordelia.lcs.mit.edu/GPSLocation*.

When a directory receives the registration, it updates an internal table that stores the information of matching operations associated with types it has heard of. Table 1 shows an example. In *namespace1*, the data type *GPSLocation* is associated with a DistantMatch operation; in *namespace2*, the data type PersonName is associated with SoundAlikeMatch and SpellAlikeMatch operations.

4.2 Query Format

A query consists of four fields: a Unique Identifier, a service type identifier, a search filter and a reply template. If the client has encountered a service before, it can use the Unique Identifier to get the current information of a service. A reply template contains a data structure similar to the service description, where only fields interested to a client is included and are left empty. A directory agent will fill in those fields with values in a matched service.

A search filter is a logical expression over a set of search records. The logical operators contain "and", "or" and "not". A search record provides the following information:

- the path to a node in a service representation
- the asserted node value for matching
- the matching function to use
- values of other parameters for invoking the matching function, if any

The first field of a record specifies the path to a node in the tree-like data structure of a service representation. The path is relative to the root, which is recognized by the service type. The second field contains the asserted value for the node. It includes all values of fields inside the subtree of the node. The third field specifies which matching function to be applied on the asserted values and the registered values of a service representation. If the third field is empty, a default matching function is applied. In our framework, the default matching function for all types is equivalence matching, which is primarily bit-by-bit exact matching.

An example of a search filter looks like:

(*or* (*and* (**record 1**)(**record 2**))(**record 3**))

4.2.1 The Matching Algorithm

In our design, all matching functions return a boolean value, true or false. The matching algorithm first checks the unique identifier field in a query. If it matches that of a service, it returns the information of the service. If the unique identifier is empty, the algorithm looks at the service type field. If that field is also empty, it returns all services. If the field is not empty, it further examines the registered services of the same type by evaluating the search filter. Each record is evaluated in order. The algorithm applies the specified matching function or a default matching function over the node value in the record, the same node value in a service representation, and parameters in the record (if present). The matching function returns a boolean value. If the values of evaluated records suffice to determine the value of the search filter, no future record evaluation is needed. Otherwise, more records are evaluated until there is no more record left.

5 Implementation

Two choices need to be made when implementing the framework. The first one is the description language for service representation. The second is the mobile code for implementing matching operations. We discuss how to make those choices below.

...	...	...
namespace1	GPSLocation	DistantMatch
	...	...
namespace2	PersonName	SoundAlikeMatch, SpellAlikeMatch
	ShoeSize	UnitConversionMatch
	...	...
...	...	...

Table 1: The table for Data Type → Matching Operations

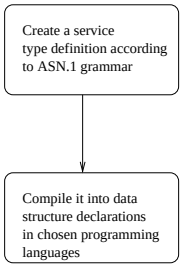

Figure 4: How to Creat a Service Type Definition in ASN.1

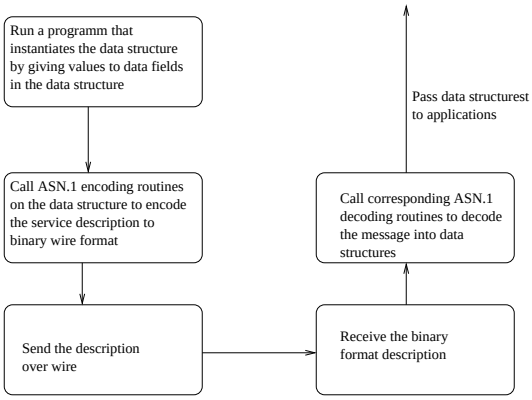

Figure 5: How to Use a Service Description in ASN.1

5.1 Description Language

As services may be implemented in multiple programming languages, we do not want to represent a service in a particular programming language. Therefore, we considered the two popular approaches for representing a tree-like data structure in a programming language independent way: ASN.1 [21] and XML [6], since a service is logically represented by such a data structure in our framework.

Figure 4 and 5 demonstrate the steps for writing a service type definition in ASN.1, instantiating a service object according to the type definition and transferring it to a remote application. Figure 6 and 7 show the steps for doing the same things in XML. As can be seen, it requires less work for using XML as the description language. However, the extra steps in ASN.1 are not useless — they are used for creating binary encodings of service representations, which are much more efficient than the character-based encoding scheme in XML [2]. XML has a verbose grammar. In the basic syntax of XML: <*mark*> *data field* < /*mark*>, the same <*mark*> appears twice just for marking the boundary of a data field. In the resulting document, tiny pieces of data are surrounded by layers of markups, which takes up bandwidth for transferring and hinders readability.

Despite XML's disadvantage, it is a standard, well-defined language and it is flexible and easy to use. Thus, we think it is

[2] To get some idea how efficient ASN.1 is, we have created a simple service representation using both ASN.1 and XML. The Packed Encoding Rule of ASN.1 takes only 19 bytes, while the XML encoding takes 491 bytes.

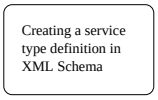

Figure 6: How to Creat a Service Type Definition in XML

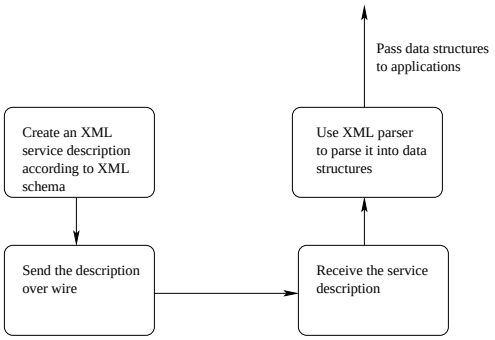

Figure 7: How to Use a Service Description in XML

worth treating simplicity for efficiency. On the other hand, it is desirable to have a human readable format of a service description, an XML document is overall more readable than the binary encoding of ASN.1. Based on the two considerations, we use XML as our service representation language. A better choice would be a customized language that is both human readable and machine parsable but is more succinct than XML.

Figure 8 shows an example of service type representation in XML Schema [14, 26, 5]. In the service type definition, *GPSLocation* and *INETLocation* are taken from another namespace. In the schema, only *Resoluation* is defined. It supports a comparison function "PixelSumGreaterThan", which takes an extra argument "minPixelSum" of positiveInteger type. Figure 9 shows a service instance registration. We have used our own extension to XML Schema.

5.1.1 Mobile Code Platform

The key to semantic service discovery is to enable a directory agent "learn" new data types and their corresponding matching operations at run time. We accomplish this by downloading service type definition and code for matching operations into the directory agent at run time. For a quick prototype implementation, we choose Java byte-code as the mobile code, for its wide availability and flexible security model [15]. To minimize security risks relate to dynamic code downloading in our directory agent, we resort to features provided by Java 2 [19, 3] platform memory and security architecture. When executed in a directory agent, the downloaded matching function is granted with no extra permissions, which means that it can not access any system resources, include accessing or creating new files in the file system, accessing or creating network connections, interacting with other threads, or creating new threads in the directory agent run time. The Java memory model ensures that the downloaded code can not corrupt a directory agent's memory, except the objects passed to it as parameters. The only potential security problem unaddressed is deny of service attacks. For example, the downloaded code runs forever or instantiates large amount of random objects or buffers with the intention to dry up the system's memory.

5.2 Directory Agent Implementation

In our prototype implementation, a directory agent is implemented in Java. We used the freely available XML parser Xerces [1] Java Parser to parse information encoded in XML into a Document Object Model (DOM) [7]

```
<?xml version="1.0" encoding="UTF-8"?>

<schema xmlns="http://cordelia.lcs.mit.edu/MySchema"
xmlns:mo="http://cordelia.lcs.mit.edu/MonitorSchema"
targetNamespace="http://cordelia.lcs.mit.edu/MonitorSchema"
targetCodebase="http://cordelia.lcs.mit.edu/MonitorSchema.jar"
elementFormDefault="qualified">

<import namespace="http://cordelia.lcs.mit.edu/MySchema"/>

<element name="monitorDesc" type="mo:MonitorDesc"/>

<complexType name="MonitorDesc">
 <sequence>
   <element name="gpsLocation" type="GPSLocation"/>
   <element name="resolution" type="mo:Resolution"/>
   <element name="protocol" type="string"/>
   <element name="netAddress" type="INETAddress"/>
 </sequence>
</complexType>

<complexType name="Resolution">
 <sequence>
   <element name="x-res" type="positiveInteger"/>
   <element name="y-res" type="positiveInteger"/>
   <element name="PixelSumGreaterThan" minOccurs="0">
     <complexType>
       <sequence>
 <element name="minPixelSum" type="positiveInteger"/>
       </sequence>
     </complexType>
   </element>
 </sequence>
</complexType>

</schema>
```

Figure 8: An XML Schema for Monitor Service

```
<?xml version="1.0" encoding="UTF-8"?>
<monitorDesc xmlns="http://cordelia.lcs.mit.edu/MonitorSchema">
 <gpsLocation>
   <latitude>60.5</latitude>
   <longitude>130.5</longitude>
   <height>20.0</height>
 </gpsLocation>
 <resolution>
   <x-res>1024</x-res>
   <y-res>768</y-res>
 </resolution>
 <protocol>X11</protocol>
 <netAddress>
   <ip>18.26.0.55</ip>
   <port>8081</port>
 </netAddress>
</monitorDesc>
```

Figure 9: An XML Description for Monitor Service

trees. To check whether a service with the required type identifier matches a query, a directory agent needs to evaluate the search records specified in the query and then perform the required logical operations on the evaluation results.

Following is a description of the steps that a directory agent takes to evaluate a search record:

1. If the matching function definition has not been loaded, load it at run time. This is accomplished by a customized Java class loader, whose findClass method is over written to use data type table shown in Table 1 to locate the definition of the matching function.

2. Get the DOM node specified by the path of the query, which we refer to as the first node.

3. Parse the asserted node value of the query and instantiate a DOM node corresponding to it, which we refer to as the second node.

4. Instantiate the parameters specified in the query, verify the number of the parameters and their types based on the corresponding specification in the schema that specifies the matching function. An example of the parameter specification is shown in Figure 8, which specifies that matching function "PixelSumGreaterThan" takes a single parameter, "minPixelSum" of positiveInteger type, in addition to the first and second nodes. The evaluation process only proceeds if the verification succeeds.

5. Invoke the match function by passing it the first node, the second node, and the parameters. The return boolean value of the function serves as the result of the evaluation of the search record.

The prototype is still under development.

We are expecting to get it done very soon.

6 Related Work

Locating an object or related information in a distributed system has been studied for many years. Service discovery is a variant of the same problem. There are many existing protocols and architectures in this area.

6.1 Semantic Web

The goal of Semantic Web [4] is to make the semantics of information presented in web pages understood by machines so that so that intelligent information retrieval is possible. W3C consortium have been working on Resource Description Framework (RDF) [22], which specifies a logical structure for modeling resources, XML and XML schema to describe structured data, and XQuery language to query a document written in XML. Though we have been working on a different problem — enabling semantic service discovery in the mobile and pervasive computing environment, we found we shared similar opinions. That is, structured data themselves are not sufficient for defining semantics. In the design of Semantic Web, semantics are conveyed by inference rules, which will be written in a web page. In our design, as the relevant semantics to the discovery problem are limited, we specify them by defining meaningful matching and comparison operations over a data type.

6.2 Lightweight Directory Access Protocol

Lightweight Directory Access Protocol (LDAP) [28, 29] is a lightweight version of X.500 Directory Access Protocol. It is designed to allow access to information and resources of other computer systems. Unlike other service discovery protocols or architectures, the information stored at an LDAP server is rather static. And there is no automatic directory server discovery mechanism. LDAP is specified in ASN.1 and are transferred according to a subset of ASN.1's Basic Encoding Rules.

Each object in LDAP is associated with a Distinguished Name, which is organized in a global hierarchy, and a set of attribute-value pairs. LDAP specifies a set of standard attribute types and their associated matching rules. An implementation of a directory server can extend them.

6.3 Service Location Protocol

Service Location Protocol (SLP) [20, 18, 16, 17] is a protocol for automatically discovering the network locations of services and resources on an IP network. The architecture of our framework is based on SLP.

Services in SLP are described by instantiating a service template. SLP defines its own description language. The attribute types include Boolean, integer, string and octet strings. The supported matching rules include integer comparison, boolean comparison and string comparisons. There is no mechanism specified to extend the set of attribute types and matching rules.

6.4 Jini

The design of Jini [13] is very similar to that of SLP, though the goal is quite different. Jini aims at providing a distributed computing platform [30]. Thus, it not only specifies how to discovery a service but also specifies how to invoke that service. The directory component in Jini is implemented as a Lookup service.

In Jini, a service is represented by a Java object. A service is described by the service ID, the service type and an array of attribute objects. Though a Jini's service is described by a powerful programming language, the matching rules are extremely simple. Only wild card and exact matching are supported.

6.5 Universal Plug and Play

UPnP [10, 11] is another architecture for enabling pervasive computing. The basic components in the UPnP architecture are devices and control points, and there is no directory component. UPnP defines both the service discovery mechanisms and the *device* $\rightarrow$ *control point* interaction mechanisms. UPnP uses HTTP and its extension as its communication protocol.

In UPnP, services and devices are described by XML documents. UPnP forums specifies the UPnP device template and the UPnP service template by XML schemas. However, the service/device's discovery is not based on a service/device description. In the discovery operation, the advertisement message and the search response message only includes a device/service type, a unique identifier and a URL to the service description.

6.6 Salutation

Similar to Jini and UPnP [8, 9], it provides a framework both for the service discovery and for service utilization. Salutation architecture defines an entity Salutation Manager (SLM) that acts as a service broker between networked entities. A network entity can either be a service or a client. The feature of Salutation is that it intends to be transport-independent. The Salutation architecture defines a transport-independent API between services or clients and a Salutation Manager, and a transport-independent API to a transport dependent entity called Transport manager. Salutation protocols are defined over SunRPC.

In the Salutation architecture, a service may consist of a set of Functional Units (FU), which is a logical subdivision of a device or service. A Service Description is composed of a set of FU Description records. A FU description record includes the type of FU defined by Salutation Consortium, a unique identifier assigned by the registering SLM and a set of Attribute Description Records. SLM support comparison functions on primitive attribute types such as string and integer.

6.7 Other Service Discovery Systems

Berkeley's Secure Secure Discovery Service (SSDS) [12] architecture provides a secure, fault-tolerate and scalable service discovery framework. In SSDS, directory servers are organized into a hierarchical structure. A server will spawn new servers if the server load reaches a threshold. A failed server can be automatically restarted by its parent. SSDS provides a very strong secure mechanism. All traffic are encrypted and endpoints are authenticated. An SSDS server will answer a query for which a client has valid capabilities. In SSDS, services and queries are described by XML. The SDS server uses its internal XSet [34] XML search engine to match a query with service descriptions. XSet supports string and integer comparisons.

MIT's Intentional Naming System [2] is another resource discovery system. Similar to the Semantic File Systems, it extends the traditional concept of "name" into a set of attribute-value descriptions. INS also implements integrates name resolution and message routing, enabling clients to continue communicating with services even if the name-to-address bindings change while a session is in progres. In INS, name lookups are also based on syntax comparisons.

Hive [24] is a distributed agent architecture for enabling ubiquitous computing. Each networked service is represented by a Hive agent. Agent software are hosted by Hive cells. An agent is represented by both its Java type and an XML description. A Hive Cell provides a discovery service for agents to find out each other. The query matching scheme is also syntax-based comparison.

IBM's T-Space [32] project provides a middle-ware solution for pervasive comput-

ing. A T-Space is a middle layer that connects all programs. A client does not have to discovery a service in order to use it. Clients and services communicate by sending tuples to an T-Space. A tuple is a vector of typed values. A service registers as a listener to a tuple. To use a service, a client writes a tuple to T-space. The T-Space performs the tuple matching. If a client's tuple matches the one service registered with, the tuple is sent to the service. Tuple matchings are simple integer and string matchings.

7 Conclusion and Future Directions

In this paper, we discussed in depth the conceptual issues involved in enabling semantic service discovery. Our main point is that using structured data to represent a service is not enough to convey semantics of the representation for the service discovery. The semantics is captured by the way data are interpreted. To enable semantic service discovery, a directory agent should understand at least the equivalence relation, the approximate equivalence relation and the partial order relation over any data type used to describe a service. To achieve this, we designed a framework that allows a directory agent to download specific matching functions related to new data types at run time.

We observe at least three problems that are worth further study. First, XML as a service description language is far from satisfying. We would like to have a more succinct language. Second, we handle only simple queries in our framework. Queries with more complicated constraints may be needed. In our framework, we have intentionally simplified the query language, including only logical operators. In another extreme, the query language can be powerful enough to specify the matching functions. We chose not to do that for at least three reasons: 1. We want the code for matching functions to be reused among clients; 2. We want the code to be verifiable. If the code is supplied by the service designer, it can be authenticated once and used by all clients; 3. We want the query to be simple. As queries may be composed by end users directly, we can not expect them to write complicated code. However, other than logical operators, whether other expressive power is needed in the query language is an open question. Third, our current work aims at service discovery in a LAN. It is possible to apply this idea to discover general objects in a Wide Area Network (WAN). In that situation, performance will be an issue. We would like to see how this can be scaled up to semantic service discovery in WAN.

References

[1] Xerces. http://xml.apache.org/xerces-j/.

[2] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the ACM Symposium on Operating Systems Principle (SOSP'99)*, 1999.

[3] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Third Edition.* Addison-Wesley, 2000.

[4] Tim Berners-Lee. http://www.w3.org/DesignIssues/Semantic.html.

[5] Paul V. Biron and Ashok Malhotra. Xml schema part 2: Datatypes. W3c proposed recommendation, W3 Consortium, may 2001.

[6] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language. W3c recommendation, W3 Consortium, oct 2000. http://www.w3.org/TR/REC-xml.

[7] Ben Chang, Andy Heninger, Joe Kesselman, and Rezaur Rahman. Document object model (dom) level 3 content models and load and save specification. W3C working draft, W3 Consortium, apr 2001. http://www.w3.org/TR/2001/WD-DOM-Level-3-CMLS-20010419/.

[8] The Salutation Consortium. Salutation architecture specification (part 1).

http://www.salutation.org/specordr.htm, June 1999.

[9] The Salutation Consortium. Salutation architecture specification (part 2). http://www.salutation.org/specordr.htm, June 1999.

[10] Microsoft Corporation. Universal plug and play device architecture. http://www.upnp.org/resources.htm, June 2000.

[11] Microsoft Corporation. Upnp template guidelines. http://www.upnp.org/resources.htm, May 2000.

[12] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. In *the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networks (Mobicom'99)*, Seattle, Washington, August 1999.

[13] W. Keith Edwards. *Core Jini*. Prentice Hall, Dec 2000.

[14] David C. Fallside. Xml schema part 0: Primer. W3c proposed recommendation, W3 Consortium, may 2001.

[15] Li Gong. *Inside Java 2 Platform Security: Architecture, API Design, Implementation*. Addison-Wesley, 1999.

[16] E. Guttman. Attribute list extension for the service location protocol. RFC 3059, IETF, Feb 2001. ftp://ftp.isi.edu/in-notes/rfc3059.txt.

[17] E. Guttman, C. Perkins, and J. Kempf. Service templates and service: Schemes. RFC 2609, IETF, June 1999. ftp://ftp.isi.edu/in-notes/rfc2609.txt.

[18] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2608, IETF, June 1999. ftp://ftp.isi.edu/in-notes/rfc2608.txt.

[19] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification, Second Edition (The Java Series)*. Addison-Wesley, 2000.

[20] J. Kempf and J. Goldschmidt. Notification and subscription for slp. RFC 3082, IETF, Mar 2001. ftp://ftp.isi.edu/in-notes/rfc3082.txt.

[21] John Larmouth. *ASN.1 Complete*. Morgan Kaufmann, 2000.

[22] Ora Lassila and Ralph R. Swick. Resource description (rdf) model and syntax specification. W3c recommendation, W3 Consortium, feb 1999. http://www.w3.org/TR/REC-rdf-syntax/.

[23] D. Meyer. Administratively scoped ip multicast. RFC 2365, IETF, 1998. ftp://ftp.isi.edu/in-notes/rfc2365.txt.

[24] N. Minar, M. Gray, O. Roup, R. Krikorian, and P. Maes. Hive: Distributed agents for networking things. In *the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, 1999.

[25] P. Mockapetris and K. Dunlap. Development of the Domain Name System. *Proc. of the ACM SIGCOMM '88, Stanford, California*, pages 11–21, August 1988.

[26] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. Xml schema part 1: Structures. W3c proposed recommendation, W3 Consortium, may 2001.

[27] Maarten van Steen, Franz J. Hauck, Philip Homburg, and Andrew S. Tanenbaum. Locating objects in wide-area systems. *IEEE Communications Magazine*, 1998.

[28] M. Wahl, A. Coulbeck, T. Howes, and S. Kille. Lightweight directory access protocol (v3): Attribute syntax definitions. RFC 2252, IETF, Dec 1997. ftp://ftp.isi.edu/in-notes/rfc2252.txt.

[29] M. Wahl, T. Howes, and S. Kille. Lightweight directory access protocol (v3). RFC 2251, IETF, Dec 1997. ftp://ftp.isi.edu/in-notes/rfc2251.txt.

[30] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A Note on Distributed Computing. Technical report, Sun Microsystems Laboratories, Inc, 1994.

[31] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, July 1993.

[32] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T spaces. *IBM Systems Journal*, 37(3), August 1998.

[33] Xiaowei Yang. A comparison of service discovery protocols and architectures. http://ana.lcs.mit.edu/ yxw/SSD/comparison.ps.

[34] Ben Y. Zhao. Xset. http://www.cs.berkeley.edu/ ravenben/xset/.