Semantic-Based Context-Aware Service Discovery in Pervasive-Computing Environments

by

Abdur-Rahman El-Sayed

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2006

Abstract

Recent technological advancements are enabling the vision of pervasive or ubiquitous computing to become a reality. Service discovery is vital in such a computing paradigm, where a great number of devices and software components collaborate unobtrusively and provide numerous services. Current service-discovery protocols do not make use of contextual information in discovering services, and as a result, fail to provide the most appropriate and relevant services for users. In addition, current protocols rely on keyword-based search techniques and do not consider the semantic description of services. Thus, they suffer from poor precision and recall.

To address the need for a discovery architecture that supports the envisioned scenarios of pervasive computing, we propose a context-aware service-discovery protocol that exploits meaningful contextual information, either static or dynamic, to provide users with the most suitable and relevant services. The architecture relies on a shared, ontology-based, semantic representation of services and context to enhance precision and recall, and to enable knowledge sharing, capability-based search, autonomous reasoning, and semantic matchmaking. Furthermore, the architecture facilitates a dynamic service-selection mechanism to filter and rank matching services, based on their dynamic contextual attributes, which further enhances the discovery process and saves users time and effort. Our empirical results indicate the effectiveness and feasibility of the proposed architecture.

Acknowledgments

I must first thank my supervisor, Professor James P. Black, for his support, patience, and encouragement throughout my Masters study. His knowledge and continuous guidance enabled me to conduct and complete this work successfully. I am also grateful to my thesis readers, Professor Tim Brecht and Professor Ian McKillop, for their helpful comments and constructive suggestions. Further, I would like to thank all members of the Shoshin research group, including Omar Khan, Hao Chen, Herman Li, and Georgia Kastidou, for their insightful discussions and comments.

I would also like to express my acknowledgements to all my friends at Waterloo, especially my great office mates, Basem Shihada and George Beskales, for their invaluable advices and social support. Last but not least, I am infinitely grateful to my family for their unconditional love and caring during the development of this thesis and over the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the vision of pervasive or ubiquitous computing proposed by Mark Weiser [73] in 1991, data is everywhere, technology becomes invisible in our lives, and a great number of devices and software components collaborate unobtrusively in a smart space to provide services. The vision is becoming a reality with advances in network and sensor technologies, the widespread deployment of network-enabled devices, and the innovations of the service-oriented computing paradigm. Fortunately, it is becoming feasible to deploy pervasive-computing environments as the required hardware, such as complex tiny sensors, is becoming available off the shelf at reasonable cost. Furthermore, the size of the required sensors and computing hardware continues to decrease, enabling technology to become effectively invisible in our lives.

A crucial challenge facing pervasive-computing environments is the development of a service-discovery protocol that allows users and applications to discover and interact with the most appropriate and relevant services, provided and advertised by many devices and software components in the environment. In addition, service-discovery techniques in such environments should handle the dynamic appearance and disappearance of devices and services in a timely, secure, and efficient manner that does not violate the privacy of users.

The context-aware computing literature defines context as "information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to interaction between a user and an application including the user and application themselves. [14]" Dan Hon *et al.* [26] classify contextual information into three main categories: computing context (available CPU, memory, bandwidth), user context (preferences, calendar, personal information), and physical context (location, time, destination, weather). On the other hand, context-awareness, which is a key characteristic of pervasive computing, is defined as "a property of a system that uses context to provide relevant information and/or service to the user, where relevancy depends on the user's task" [14]. Consequently, context-aware service discovery can be defined as the ability to make use of context information to discover the most relevant services for the user.

Typically, pervasive-computing environments provide various services, which can be hardware-based, such as a printer or a light, or complex software-based facilities, such as an online e-commerce service. These heterogeneous services have different properties, capabilities, interfaces, and invocation schemes. As a result, many challenging questions arise. How do we standardize the description of these heterogeneous services? How do we capture their semantics? How do we enable users/agents to discover and invoke them autonomously? How do we define the "most appropriate" service? How do we represent the static and dynamic contextual information about both the user and the available services? Most importantly, how do we incorporate this information into the discovery protocol to make it context-aware?

Various service-discovery protocols have been proposed, designed, and implemented. While they share the main goal of providing a mechanism for service advertisement and discovery, they vary significantly in aspects like the architectural design and working environment (i.e., LANs, mobile ad hoc, Internet). Current service-discovery protocols, such as Jini [58], UPnP [59], Salutation [52], SLP [19], Ninja/SDS [18], INS/Twine [2], and UDDI [10], are not suitable for pervasive-computing environments due to two main challenges.

First, they rely on a syntactic description of services and on keyword-based search mechanisms. Thus, they are prone to low precision and recall. Service descriptions can be syntactically different but semantically equivalent. As a result, a user searching for a “print” service would not be able to locate a “printing” service. Similarly, users searching for “buying” services will not be able to locate “purchasing” services, because they have a different syntactic representation, even though they have the same meaning. In information retrieval terminology, this leads to poor recall, where recall is defined as the ratio of the number of relevant services discovered to the total number of relevant services in the environment.

Another important issue is that service descriptions can be semantically different but syntactically equivalent. As a result, a keyword-based search mechanism, which most discovery architectures rely on, will return irrelevant services for the user as “matching” services, due to that fact that a keyword can have multiple meanings (homographs). Similarly, if discovery is based on a syntactic representation of interface parameters rather than keywords, irrelevant services will still be discovered as “matching” services. Consider for example a user searching for a stock quote service that takes as an input a string and as an output it returns a float. Many services match such a request, and as a result, irrelevant services will be returned to the user as “matching” services. In information retrieval terminology, this leads to poor precision, where precision is defined as the ratio of the number of relevant services discovered to the total number of irrelevant and relevant services discovered.

The second main challenge is that the current discovery protocols are not context-aware; they do not consider contextual information in discovering services, and as a result, they fail to provide the most relevant and appropriate services for users [12, 76]. This is crucial for pervasive-computing environments. Consider for example a user searching for a printing service. The service-discovery protocol should exploit the context of the user, which could include the location of the user and her preferences, as well as the context of the service, which could include the location and the current load of the printer (expressed in terms of queue length). In this scenario, the protocol should discover the nearest and least-loaded printing service for the user. Similarly, if the user is searching for a software-based facility, the protocol should discover the service with the highest Quality of Service (QoS). Ignoring contextual information during the discovery phase places the burden of choosing the most appropriate or relevant service on the user, by forcing her to analyze and understand each matching service description manually — contradicting two key aspects of pervasive computing, unobtrusiveness and user friendliness.

1.1 Motivating Scenario

To demonstrate the benefits and advantages of relying on a semantic service description and incorporating contextual information into service discovery, we present the following scenario. Alice is a new undergraduate student on campus. She plans to go out on a Saturday night and eat in a restaurant. She would like to find a nearby restaurant using her PDA and obtain its food menu to determine if the restaurant is suitable (e.g., vegetarian). Then, she plans to print out a map describing the route to the chosen restaurant from her current location. Finally, she wants to drive to the restaurant and park her car freely, assuming that the restaurant does not have its own parking lot.

Using a typical service-discovery protocol that does not make use of contextual information nor semantics, after submitting a request for a "restaurant" facility, Alice's PDA will receive a long list of matching restaurant services. Then, she has to inspect the description of each restaurant manually and determine if it is suitable, according to her preferences (i.e., offers vegetarian food, nearby) — a tedious and cumbersome task. In addition, it is possible that many matching restaurant services will not be discovered, because their syntactic description might be different than the one used in the service query issued by Alice. On the other hand, if the discovery protocol utilizes the contextual information about Alice, such as her location and preferences (e.g., vegetarian food), and the contextual information about the restaurants, such as the location and available seats, the nearest restaurants offering vegetarian food will be discovered, ranked according to Alice's preferences, and sent to her PDA, saving her a considerable time and effort. Moreover, if the discovery protocol relies on semantics rather syntax, through reasoning and semantic matchmaking, irrelevant services that include the word "Restaurant" in their description will be eliminated, while restaurant services that are advertised using a syntactic representation different than the one used in the service request (e.g., Bistro or Café instead of Restaurant) will be discovered and considered as matches.

After choosing a specific restaurant, Alice obtains a map describing the route to the restaurant from her current location (using facilities like GoogleMaps or MapQuest) and wishes to print it. By exploiting contextual information, such as her location and preferences, and the location and queue length of the available printers, the discovery protocol will save Alice the time and effort required to inspect each printer description manually, by discovering and ranking the nearest and least-loaded printers for her. In addition, if service querying and matchmaking is based on semantics rather than syntax, with the support of reasoning, services that are advertised using a syntactic description different from "printer" (e.g., LaserPrinter, InkJetPrinter) will be discovered and considered for matchmaking. Notice that an adequate context-aware discovery arhictecture should allow Alice to prioritize her preferences by indicating their importance (weight). For instance, when seeking a printer, Alice might want to place a higher importance on the location of the printer than its current load.

Finally, as Alice drives to the restaurant and approaches it, she needs to park her car. By exploiting her context (location) and the context of the parking services (cost, location, current number of free spots), a context-aware discovery protocol can assist Alice in locating the nearest free parking with the highest number of available spots.

This motivating scenario illustrates the shortcomings of typical service-discovery protocols, the advantages of relying on a semantic-based matchmaking mechanism, and the benefits of utilizing contextual information during discovery.

1.2 Semantic-Based Context-Aware Discovery

We have tackled the two major challenges that prevent current discovery protocols from being suitable for pervasive-computing environments. As for the first challenge, to capture the semantic description of services, set a common understanding, and provide a semantic- instead of a syntax-based search facility, we rely on an ontology-based mechanism to describe services and contextual information.

For describing the semantics of services, the latest research in service-oriented computing recommends the use of the Web Ontology Language for Services (OWL-S) [64], which is based on the Web Ontology Language (OWL) [65]. OWL is currently the de facto standard for constructing ontologies. It is a part of the Semantic Web project [70], which aims to define and add a standardized machine-readable meaning to information published on the World Wide Web. By utilizing this machine-readable meaning, software agents will be able to find, integrate, understand, and "reason" autonomously about information. OWL-S, on the other hand, is an ongoing effort to enable automatic discovery, invocation, and composition of Web Services (WS). It is an ontology designed to describe the properties and capabilities of web services. Instead of developing our own service ontology, we reuse the OWL-S ontology to describe services. However, since OWL-S does not include a semantic description of contextual information, it does not support context-aware discovery in pervasive-computing environments. Thus, we extend the OWL-S ontology to include a semantic description of contextual information, according to our discovery requirements and goals. Furthermore, we extend the OWL-S ontology to facilitate new invocation schemes based on the proposed architecture.

To tackle the second major challenge, making discovery context-aware, we integrate our service-discovery architecture with a context engine responsible for representing and maintaining information describing the current situation of service providers, users, and all services within the environment. Consequently, when a service request is issued, the discovery protocol coordinates with the engine to retrieve contextual information about both the user and the available services, and calls a semantic matchmaking algorithm to discover the most suitable ones. Finally, the protocol refines the result by filtering and ranking the set of matching services to enhance the overall quality of the result and to save users time and effort.

Very recently, Anand Ranganathan *et al.* [46] proposed a benchmark for evaluating pervasive-computing environments. Since service discovery is a vital functionality, the authors identify the following aspects to evaluate different discovery protocols for such environments.

1. **Precision and Recall.** Essentially, increasing recall and precision is desirable. To achieve this, we use an ontology-based approach to describe services, enabling the discovery protocol to "understand" and "reason" about services to discover the relevant ones provided in the environment and to exclude the irrelevant ones from the result.

2. **Context-Sensitivity**. Does the discovery protocol consider the contextual information of the user/services? To satisfy this criteria, we integrate our architecture with a context engine that maintains contextual information. By coordinating with this engine, the discovery protocol utilizes the context of the user and services to discover the most suitable ones.

3. **Semantics**. Does the discovery protocol rely on semantics or syntax (keywords) to represent and answer service requests? We designed the architecture to provide a capability-based search facility that relies on semantics rather than a keyword-based search mechanism, which leads to poor recall and precision.

4. **Scalability**. Does the discovery protocol scale with regards to the number of devices and services (i.e., large-scale environments)? Addressing this issue is considered future work, as will be shown.

1.3 Thesis Contributions and Outline

This thesis addresses the need for a service-discovery protocol that supports the upcoming pervasive-computing paradigm [73]. Specifically, the contributions of this thesis are as follows.

- Construction of an OWL-based ontology to facilitate context-aware discovery.
- Design and implementation of a semantic-based discovery architecture that provides a capability-based search facility and that exploits meaningful contextual information to discover and rank the most appropriate services for users and agents.
- The development of several services with different invocation schemes as a proof-of-concept for the validity of the architecture.

The remainder of this thesis is organized as follows. In Chapter 2, we present an overview of Impress [4], the umbrella project for this thesis, as well as an overview of the Semantic Web technologies, including OWL and OWL-S. In the same chapter, we discuss the existing approaches towards context-aware discovery and highlight their shortcomings. The key aspects of the proposed discovery architecture, such as context representation and publication, service description, service request, service matchmaking, service ranking, and service invocation, are presented in Chapter 3, which constitutes the core of the thesis. We discuss the current implementation status and present an overview of our prototype in Chapter 4. Finally, we conclude and present future work in Chapter 5.

Chapter 2

Background and Related Work

In this chapter, we present an overview of the umbrella project for this thesis, Impress. Then, we provide some background information on the Semantic Web technologies, including the OWL language and the OWL-S ontology. Afterwards, we present the current approaches towards context-aware service discovery and discuss their limitations and shortcomings.

2.1 Impress

The work presented in this thesis is part of an ongoing project at the University of Waterloo, Impress [4], which aims to turn ubiquitous or pervasive computing into a reality. The focus of the Impress project is to provide a feasible platform for ubiquitous computing environments that supports the development of ubiquitous computing applications. As for the requirements of this platform, it should provide a mechanism to identify the entities within the environment uniquely, and support their mobility and heterogeneity. In addition, the platform should provide a secure communication infrastructure to enable the exchange of information among entities in a manner that does not violate the integrity of the information or the privacy of users. The platform should also be extensible.

The above requirements are supported by Jabber [30], an open-source, distributed, XML-based instant-messaging system, and hence, the Impress project and the architecture presented in this thesis are based on it. In 2004, the Internet Engineering Task Force (IETF) formalized Jabber's XML streaming protocols and approved them under the name of XMPP, the Extensible Messaging and Presence Protocol [29].

Jabber defines the notion of a unique entity and permits it to exchange unconditional XML-based messages with other entities in the network, in a secure manner. In addition, Jabber provides a Publish/Subscribe ("pubsub") mechanism that suits the requirements of the context engine and the proposed discovery architecture, as will be shown in future sections. Furthermore, new functionalities can be introduced into Jabber easily, since it defines a clear extensibility method, Jabber Enhancement Proposals (JEP) [51]. Finally, Jabber has proven to be deployable. It is running on thousands of servers across the internet and used by millions of users. Further details on how Jabber/XMPP can be used as a pervasive-computing platform can be found in [5].

2.2 The Semantic Web

The Semantic Web was invented by Tim Berners-Lee, the inventor of WWW, HTTP, and HTML. In his own words, he describes it as "an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation" [3].

We currently have an extremely large amount of electronic data available through the web. Yet, we cannot search it directly to find what we want. We cannot directly locate information regarding all red Ford Mustang cars which are located in the Kitchener-Waterloo region in Canada with a cost less than $7000. Even though this information is provided in different web sites (e.g., autonet.ca, autotrader.ca), it is described in different formats, terminologies, and layouts. Similarly, there is no standard way to search for all the web pages that have been authored by some one called Mike Lee, because different pages express this information in different ways (e.g., "page is created by Mike Lee", "page is published by Lee, Mike", etc.).

Thus, a human is required to visit each web site, understand its content, and finally arrive at some conclusions. Google cannot answer the previously mentioned search requests, since it relies on keyword-based search mechanisms. To summarize, even though this information we are looking for is available at different locations in the web, we cannot reach it directly because every website has its own data standards, terminologies, and HTML layout.

Researchers have tried using artificial intelligence techniques, such as natural language combined with schema matching methods, to make computers "smart" enough to understand and reason about the data published on the web in order to answer user requests using a unified query interface. An example of such an effort is the MetaQuerier system [8]. Unfortunately, natural language and schema matching techniques have proven to be complex and have not yet shown satisfactory results for querying and reasoning about web data using a unified interface. Instead of making computers "smarter," the semantic web project adopts a feasible and a much more promising strategy, which is making the data "smarter" to allow computers to understand and reason about it.

The main goal of the semantic web is to introduce a well-defined meaning to data, where this meaning represents a common machine-readable format for interchange of data that can be understood by computers. The vision here is that by using the semantic web technologies, software agents will be able to find, share, combine, understand, and reason about data.

The semantic web is based on the concept of ontologies. An ontology is simply a formal description of concepts in a real-world domain. It captures the concepts and the relationships between those concepts in that domain. Thus, an ontology provides a shared and common understanding of a particular domain. Ontologies are expressed in a machine-readable format that enables software agents to understand and reason about the concepts within the domain. The standard World Wide Web Consortium (W3C) language for creating ontologies is OWL, the Web Ontology Language, described in the next section.

2.2.1 Web Ontology Language (OWL)

The Web Ontology Language (OWL) is currently the de facto standard for creating and representing ontologies. It addresses the shortcomings of previous markup languages, such as XML, XML Schema [66], XML Namespaces [71], Resource Description Framework (RDF) [68], RDF Schema (RDF-S) [62], and DAML+OIL [11].

OWL is composed of three main aspects: classes, relationships, and instances. Classes represents the concepts within the real-world domain. For instance, in a university ontology, classes Student and Course may exist. Each class is identified using a global unique identifier that is composed of a namespace, usually a web location (e.g., http://www.uwaterloo.ca/Ontology#), and an ID (e.g., Course).

The second main aspect of OWL is relationships, which are captured using properties, where every property has a domain and a range. OWL provides two main types of properties: object and datatype properties. Object properties relate two classes within the ontology. For instance, in the university ontology, the property enrollsIn has the Student concept as a domain and the Course concept as a range. On the other hand, datatype properties connect classes and datatypes. For instance, the hasName property has the Student concept as a domain and the XML Schema definition of a String (XSD:String) as a range. The following OWL code describes the Student class, Course class, and the hasName and enrollsIn properties.

```
<owl:Class rdf:ID="Student"/>
<owl:Class rdf:ID="Course"/>

<owl:DatatypeProperty rdf:ID="hasName">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Student"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="enrollsIn">
```

```
  <rdfs:domain rdf:resource="#Student"/>
  <rdfs:range rdf:resource="#Course"/>
</owl:ObjectProperty>
```

The last main aspect of OWL is instances or individuals. They are analogous to Java instances of Java classes. Instances within the ontology represent specific elements with actual data. For example, CS338 can be an instance of the Course concept, while HPLaserJet200 can be an instance of the LaserPrinter concept. The following OWL code describes an instance of the Student class, Mike, who is enrolled in CS338, an instance of the Course class.

```
 <Student rdf:ID="Mike">
    <enrollsIn rdf:resource="#CS338"/>
    <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Mike Alan
    </hasName>
</Student>

<Course rdf:ID="CS338">
   <entitled rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
   Database Management Systems
   </entitled>
</Course>
```

One of the unique aspects of OWL that is not provided by other languages such as RDF/RDF-S, is that its properties can have specific characteristics, such as transitive, symmetric, or inverseOf. This helps in modeling real-world relationships and permits software agents to infer or deduce information. For example, assume that we have a property called hasAncestor, which is stated to be a transitive property.

Also, assume that in our ontology we know that the instance Mike hasAncestor John, and John hasAncestor Ali. From this information, an OWL reasoner can deduce that Ali is an ancestor of Mike. As another example, assume that we have a property called hasFriend, which is stated to be a symmetric property. Also, assume that we know that Ali hasFriend John. From this information, an OWL reasoner can deduce that John is also a friend of Ali. Finally, assume that we have a property called hasChild, which is stated to be the inverseOf the hasParent property, and we know that Ali hasChild John. From this information, an OWL reasoner can deduce that John's parent is Ali.

Another unique functionality that OWL provides is the ability to place value and cardinality restrictions on properties. For example, it is possible to specify that the hasFather property cannot have more than one instance (cardinality constraint), while the hasAge property cannot have a value higher than 150 (value constraint).

2.2.2 Web Services Ontology (OWL-S)

The Web Services Ontology (OWL-S) is an ontology that describes the properties, characteristics, and capabilities of web services. It is an enhancement of the DAML-S ontology [61], which describes web services semantically. As a core goal, OWL-S aims to enable the autonomous discovery, composition, invocation, and monitoring of web services by relying on a computer-interpretable description of services. As the OWL-S authors mention [64], automatic web service discovery "is an automated process for location of web services that can provide a particular class of service capabilities, while adhering to some client-specified constraints," while automatic web service invocation is "the invocation of a web service by a computer program or agent, given only a declarative description of that service, as opposed to when the agent has been pre-programmed to be able to call that particular service." Automatic service composition, on the other hand, "involves the automatic selection, composition, and interoperation of web services to perform some complex task, given a high-level description of an objective."

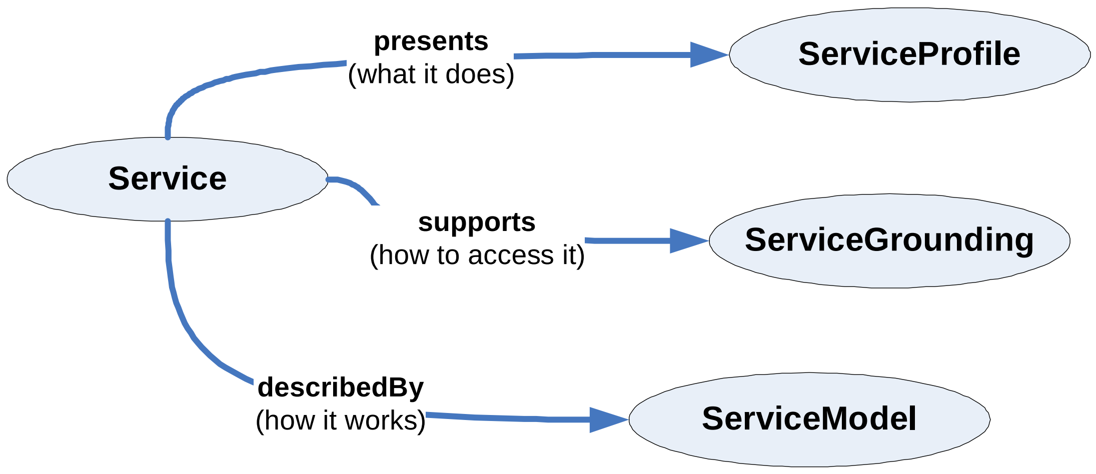


Figure 2.1: Overall structure of the OWL-S ontology

As shown in Figure 2.1, the OWL-S ontology is composed of three main concepts: a ServiceProfile describing the capabilities of the service (i.e., inputs/outputs), a ServiceGrounding describing the invocation details of the service (e.g., communication scheme, address, ports, etc.), and a ServiceModel describing the sub-tasks of the service and their execution order. The latter is used primarly to facilitate the composition of web services into sub-tasks to accomplish a desired goal.

The ServiceProfile describes the functional and non-functional aspects of a web service, and is therefore used for discovery. Most importantly, it describes the service inputs, outputs, preconditions, effects, and results. The capabilities (inputs/outputs) are expressed using concepts within the ontology or XML schema data types, while the preconditions and effects are described using logical formulas. In this thesis, we focus mainly on the ServiceProfile (or Profile), since it is used for service discovery. Autonomous service composition and invocation are beyond the scope of this thesis.

2.3 Existing Context-Aware Discovery Protocols

As [12] and [76] mention, only a few service-discovery architectures consider meaningful contextual information to discover the most appropriate services for users. In this section, we discuss relevant architectures that utilize contextual information in discovering services.

2.3.1 Location-aware Protocols

The Jini service-discovery protocol enables providers to include a location attribute in service advertisements, and thereby allows users to specify the desired physical location of a service during the lookup process. Similarly, the service-discovery protocol used in the Cooltown [21] project is location-aware; it permits users to discover nearby services. These two protocols exploit context in a limited manner; they only consider the location of the service or user to discover nearby services. However, context information is more than location and includes other meaningful details, as we show. Furthermore, these protocols do not share a common representation of location information.

2.3.2 Jini with "Context Attribtues"

Lee and Helal [36] realized the limitations of existing discovery protocols and introduced the concept of a "context attribute" associated with a service, as a part of its description. The authors augmented Jini with such attributes. In this extended protocol, service providers describe the context attributes associated with their services by coding specific Java classes, to be evaluated by Jini's lookup service when a relevant service request is issued. After evaluating all attributes, the lookup service ranks services according to a ranking expression, which is also defined by service providers, and returns the top matches to the user.

2.3.3 CB-SeC

Context-Based Service Composition (CB-SeC) [44] is a framework designed for context-aware service discovery and composition. Figure 2.2 gives an overview of the framework, which is composed of two main layers, a context management

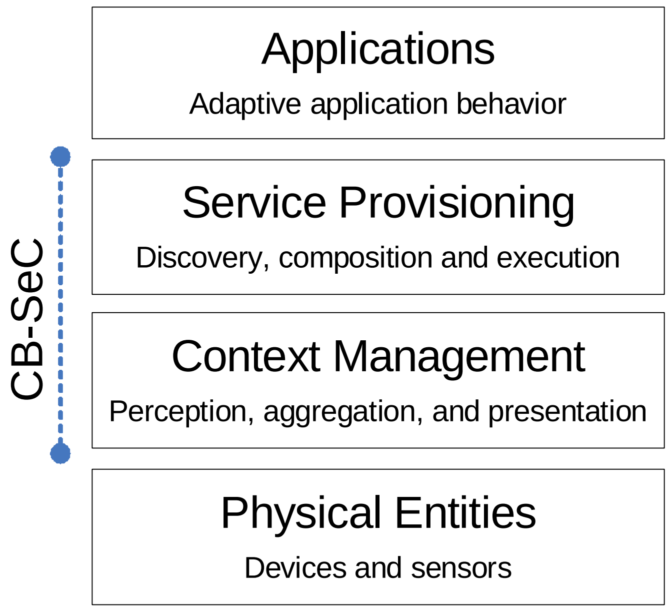

Applications
Adaptive application behavior
Service Provisioning
Discovery, composition and execution
Context Management
Perception, aggregation, and presentation
Physical Entities
Devices and sensors
CB-SeC

Figure 2.2: Overview of the CB-SeC framework

unit that receives, aggregates, and presents contextual information, and a service provisioning unit that handles the discovery, composition, and execution of services.

The framwork relies on a Context Gatherer module, which is responsible for gathering contextual information using hardware and software sensors. This information is then stored in a database, which is queried by a Brokering Agent to locate the most appropriate services on behalf of users. In service descriptions, providers can include a Context Function that indicates the dynamic contextual information associated with a service, such as the load on a printer. This function is exploited by the Brokering Agent to select the best service among matching services. Figure 2.3 shows an example of a CB-SeC service advertisement described using an attribute-value pair representation.

2.3.4 Context-sensitive Superstring

Robinson and Indulska [49] presented a context-sensitive discovery protocol as an extension to Superstring [48], a service-discovery framework that uses a struc-

```
Attributes*

service-identifier      =       "pai-acc121";
num instances           =       3 ; number of allowed instances
type                    =       "W-service"; //Web or M-service
isMobile                =       "No"; //whether the service can move to other places
description             =       "accommodation booking service";
provider-identifier     =       "PAI";
input-parameters        =       {Int Num of Persons, Int Num of Days, String Contact Name};
output-parameters       =       {XML Doc accommodation Details};
price                   =       5;      //e-coins per invocation

Capsule*

location                =       "pai-acc.diuf.com.ch";
protocol                =       https;
port                    =       80;

Constraints & Requirements*

diskfree                        >        20; //Kbytes
memoryfree                      >=       128; //Kbytes
OpSys                   =        "Palm OS, Linux"

Context Function                //represents the sensitivity of the service to context

CoF                             =        ping iiufps31.unifr.ch
```

Figure 2.3: Sample CB-SeC service description

tured peer-to-peer network to propagate service advertisements and requests. Their architecture enables service providers to include context attributes in service descriptions. Nevertheless, these attributes are only considered for matchmaking if included in service requests. Dynamic service ranking and selection is supported. However, in contrast to CB-SeC and the extended Jini protocol, defining a ranking expression is the responsibility of the requesting user.

The context-sensitive Superstring protocol supports *persistent queries*, enabling it to notify the requesting user of new matching services, and *query relaxation*, a mechanism by which service queries (requests) are weakened if no exact matches were found.

2.3.5 Issues with Current Protocols

Even though the protocols and frameworks described above incorporate context awareness in the discovery process, they suffer from a major drawback, which is their reliance on a syntactic representation of contextual information and service descriptions. Thus, unlike the architecture presented in this thesis, which also incorporates context awareness, they do not support capability-based search, semantic matchmaking, autonomous reasoning, or unambiguous knowledge sharing. Furthermore, they are prone to poor precision and recall, since they rely on keyword-based search mechanisms. We tackle these issues by utilizing concepts from the semantic web, and the context-aware and service-oriented computing paradigms.

Recently, Tom Broens *et al.* [7] proposed a service-discovery architecture that incorporates contextual information into discovery and utilizes semantic-web technologies. Similar to this thesis, the authors of [7] use the OWL language to construct the ontologies describing services and contextual information. However, while they create their own ontologies, we extend the OWL-S ontology, since it is rich and general enough to describe any service, it facilitates the autonomous invocation of web services, and it is the standard ontology for describing the properties and capabilities of web services in computer-interpretable form. Unlike this thesis, the authors of [7] process the contextual attributes in Boolean format only (e.g., nearby / not nearby), so they can use "concept lattices" [74] to rank matching services. This technique suffers from a major limitation. Assuming that there are two matching services, which are considered to be "nearby," since their architecture relies on Boolean contextual attributes, it fails to determine which service is nearer to the user (i.e., the most suitable service). Similarly, it fails to determine which printer has the minimal load (queue length). It will return a list of matching printers with a queue length less than a certain value, failing to identify the most suitable printers. In addition, unlike the proposed architecture in this thesis, their protocol does not facilitate a weighting mechanism by which users can place a higher importance on specific attributes than others. In other words, their architecture does not enable a user looking for a printing service to place a higher importance on the location of the printer than its load. Another difference between the two architectures is the mechanism by which contextual information is stored and processed. Tom Broens *et al.* [7] store the actual values of contextual attributes into a database containing the ontology classes and instances. However, since such attributes are dynamic and change frequently, it is inefficient to update the ontology whenever their values change, especially since the current OWL processing tools are in their infancy. In contrast to their approach, for the sake of flexibility and performance, we store the actual values of dynamic contextual attributes in a pubsub system that is engineered for efficiency, and store their references in the ontology instances.

Finally, Cuddy and Lutfiyya [12] present a context-aware service-selection mechanism that considers and assigns weights to static and dynamic contextual information associated with services. They integrate this selection mechanism into the Service Location Protocol (SLP). We base our dynamic service-selection technique on their mechanism, which ranks services based on the values and weights of their associated contextual attributes. However, in addition to service selection, we present the design and implementation of a complete context-aware service-discovery architecture. Moreover, instead of using attribute-value pairs to describe services, and static and dynamic contextual information as in [12], we rely on an ontology-based approach. Furthermore, the architecture presented in this thesis is designed to include a standardized mechanism by which services, whether software- or hardware-based, can register and publish contextual information, to be used for dynamic service-selection and ranking.

Next, we discuss the design decisions and the key aspects of the proposed architecture, such as context representation and publication, service description, advertisement, request, matchmaking, ranking, and invocation.

Chapter 3

Architecture

In this chapter, the key aspects of the proposed discovery architecture are discussed, including how services are described semantically (service description), how services are advertised and how their contextual information is published (service advertisement), how users or software agents request services (service request), how the discovery protocol locates services that satisfy the request (service matchmaking), and how it uses the contextual information to rank the matching services (service ranking).

3.1 Overview

Our architecture is designed for use in a pervasive-computing platform, which should provide a mechanism to uniquely identify the entities within the environment, enable them to exchange information in a secure fashion, and support their mobility and heterogeneity. Since Jabber supports these requirements, we have based our pervasive-computing environment on it. The overall structure of the architecture is presented in Figure 3.1. As the figure shows, the main entities in the architecture are the users/agents, services, a discovery component, and the context engine, where each entity is identified using a unique Jabber ID (JID). The JIDs of the discovery component and context engine are advertised using Jabber's simple,

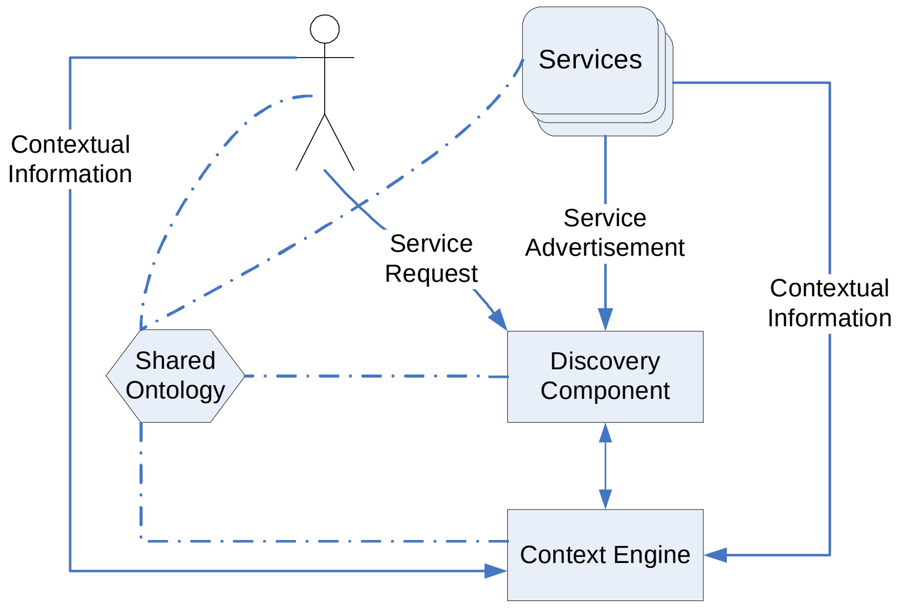


Figure 3.1: Overall structure of the discovery architecture

syntax-based, built-in discovery protocol [23].

The contextual information and service descriptions are expressed using a machine-readable ontology that is shared among all the entities. The context engine is responsible for acquiring and maintaining the contextual information, while the discovery component is mainly responsible for storing service advertisements and answering service requests sent by the users/agents. To determine the JID of the discovery component, first time users/agents send a simple discovery query defined by the built-in discovery protocol to the central Jabber server, which responds with a list of supported features and services (e.g., multi-user chat, context engine, discovery component), along with the JIDs of the entities that provide them.

For the sake of flexibility and extensibility, the context engine and discovery component are separate. However, whenever a service request is issued by a user, the discovery component coordinates with the context engine, by obtaining its JID using Jabber's discovery protocol, to retrieve the contextual information about the user and the available services. Afterwards, the discovery component calls a semantic matchmaking algorithm, which exploits the machine-readable format of the ontology to reason about the available services and discover the most appropriate ones, based on the retrieved contextual information. Finally, the component ranks the matching services and returns a description of the top-ranked ones to the user/agent.

If the description of a matching service is satisfactory, the user/agent can utilize it according to its invocation scheme. This invocation can be manual or autonomous. In the latter case, a software agent or program inspects the results of a discovery query, understands the necessary invocation details (e.g. input/output messages, communication scheme), and invokes the service by supplying appropriate inputs (which might be provided by the user). On the other hand, manual invocation requires a software agent to be pre-configured by a human that inspects the invocation details manually and programs the agent accordingly. In this thesis, since the focus is on designing and developing the core functionalities of a discovery protocol that supports the envisioned scenarios of pervasive computing, we currently adopt a manual invocation approach. However, since we rely on the OWL-S ontology to describe services, we capture the semantic description of the arguments used in invoking services. Thus, our architecture can be extended to support autonomous service invocation by developing a component that is capable of interpreting the inovcation details of a web service and invoking it "on the fly" without any pre-configuration or pre-programming. This component can be developed based on tools like the OWL-S API [55], which enables the execution of web services described using the OWL-S ontology with a minor pre-configuration effort.

3.2 Shared Ontology

Both the contextual information and service descriptions are represented using an ontology-based approach. Using a shared ontology, we facilitate knowledge sharing, enable reasoning and capability-based search, and ensure a common understanding among all entities in the environment. Instead of creating our own ontologies from scratch, we exploit the re-usability feature of ontologies and extend available ones. For representing contextual information, we use the Standard Ontology for Ubiquitous and Pervasive Applications (SOUPA) [9]. SOUPA is an ontology designed to "model and support pervasive computing applications." It reuses concepts from other ontologies, such as DAML-Time [24], DAML-Space [25], and FOAF [6], and represents generic concepts in pervasive-computing environments, such as person, agent, time, space, and event. Figure 3.2 shows an overview of the SOUPA ontology.

Even though the publicly available SOUPA ontology provides an adequate semantic representation of contextual information, which is the main reason we use it, it lacks a clear semantic description of services. Thus, for describing services, we utilize the OWL-S ontology and extend it (Section 3.4.1) based on our discovery requirements and goals. This combined ontology is shared among all the entities in the environment, including the context engine, the discovery component, services, users, and providers.

Our approach assumes that all the entities in the environment use the same global ontology. In large-scale environments, this might not be practical, as some entities may use other ontologies to describe the same concepts. In this case, ontology mapping techniques, such as those in [37, 38, 54], can be used to overcome this issue. Notice that ontology mapping is a common issue in any system that relies on ontologies for knowledge representation and sharing, and not only the architecture presented in this thesis.

3.3 Context Engine

The context engine maintains information about the environment, providers, users, and services. This information is obtained from software and hardware sensors (context sensors) as well as from the services in the environment, and is stored in the shared-ontology database as RDF triples [68] after possible aggregation and other processing. However, to support continuous queries and facilitate context exchange and collection, the actual values of contextual information are stored in a pubsub system. In other words, the context engine stores references to the pubsub

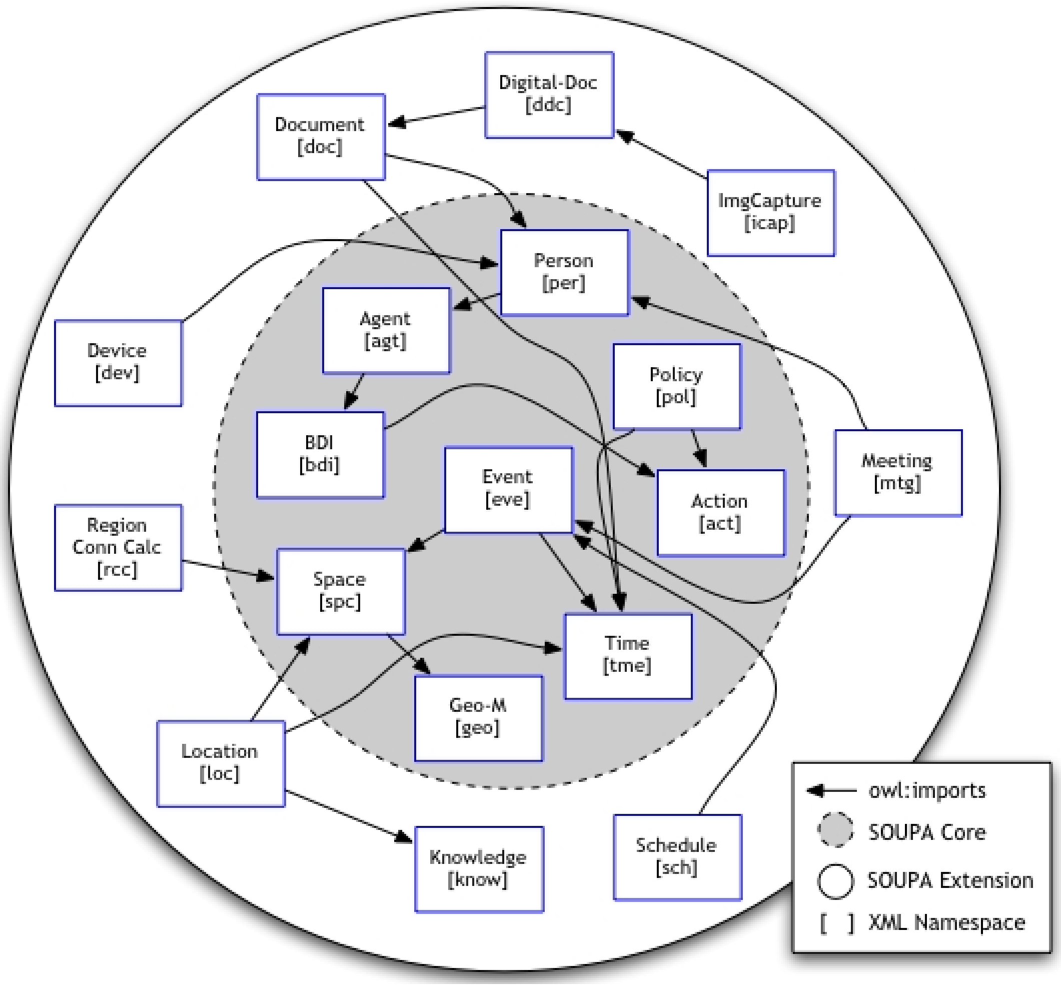


Figure 3.2: The SOUPA ontology

nodes in the shared-ontology database. In order to retrieve the value of a contextual attribute, a SPARQL query [69] is sent to the context engine, which parses the query and retrieves the actual value(s) from the corresponding pubsub nodes. Details on how the pubsub nodes are referenced in the ontology and on how services publish the value of the contextual information are described in Sections 3.4.1 and 3.4.2, respectively. Further information about the context engine can be found in [34].

3.4 Service Discovery

The discovery component is mainly responsible for storing service advertisements and coordinating with the context engine to answer service requests issued by users/agents. As mentioned previously, incorporating context-awareness into the service-discovery process enables the discovery of the most appropriate services. A few of the existing discovery-protocols exploit context in a limited sense by considering only the location of services and users [12, 76]. The architecture proposed in this thesis not only considers various semantically-described contextual information to discover the most appropriate services, but it also facilitates a dynamic service ranking and selection mechanism, which in turn, saves users effort and time. Various aspects of the discovery component are described in this section.

3.4.1 Service Description

In order to capture the semantic description of services, set a common understanding, and provide a capability- instead of a keyword-based search facility, we rely on an ontology-based mechanism to describe services. Even though OWL-S is tailored for web services, we believe it is rich and general enough to describe any service. Yet, web services, unlike most services in pervasive-computing environments, do not have any physical-location limitation and can be invoked from anywhere. Furthermore, OWL-S does not include a semantic description of contextual information. Thus, in its current state, OWL-S does not support context-aware discovery in pervasive-computing environments. We therefore extend the OWL-S

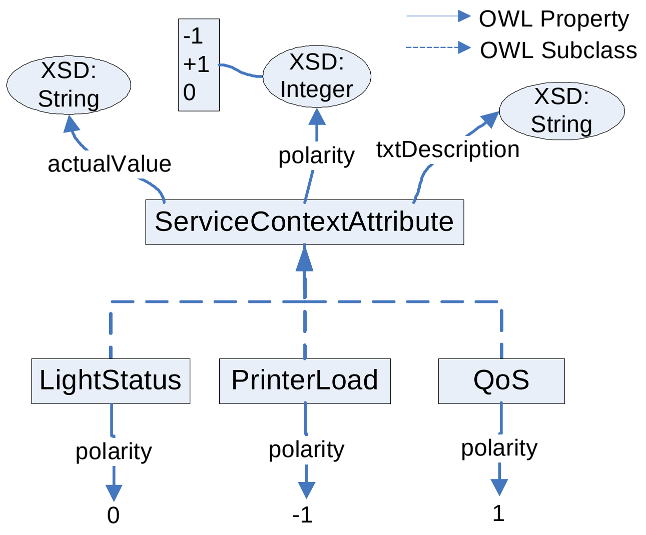


Figure 3.3: The ServiceContextAttribute class

ontology to include a semantic description of the dynamic and static contextual information associated with services, and to facilitate new invocation schemes based on the proposed architecture. Note that before extending the OWL-S ontology, we customized it by removing the concepts/classes required for service composition, since it is a complex problem and beyond the scope of this thesis.

Given that OWL-S does not support context-aware discovery, we extend it by adding a new class, ServiceContextAttribute. This new class captures the description of contextual information associated with services, information that is not provided by SOUPA or any other ontology, such as the load on a printer, the number of available tables in a restaurant, the status of a light, or the number of free spots in a parking service.

As shown in Figure 3.3, every instance of the ServiceContextAttribute class has three properties: actualValue, polarity, and txtDescription. The polarity indicates the desired value of the contextual attribute. It can be 0, to exclude the attribute from service ranking, +1, or -1, to indicate whether large or small values of the attribute are desired, respectively. For example, any instance of the PrinterLoad class, which is a subclass of ServiceContextAttribute, has a polarity of -1, denoting that the load on a printer should be as small as possible, while any QoS instance has +1 as its polarity, indicating that is should be as high as possible. The actualValue property refers to the pubsub node where the actual value of the attribute is published. The following OWL code describes the PrinterLoad class as a subclass of ServiceContextAttribute with a value restriction of -1 on its polarity.

```
<owl:Class rdf:about="#PrinterLoad">
    <rdfs:subClassOf rdf:resource="#ServiceContextAttribute"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:about="#polarity"/>
        </owl:onProperty>
        <owl:hasValue rdf:datatype="http://www.w3.org/2001/
         XMLSchema#int">-1</owl:hasValue>
      </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

In the architecture, the static information about a service, such as the service name and description, is stored in the ontology database. However, the dynamic contextual attributes, such as the load on a printer, should not be stored in the shared ontology, since they change frequently. It is impractical and inefficient to update the ontology whenever the actual value of a dynamic attribute changes, especially since the current OWL processing tools are in their infancy. It is more efficient and convenient to store the values in a separate pubsub system, and store static references to the pubsub nodes in the OWL-based ontology. The following OWL code describes an instance of the PrinterLoad class, identified as HPLaser2020_Load, along with its polarity, textual description, and pubsub node where the load is being published (actualValue).

```
<hasSCA>
  <PrinterLoad rdf:ID="HPLaser2020_Load">
  <actualValue rdf:datatype="http://w3.org/XMLSchema#string">
  pubsub.otter.uwaterloo.ca/home/otter.uwaterloo.ca/engine/302921/
  </actualvalue>
  <attrDescription rdf:datatype="http://w3.org/XMLSchema#string">
  The number of jobs in queue for this printer
  </attrDescription>
  <polarity rdf:datatype="http://w3.org/XMLSchema#int">-1
  </polarity>
  </PrinterLoad>
</hasSCA>
```

Having defined the semantics of contextual attributes associated with services, we extend the ServiceProfile class to include two new properties: locatedIn, which indicates the physical location of the service (assumed to be static), and hasSCA, which connects the ServiceProfile and ServiceContextAttribute classes, allowing services to have multiple **dynamic** contextual attributes. In addition to the location, custom static contextual information is captured and represented in the ontology, such as the pagesPerSecond and hasColor concepts, which are inlcuded in the Printer class (a subclass of ServiceProfile), as will be shown below.

An overview of the extended OWL-S ontology is shown in Figure 3.4. The service profile captures the service name, description, location, provider information, capabilities (inputs/outputs), and static and dynamic contextual attributes. Both the service name and textual description are represented as strings. The location of a service is captured using the locatedIn property and represented using a concept from the SOUPA ontology, GeographicalSpace. It is a subclass of Space with three properties, spatiallySubsumes, spatiallySubsumedBy, and hasCoordinates. The spatiallySubsumes property is defined to be *transitive*. It is also defined to be the *inverseOf* another property, spatiallySubsumedBy. The hasCoordinates property has the LocationCoordinates class as its range, which captures the GPS longitude

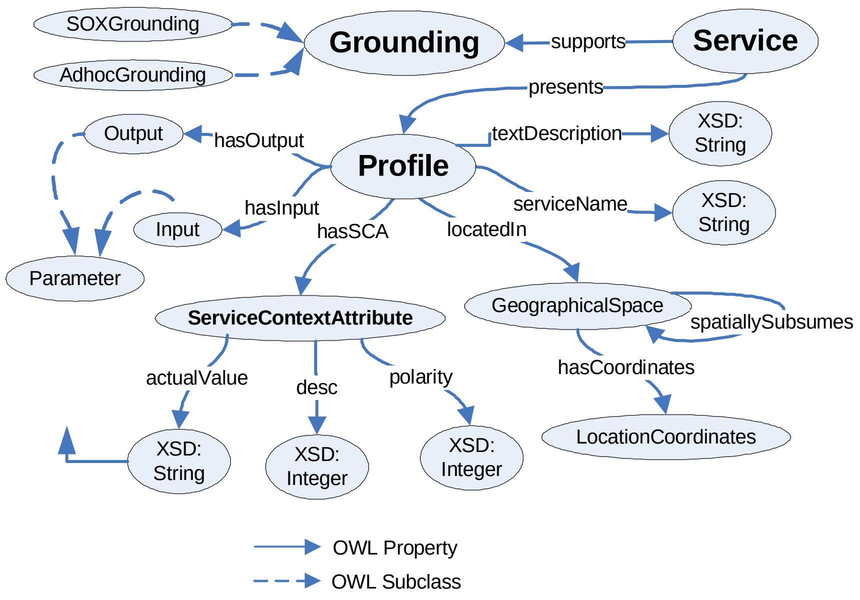


Figure 3.4: The extended OWL-S ontology

and latitude of the location. Using the GeographicalSpace concept, the discovery protocol can reason about locations and determine nearby services. Dynamic contextual information of services is captured using the hasSCA property, which has the ServiceContextAttribute class as its range. As for the service capabilities, the inputs and outputs are expressed using the hasInput and hasOutput properties, which point to a Parameter class. Any instance of a Parameter has a type (parameterType), which can be an OWL class (e.g., Book) or a primitive/custom type defined by XML Schema [66] (e.g., xsd:int, xsd:date, xsd:customType), and an optional value. As for the service grounding, we have extended the OWL-S ontology to support two new invocation schemes, SOXGrounding and AdhocGrounding, described in Section 3.4.6.

In pervasive-computing environments, the definition of a service is very broad. It can be hardware-oriented (such as a printer or a light), software-oriented (an online-shopping or text-translation service), or an abstract service with no groundings (restaurant, theatre, or parking). It is important to incorporate all three types of services in the discovery architecture to assist users or agents in the environment. Typically, different services have different concepts and contextual attributes. Thus,

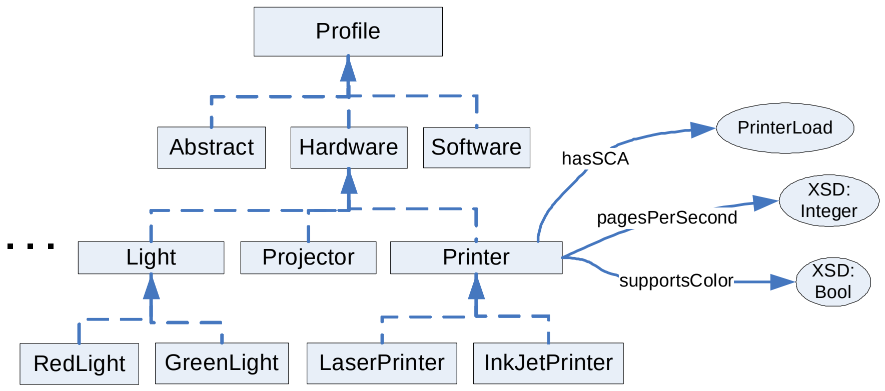


Figure 3.5: Hierarchy of service profiles

a hierarchy of service profiles can be used to express common understanding and prevent redundancy and confusion. Since there is no standard OWL-S ontology for hardware or software services, we created a preliminary hierarchy by extending the Profile with various classes, shown in Figure 3.5, to be shared among the entities in the environment. Referring to the hierarchy, any instance of the Printer class can have custom concepts, such as supportsColor and pagesPerSecond, with ranges as XSD:Bool and XSD:Integer, respectively. Fortunately, using OWL-S restrictions, it is possible to guarantee that related services have the same set of SCAs (e.g., all printers must have a load).

Figure 3.6 shows the OWL code for a LaserPrinter instance, identified as HPLaser-2020 (fictitious model name), which is located in DC3326 (an instance of GeographicalSpace), includes custom properties (supportsColor and pagesPerSecond), and has contextual information defined by an instance of the PrinterLoad class, identified as HPLaser2020_Load.

We present the following example to demonstrate the advantages of the semantic-based approach that we adopted, aside from enabling knowledge sharing, capability-based searching, and setting a common understanding. Consider a user looking for a facility with the following criteria.

(i) a printing service (Printer)

```
<LaserPrinter rdf:ID="HPLaser2020">
 <serviceName rdf:datatype="http://w3.org/XMLSchema#string">
 HPLaser2020
 </serviceName>
 <locatedIn rdf:resource="#DC3326"/>
 <supportsColor rdf:datatype="http://w3.org/XMLSchema
  #boolean">true
 </supportsColor>
 <pagesPerSecond rdf:datatype="http://www.w3.org/2001/
  XMLSchema#int">3
 </pagesPerSecond>
 <hasSCA>
   <PrinterLoad rdf:ID="HPLaser2020_Load">
    <actualValue rdf:datatype="http://w3.org/XMLSchema#string">
     pubsub.otter.uwaterloo.ca/home/otter.uwaterloo.ca/engine/302921
    </actualvalue>
    <attrDescription rdf:datatype="http://w3.org/XMLSchema#string">
     ....</attrDescription>
    <polarity rdf:datatype="http://w3.org/XMLSchema#int">-1
    </polarity>
    </PrinterLoad>
 </hasSCA>
<textDescription rdf:datatype="http://www.w3.org/2001/
 XMLSchema#string">This instance describes the HPLaser2020 printer
</textDescription>
</LaserPrinter>
```

Figure 3.6: Sample service-profile instance

(ii) located in the University of Waterloo (WaterlooUniversity)

Also, assume that the following information is stored and represented in our shared ontology.

(1) HPLaser2020 is an instance of LaserPrinter

(2) LaserPrinter is a *subClass* of Printer

(3) HPLaser2020 is *locatedIn* ShoshinLab

(4) WaterlooUniversity *spatiallySubsumes* DavisCenter

(5) DavisCenter *spatiallySubsumes* ShoshinLab

Using an OWL reasoner, the discovery protocol can deduce from (1) and (2) that HPLaser2020 is a Printer service, which satisfies (i). Likewise, since spatially-Subsumes is a transitive property, the protocol can deduce that WaterlooUniversity *spatiallySubsumes* ShoshinLab, and then further deduce that HPLaser2020 is located in the University of Waterloo, satisfying (ii). Thus, with the support of reasoning, the protocol identifies HPLaser2020 as a matching service, even though its description does not match the service request syntactically.

3.4.2 Service Advertisement

Having explained how services are described and classified, we discuss how providers can construct service advertisements to be stored and queried by the discovery component. It is time-consuming to describe a service by constructing an OWL description manually. A tool is required to support service providers in constructing the description and advertisement of their services. Through the tool, providers select the type of service they want to advertise from the hierarchy of service profiles. Once the type has been chosen, the tool queries the shared ontology to retrieve the information the provider is required to fill in, including the service description, capabilities (inputs/outputs), and optional grounding. Once the provider fills in the information and submits the advertisement, it is converted from a web form into an OWL description, which is then stored in the ontology database.

The contextual information about the service can be classified into low-level and derived context. Low-level contextual information is obtained from the service directly and published to the pubsub system without any processing. In contrast, it may be desirable to aggregate the contextual information or process it before its publication, as it may be obtained in a raw format using various low-level sensors. If so, a context processor (a component of the context engine) fetches the raw value, processes it, and finally publishes the resulting values.

As mentioned, the context engine is responsible for maintaining the contextual information and constructing the appropriate pubsub nodes. Figure 3.7 gives an overview of the context registration and publication process. As a first step, services register with the context engine and specify the desired type of contextual information to be published. For example, a printing service can register with the context engine and publish contextual information defined by the PrinterLoad class. This is done by sending an XML message that indicates the desired contextual attribute, which must be a subclass of ServiceContextAttribute (e.g., PrinterLoad), and the URI of the service-profile instance. An example of a registration message is shown in Figure 3.8. Consequently, the context engine creates a new pubsub node using an XML message like the one shown in Figure 3.9[1], a new instance of the chosen ServiceContextAttribute subclass (e.g., HPLaser2020_Load), and stores the address of the new pusub node into the actualValue property of this new instance. Afterwards, the engine sends the address of the pubsub node to the service in an XML message similar to the one shown in Figure 3.10, so it can start publishing the actual value.

3.4.3 Service Request

Service discovery protocols can be classified into two main categories, directory-based and directory-less protocols. In the latter, all communication messages, such

[1]In the figure, due to space limitations, the actual pubsub node is aliased to 3954

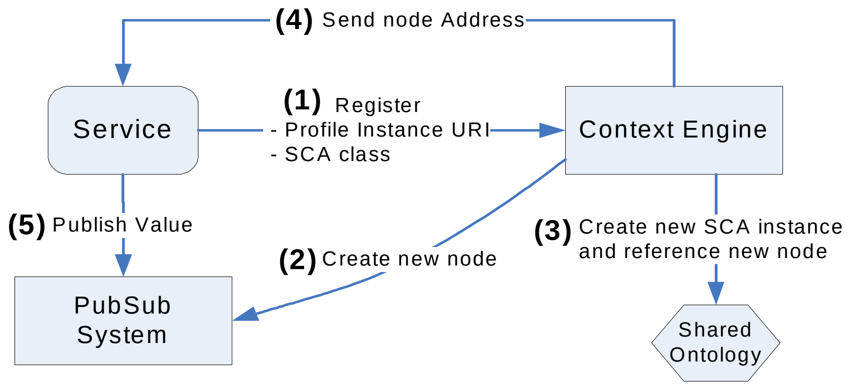


Figure 3.7: Overview of the context registration and publication process

```
<iq type="set" to="engine@otter.uwaterloo.ca/ContextEngine">
<query xmlns="http://impress.com/context-engine">
  <serviceURI>http://otter.uwaterloo.ca/Services.owl#MyPrinter</serviceURI>
  <SCA>http://otter.uwaterloo.ca/Services.owl#PrinterLoad</SCA>
</query>
</iq>
```

Figure 3.8: Context registration request

```
<iq type="set"
from="engine@otter.uwaterloo.ca/ContextEngine"
to="pubsub.otter.uwaterloo.ca"
id="publish2">
<pubsub xmlns="http://jabber.org/protocol/pubsub">
 <create node="home/otter.uwaterloo.ca/engine/3954"/>
</pubsub>
</iq>
```

Figure 3.9: A Jabber message to create a new pubsub node

```
<iq from="engine@otter.uwaterloo.ca/ContextEngine" type="get"
 id="RMTqg-6" to="printer@otter.uwaterloo.ca/Printer" >
<query xmlns="http://impress.com/discover#context">
 <PubSubNode>
  pubsub.otter.uwaterloo.ca/home/otter.uwaterloo.ca/engine/3954
 </PubSubNode>
</query>
</iq>
```

Figure 3.10: Context registration response

as service advertisements and requests, are exchanged between service providers and users directly by means of broadcast or multicast. In contrast, in directory-based schemes, such as the architecture proposed in this thesis, service providers register their services with a particular node (e.g., a Jabber server) or a group of nodes in the network. Afterwards, users can locate the advertised services by browsing or sending service requests (service queries) to the directories. We identify three main requirements for a service request.

- **Simplicity**. The request should be expressed in a simple way. Discovery protocols should not require the user to construct long, complex queries. This responsibility should be placed on the discovery protocol and not the user. For example, assume that service descriptions are stored in a relational database. Users will find it inconvenient and time-consuming to construct complex, nested SQL queries in order to discover services. Ideally, users should be able to specify the desired properties/capabilities of the service using a simple form. The discovery protocol should convert this request message to the corresponding query, which can be long and/or complex.

- **Flexibility**. A service request should be flexible; it should enable the user to search for services by a combination of one or more search criteria. Also, the system should not impose rigid restrictions on the format of the request.

- **Semantic-based**. A service request should be based on semantics and not syntax, since the latter leads to undesirable low recall and precision.

In the proposed architecture, service requests are expressed using a Jabber/XMPP information-query (IQ) message, and sent from the user to the discovery component. As a basic requirement, the request must include the desired category of service, namely the service profile. Optionally, the desired service location and capabilities (inputs/outputs) can be included in the request.

A sample request for a printer service in the University of Waterloo is shown in Figure 3.11. The user, alice@otter.uwaterloo.ca, sends an information-query (IQ) message to the discovery component, which is uniquely identified as discovery@otter.uwaterloo.ca. The request includes the desired profile (Printer) and location (WaterlooUniv). Notice that both are expressed using concepts from the ontology (semantics), rather than keywords (syntax).

To provide a capability-based search functionality, in addition to the location and service profile, users can specify the desired capabilities (inputs/outputs) in the request message, which is usually the case when users search for software-based services. A sample service request for an e-commerce service that takes as an input the concept of a Book and returns as an output the concept of Price is presented in Figure 3.12, where both Book and Price are defined as concepts in the ontology, rather than keywords.

We now discuss the proposed service-request format based on the requirements identified earlier. First, it is simple. Unlike most OWL-S matchmakers, such as [31] and [56], it does not require the user or software agent to construct OWL classes or complex queries written in languages like RDQL [63], SquishQL [42], OWL-QL [16], or SPARQL [69]. Instead, the proposed service-request scheme relies on a simple XML-based request format and places the responsibility for constructing complex queries on the discovery protocol itself. Second, the request format is flexible; it

```
<iq type="set" from="alice@otter.uwaterloo.ca"
    to="discovery@otter.uwaterloo.ca/Discovery">
 <query xmlns="http://impress.com/discover#">
   <profile>http://otter.uwaterloo.ca/Services.owl#Printer</profile>
   <location>http://otter.uwaterloo.ca/Services.owl#WaterlooUniv</location>
 </query>
</iq>
```

Figure 3.11: Sample service-request message

does not restrict users or software agents to search by a specific criteria. The request may include the desired profile only, a combination of a profile and location, a combination of a profile and a set of inputs and/or outputs, or a combination of all attributes. Last, as pointed out earlier, the request is expressed using semantics (ontology classes/instances) rather than syntax (keywords), as can be seen in Figures 3.11 and 3.12.

Software agents can be programmed easily to issue service requests using this XML-based scheme. Human users, on the other hand, require a user-friendly interface that enables them to express their requests in an unobtrusive mechanism. Since our key focus is to design and develop the core functionalities of a discovery architecture in a pervasive computing environment, as ongoing work, with the aid of the Human-Computer Interaction (HCI) concepts and tools, we plan to develop several discovery applications that run on top of the discovery protocol to assist users in specifying/saving their preferences and locating/invoking particular services. These discovery applications (or agents) should provide a user-friendly interface and convert users' requests into the corresponding XML-based service-request messages, to be sent to the discovery component. Figure 3.13 shows the interface of a sample application, PrinterFinder, developed to use the discovery protocol to assist users in locating and exploiting nearby printers. Once the user submits the request, the application converts it into the appropriate XML code and sends it in a Jabber message to the discovery component, which can be located using Jabber's simple,

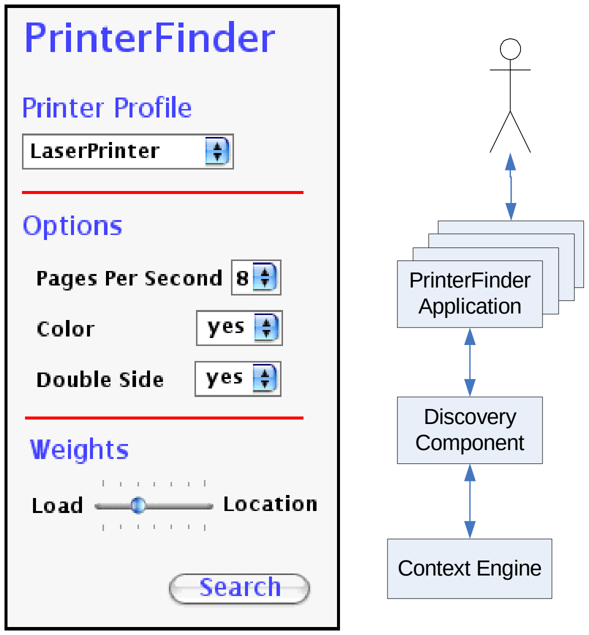
```
<iq type="set" from="alice@otter.uwaterloo.ca"
to="discovery@otter.uwaterloo.ca/Discovery">
<query xmlns="http://impress.com/discover#">
   <profile>http://otter.uwaterloo.ca/Services.owl#Software</profile>
   <inputs>
     <input>http://otter.uwaterloo.ca/Services.owl#Book</input>
   </inputs>
   <outputs>
     <output>http://otter.uwaterloo.ca/Services.owl#Price</output>
   </outputs>
</query>
</iq>
```

Figure 3.12: Capability-based service-request message

built-in discovery protocol. Then, the component parses the request, performs the matchmaking, and sends the results to the discovery application, which in turn, understands the semantics of the matching service, including its invocation information, displays its details to the user in a friendly manner, and enables her to use it accordingly.

3.4.4 Service Matchmaking

Service matchmaking is the process of matching the user's service request against the available service descriptions. In this thesis, the service request is expressed using an XML message, while service descriptions are stored as OWL data in RDF triples. Many languages have been proposed and implemented to query RDF/OWL data, including SquishQL [42], RDQL [63], RQL [33], OWL-QL [16], and SPARQL [69]. The SPARQL query language is based on RDQL and SquishQL. It is considered an enhancement with more features and improvements. It does not impose


PrinterFinder

Printer Profile: LaserPrinter

Options

Pages Per Second: 8

Color: yes

Double Side: yes

Weights

Load — Location

Figure 3.13: Interface and architecture of the PrinterFinder application

rigid constraints on the query as in languages like RQL, and it is expected to become the standard W3C method for querying RDF data. Furthermore, it has stable open-source implementations. For these reasons, we chose SPARQL as a language to query the service descriptions stored in the shared-ontology database.

Once the discovery component receives a request, as a first step, it parses it to determine the requested service-profile. Second, by utilizing the semantics, it expands the request according to the hierarchy of services and locations. For instance, if the user located in WaterlooUniv is looking for a Print service, as in Figure 3.11, the component expands the request to include LaserPrinter and InkJetPrinter services, by exploiting the OWL subClassOf property. Similarly, by exploiting the spatiallySubsumes and spatiallySubsumedBy properties of the GeographicalSpace class, the location WaterlooUniv is expanded to include any space it subsumes (e.g., ShoshinLab, DC3326). Subsequently, the discovery component coordinates with the context engine to retrieve the contextual information about the user (e.g., location and preferences). Currently, as for the user context, we only consider the location. However,

```
 1 PREFIX impress: <http://otter.uwaterloo.ca/Services.owl#>
 2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 3 PREFIX spc:<http://pervasive.semanticweb.org/2004/06/space>
 4 SELECT ?service
 5 WHERE
 6 {
 7   { (?service, rdf:type, impress:Printer) .
 8     (?service, impress:locatedIn, impress:WaterlooUniv)
 9   }
10 UNION
11   { (?service, rdf:type, impress:Printer) .
12     (?service, impress:locatedIn, ?x) .
13     (?x, spc:spatiallySubsumedBy, impress:WaterlooUniv)
14   }
15 }
```

Figure 3.14: A SPARQL query to find matching services

the architecture can be extended to include user preferences and weights.

Once the request has been expanded and the contextual information about the user has been retrieved, as the third step, the component constructs and sends a SPARQL query to the shared-ontology database to retrieve the matching services, according to the search criteria specified in the request. Figure 3.14 shows a sample query to answer the service request presented in Figure 3.11.

The first three lines define aliases for the namespaces used (PREFIX) in the query. As can be seen, the query consists of two main parts, the `SELECT` and `WHERE` clauses, similar to SQL queries. The `SELECT` clause indicates what the query should return. On the other hand, the `WHERE` clause consists of triple patterns to be matched against the triples in the RDF data. Essentially, it specifies the conditions used to filter the results. For example, line 7 specifies that the service must have a type property, which is defined in the RDF ontology, with the Printer class as its range, while lines 12 and 13 together specify that a service must have a locatedIn property with any instance X as a range, where X must have a spatiallySubsumedBy property with WaterlooUniv as its range. Since the type property is defined to be *transitive* and the spatiallySubsumedBy property is defined to be the *inverseOf* spatiallySubsumes, the query result will include all services (profiles) that are either a direct instance of the Printer class, or an instance of a subclass of the Printer class, which are located in or (`UNION`) spatially subsumed by the University of Waterloo. As a result, a list of matching services is returned to the discovery component. Note that OWL does not support reflexive properties yet [67].

To illustrate how software-based service requests are handled, Figure 3.15 shows the SPARQL query used to answer the service request presented in Figure 3.12, which requests services that take as an input the concept of a Book and return the concept of a Price. Recall that in the OWL-S ontology, any parameter has a type, which can be an OWL class or a primitive/custom datatype defined by the XML Schema. Accordingly, the query shown in Figure 3.15 returns instances of the Software profile that take any input parameter X and return any output parameter Y, provided that X's type is Book and Y's type is Price.

A number of OWL-S matchmaking algorithms have been proposed and implemented [31, 39, 56]. Unlike the architecture presented in this thesis, they only consider the functional capabilities (inputs/outputs) described by the OWL-S service profile, and do not incorporate the non-functional aspects, such as service location and category (profile hierarchy), into the matchmaking process. However, they expand the service request based on the subsumption relationships between the requested and advertised inputs/outputs. For instance, assume that the following information is available in the ontology.

(1) BusinessBook is a subclass of Book

(2) Service1 accepts Book as input and returns Price as output

(3) Service2 accepts BusinessBook as input and returns Price as output

(4) Service3 accepts ISBN as input and returns Book as output

```
PREFIX impress: <http://otter.uwaterloo.ca/Services.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?service
WHERE  {
    ?service rdf:type impress:Software .
    ?service impress:hasInput ?x .
    ?x       impress:parameterType "impress:Book"
    ?service impress:hasOutput ?y .
    ?y       impress:parameterType "impress:Price"
}
```

Figure 3.15: A capability-based SPARQL query

(5) Service4 accepts ISBN as input and returns BusinessBook as output

Since BusinessBook is subsumed by Book (a more general concept), if the user is looking for a service that takes Book as an input and returns Price as an output, both Service1 and Service2 will be considered matches. However, if the user is looking for a service that takes BusinessBook as input and returns Price as output, Service1 will not be considered a matching service because its input (Book) may have extra or different attributes compared to BusinessBook. Likewise, if the user is looking for a service that takes ISBN as an input and returns Book as an output, both Service3 and Service4 will be considered matches, as they satisfy the request. However, if BusinessBook were chosen as the output instead of Book, Service3 will not be considered a matching service because its output (Book) violates the expectation of the user. In other words, the user is interested in services that return information about business books only. This example illustrates the functionality and use of expanding a capability-based request based on subsumption relationships. Currently, in the proposed architecture, the service location and category (profile) are expanded. However, the architecture can be extended with the algorithms presented in [31, 39, 56] to support this capability (input/output) expansion functionality.

3.4.5 Dynamic Service Selection and Ranking

Typically, after executing the service request and retrieving information about matching services, discovery protocols send a response message containing the results to the user. Afterwards, the user inspects the result and selects one of the matching services accordingly. If the selected service is not satisfactory, the user re-inspects the result and selects another service, until she finds one that satisfies her requirements. This service selection process is tedious and hard as users might not be knowledgeable enough to differentiate among matching services to select the most suitable one [76]. Motivated by this fact, we incorporate a service selection and ranking mechanism into the proposed architecture to enable the discovery of the most suitable services, and as a result, save users time and effort.

Figure 3.16 presents an overview of the ranking strategy. As the first step, after obtaining a list of matching services, if there is more than one, the discovery component extracts the dynamic contextual information associated with each matching service, by retrieving the polarity and actualValue of every instance of ServiceContextAttribute associated with a matching service, using a SPARQL query such as the one shown in Figure 3.17. As an example, for a matching Print service that publishes the current load of the printer, the discovery component obtains the polarity and sends a query to the context engine, which fetches the actual value from the appropriate pubsub node. Notice that using OWL-S restrictions, it is possible to guarantee that relevant services have the same set of SCAs.

Second, having obtained both the polarity and actual value for every attribute, the discovery component constructs a ranking table for each matching service. The ranking table consists of a list of contextual attributes along with their weight, actual value, and polarity. The weight represents the “importance” of the attribute in the discovery phase. Currently, in order for the architecture to be as unobtrusive as possible, all attributes are weighted equally. However, this can change according to user requirements, which can be stated explicitly in the request message or implicitly through stored preferences. For instance, Alice might be interested in

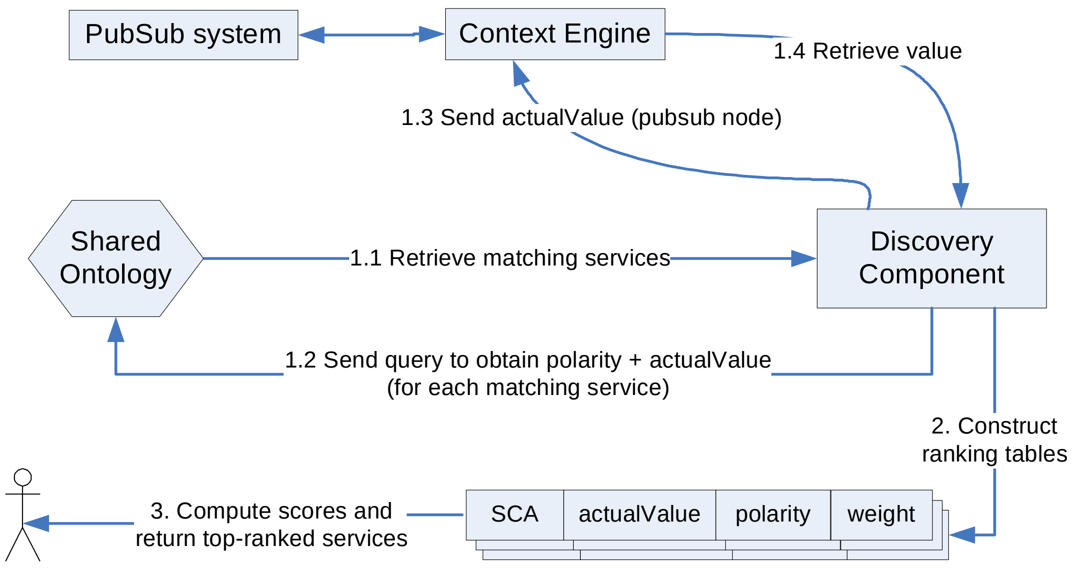

PubSub system
Context Engine
1.4 Retrieve value
1.3 Send actualValue (pubsub node)
Shared Ontology
1.1 Retrieve matching services
Discovery Component
1.2 Send query to obtain polarity + actualValue (for each matching service)
2. Construct ranking tables
3. Compute scores and return top-ranked services
SCA
actualValue
polarity
weight

Figure 3.16: Overview of the ranking strategy

```
PREFIX impress: <http://otter.uwaterloo.ca/Services.owl#>
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl:     <http://www.w3.org/2002/07/owl#>
PREFIX soupa:   <http://pervasive.semanticweb.org/ont/2004/06/space#>
SELECT ?attribute ?pol ?actual
WHERE {
        impress:ServiceURI    impress:hasSCA       ?attribute
        ?attribute            impress:actualValue  ?actual
        ?attribute            impress:polarity     ?pol
       }
```

Figure 3.17: A SPARQL query to obtain the polarity and actualValue

Table 3.1: Ranking table for a printer service

ServiceContextAttribute	actualValue	polarity	weight
PrinterLoad	14	-1	0.50
Distance	21	-1	0.50

Table 3.2: Ranking table for a text translation service

ServiceContextAttribute	actualValue	polarity	weight
QoS	28	+1	1.00

locating the nearest printer regardless of its current load. Her request can be achieved by assigning a higher weight for the location than any other attribute (e.g., load). The proposed architecture can be extended to support such functionality. Example of ranking tables for a printer and a text translation service are shown in Tables 3.1 and 3.2, respectively.

As the third step of the ranking strategy, the discovery component computes a score for each matching service based on its ranking table. Assuming that the total number of contextual attributes for a matching service is n, the polarity of the ith attribute is P_i, the actual value of the ith attribute is V_i, and the weight of the ith attribute is W_i, a score S is computed for each service using the following equation.

$$S = \sum_{i=1}^{n} V_i \times P_i \times W_i$$

For example, the printing service with the contextual attributes presented in Table 3.1 has $S = (14 \times -1 \times 0.50) + (21 \times -1 \times 0.50) = -7 + -10.5 = -17.5$

Notice that some of the contextual attributes can be published directly (low-level context), such as the load on a printer. On the other hand, some attributes might require some sort of processing or aggregation before publication (derived context). For example, the status of a light might be obtained as a string (e.g., "off", "on") and normalized to an integer value (i.e., 0/1). Likewise, the distance between the user and a matching service is not provided directly. It is computed by considering the GPS coordinates of the user and the service, which are captured using instances of the LocationCoordinates class. Also, notice that there could be a significant difference in the magnitudes of the contextual attributes. Thus, to ensure a fair ranking, the actual values of contextual attributes should be normalized and mapped to a scaled value of 0 to 100. This can be achieved using various methods. One approach is to use a linear, Min-Max normalization technique, where the minimum and maximum values of each contextual attribute in a ranking table are identified (e.g., minimum and maximum load among all matching printers is 3 and 40, respectively). Then, based on these Min-Max values, the contextual attributes are normalized to a value between 0 and 100.

After performing the necessary processing, normalization, and computing a score for each matching service, as the final step, the discovery component ranks the services and returns the top ones to the user in a Jabber/XMPP XML message, according to an adjustable threshold. An example is shown in Figure 3.18. In this example, the discovery component located three printing services, computed a score for each based on the location and load, and finally returned the URIs of the top-ranked ones to the user/agent. Algorithm 1 gives a general overview of the service matchmaking and ranking process used in the architecture.

3.4.6 Service Invocation

Discovering services is not enough. An adequate mechanism is required to enable users or software agents to utilize a service once it is discovered and considered suitable. This process of utilization, which involves communication protocols, network addresses and messages, is referred to as service invocation. Current discovery protocols provide three different levels of support for invocation [76]. At the basic level, the protocol provides the user/agent with only the location (network address) of the service. In this case, the responsibility for defining the functional operations and communication scheme is placed on the service. At the next invocation level, the protocol defines the communication scheme for the service (e.g., RPC), in addition to the location of the service, as in Jini, which relies on and defines a Remote

```
<iq from="discovery@otter.uwaterloo.ca/Discovery" type="get"
to="alice@otter.uwaterloo.ca">

<query xmlns="http://impress.com/discovery-results">

 <matching-services>
   <service>http://otter.uwaterloo.ca/Services.owl#HPLaser2020</service>
   <service>http://otter.uwaterloo.ca/Services.owl#HPLaser5000</service>
   <service>http://otter.uwaterloo.ca/Services.owl#MyInko</service>
 </matching-services>

</query>

</iq>
```

Figure 3.18: Top-ranked services returned in an XML message

Algorithm 1 Service matchmaking and ranking

```
1. Receive service-request containing service profile $\boldsymbol{SP}$
2. Expand request (location/profile) and form SPARQL query $\boldsymbol{Q}$
3. Execute $\boldsymbol{Q}$ and retrieve semantically-matching services $\boldsymbol{SMS}$
4. If (number of $\boldsymbol{SMS} > 1$)
  4.1 For each matching service $\boldsymbol{MS}$ in $\boldsymbol{SMS}$
    4.1.2 For each ServiceContextAttribute $\boldsymbol{SCA}$ in $\boldsymbol{MS}$
       4.1.2.1 weight = 1 / #SCA
       4.1.2.2 polarity = $MS.SCA.polarity$
       4.1.2.3 PubSubNode = $MS.SCA.actualValue$
       4.1.2.4 actualValue = getValueFromEngine(PubSubNode)
    4.1.3 Construct ranking table for $\boldsymbol{MS}$
    4.1.4 Compute score for $\boldsymbol{MS}$
5. Order services descendingly and return top-ranked ones.
```

Method Invocation (RMI) scheme. At the third level, in addition to the service location and communication scheme, the protocol defines the functional operations and message formats of the service, as in UPnP, Salutation, and UDDI.

As the OWL-S authors mention, autonomous service invocation refers to "the invocation of a web service by a computer program or agent, given only a declarative description of that service, as opposed to when the agent has been pre-programmed to be able to call that particular service" [64]. To provide a complete discovery architecture, decrease the responsibilities placed on the user/agent, and facilitate autonomous service-invocation, we adopt the third invocation level, where the architecture defines the necessary details to utilize the service, such as its location, underlying communication scheme, and operational messages. These details are captured using the ServiceGrounding concept in the shared-ontology database. Having based our discovery architecture on Jabber/XMPP, we extended the OWL-S ontology to describe the invocation of Jabber-based services, by defining two new classes, AdhocGrounding and SOXGrounding, as subclasses of ServiceGrounding. In the following

```
<iq type="set" to="light@otter.uwaterloo.ca" id="exec1456">

  <command xmlns='http://jabber.org/protocol/commands'
           node='turnOn'
           action='execute'/>

</iq>
```

Figure 3.19: Sample ad-hoc command

sections, we discuss the invocation schemes provided by Jabber/XMPP, along with their advantages and disadvantages, and present the details of these new grounding classes.

3.4.6.1 Jabber Adhoc Commands

The simplest Jabber/XMPP invocation mechanism is a command-based scheme, Jabber Ad-hoc Commands (JAC) [43]. JAC defines a clear protocol that allows Jabber entities to publish, execute, and attach payloads to custom commands. These commands can be a one-time request, or can be executed in multiple stages through a command session. For instance, a Jabber entity can publish a restart and a turnOn command, which can be one-time requests. Likewise, it can publish a configure command, which can be a multi-stage command that, once executed, provides the user with extra parameters to submit (e.g., service type to configure). For the sake of simplicity, we consider one-time commands only, however, the architecture can be easily extended to support multi-stage commands.

When ad-hoc commands are sent to an entity as XML messages defined by JAC, the entity can parse them and take action accordingly. An example of a turnOn ad-hoc command sent to an entity identified as light@otter.uwaterloo.ca is presented in Figure 3.19.

The JAC scheme is suitable and convenient for simple command-based services

```
<iq type="get" from="alice@otter.uwaterloo.ca"
    to="discovery@otter.uwaterloo.ca/Discovery">

<query xmlns="http://impress.com/discover#">
  <profile>http://otter.uwaterloo.ca/Services.owl#Light</profile>
  <location>http://otter.uwaterloo.ca/Services.owl#DC3326</location>
  <inputs>
    <input>http://otter.uwaterloo.ca/Services.owl#DimLight</input>
  </inputs>
</query>
</iq>
```

Figure 3.20: Capability-based request for a light service

that do not require much interaction or input from users, such as a light service. Therefore, we represent the ad-hoc commands associated with a service as its inputs. For instance, a Light service can have a hasInput property with an instance of the Input class (a subclass of Parameter) as its range. This instance has a parameterType property with the TurnLightOn concept as its range. The TurnLightOn concept is a subclass of a new concept in the extended OWL-S ontology, AdhocCommand. This semantic-based description enables the discovery protocol to provide a capability-based search facility, as explained earlier. Figure 3.20 shows a request for a light service located in room DC3326 that supports the DimLight capability (input). Details on how the JAC invocation information is captured semantically are presented in Section 3.4.6.4.

3.4.6.2 Jabber RPC Extension

Jabber/XMPP has a simple RPC extension [1], which defines a technique to transport XML-RPC encoded requests and responses over Jabber/XMPP. The Simple Object Access Protocol (SOAP) is an extension and enhancement of XML-RPC. It is more sophisticated, flexible, and is known as the de facto standard for exchanging structured information in decentralized environments. Because of these facts and the significant semantic coupling required between XML-RPC senders and receivers, Jabber's RPC extension is not used.

3.4.6.3 SOAP Over XMPP

The SOAP Over XMPP (SOX) scheme [17] enables Jabber entities to transport SOAP envelopes through Jabber/XMPP XML messages. Basically, it enables the development of web services that utilize Jabber/XMPP as a transport mechanism instead of HTTP. Generally, SOAP envelopes can be sent and received through HTTP or SMTP messages. Unlike HTTP, Jabber/XMPP is capable of transporting synchronous and asynchronous messages, and unlike SMTP, it transports asynchronous real-time messages in a fast and efficient manner. By using SOX, web services will not require complex protocols, such as WS-Routing [41] and WS-referral [40], to support users behind a firewall, or users without a static, public IP address.

The service-oriented computing paradigm recommends the use of web services for developing software systems. Web services have many advantages, including component reusability, interoperability between applications, open standards and protocols, and ease of distributed integration (loosely coupled structure). For these reasons, the architecture presented in this thesis supports web services that use Jabber/XMPP as a transport mechanism. Normally, the interface of a web service is described using the Web Services Description Language (WSDL) [60], which is an XML-based language that describes the invocation details (e.g., communication protocol, message formats) of a web service, in a syntactic manner. Details on how the SOX invocation information is captured semantically are presented in the next section.

3.4.6.4 AdhocGrounding and SOXGrounding

The SOX scheme supports the invocation of complex software services. It is more flexible and sophisticated than the XML-RPC scheme. Yet, the JAC scheme is more suitable and convenient for simple command-based services that do not require a complex SOAP-based invocation mechanism or much interaction from users. We support both schemes in our discovery architecture by creating two new service-grounding classes, AdhocGrounding and SOXGrounding.

These groundings describe the semantics of the invocation details. They facilitate the invocation of services by providing a mapping from an abstract to a concrete specification of service capabilities. In other words, as described by the OWL-S authors, "the central function of an OWL-S grounding is to show how the (abstract) inputs and outputs of an atomic process are to be realized concretely as messages, which carry those inputs and outputs in some specific transmittable format" [64]. We note that in OWL-S, there are two types of services, one that consists of a single process (atomic), and one that consists of multiple composed processes (composite). For this thesis, we only consider atomic services, as service composition is a complex problem beyond the scope of this thesis.

A description of the AdhocGrounding class is shown in Figure 3.21. This class specifies the address of the Jabber/XMPP entity that provides the service (xmppProvider), the version of the JAC protocol used (jacVersion), and the mapping of the abstract service inputs to their concrete realization within the JAC scheme (AdhocMapping). For clarification, an instance of the AdhocGrounding class associated with a light service-profile is shown in Figure 3.22. As can be seen in the figure, the instance defines the service location, which is the address of the Jabber entity that provides the service, as `xmpp:light@otter.uwaterloo.ca`. It also defines the mapping between the abstract inputs, which are defined using instances of the Input class with parameter types DimLight and TurnLightOn (subclasses of AdhocCommand), and their corresponding concrete ad-hoc commands.

The SOXGrounding class is more complicated than the AdhocGrounding one, since SOAP-based web services are far more complex than command-based JAC services. As mentioned previously, the interface of a web service is described using a WSDL document that captures the invocation details of the service, in a syntactic form. The structure of a WSDL document is shown in Figure 3.23.

The <types> elements indicate the datatypes used by the web service and are described using XML Schema. These datatypes can be primitive (e.g., XSD:int, XSD:string) or custom (e.g., XSD:customType). The <message> elements describe

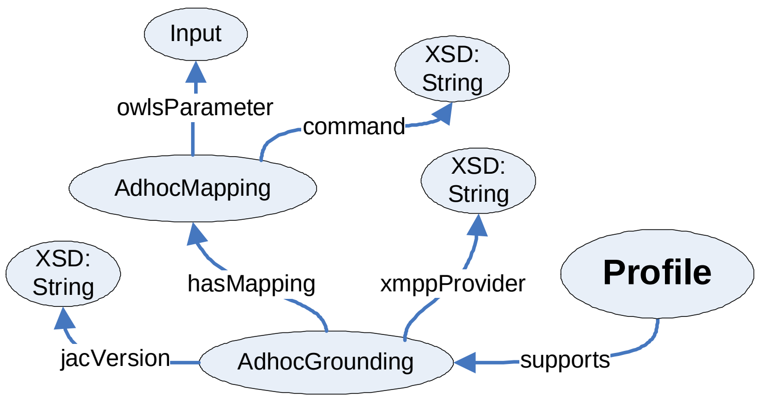


Figure 3.21: The AdhocGrounding class

the messages used in the operations provided by a web service, where a message consist of one or more parts captured using the <part> elements. For instance, a text-translation web service can have two messages. The first message (*msg1*) consists of two parts, the translation mode (*inPart1*) and the text to be translated (*inPart2*), while the second message (*msg2*) consists of one part only, the translated text (*outPart1*). Notice that a part has a name and a type. To indicate the operations that the service provides, the <portType> element is used. It describes the input and output messages used in each operation, which must belong to a certain port. In other words, a service can have multiple portTypes, where every portType has one or more operations. For instance, the text-translation service can have a portType named *translationPortType* with an operation called *translate* that takes *msg1* as an input and returns *msg2* as an output. Binding details, including the transportation mechanism (e.g., Jabber, HTTP, SMTP) used by each portType, are captured using the <binding> elements. Finally, the <service> element describes the service name and location, according to the chosen transport mechanism.

Fortunately, the OWL-S ontology contains a class, WSDLGrounding, which provides a mapping from the abstract OWL inputs/outputs described in the Service-Profile to the concrete capabilities provided by a web service and described using a WSDL document. A WSDLGrounding instance indicates the WSDL version used,

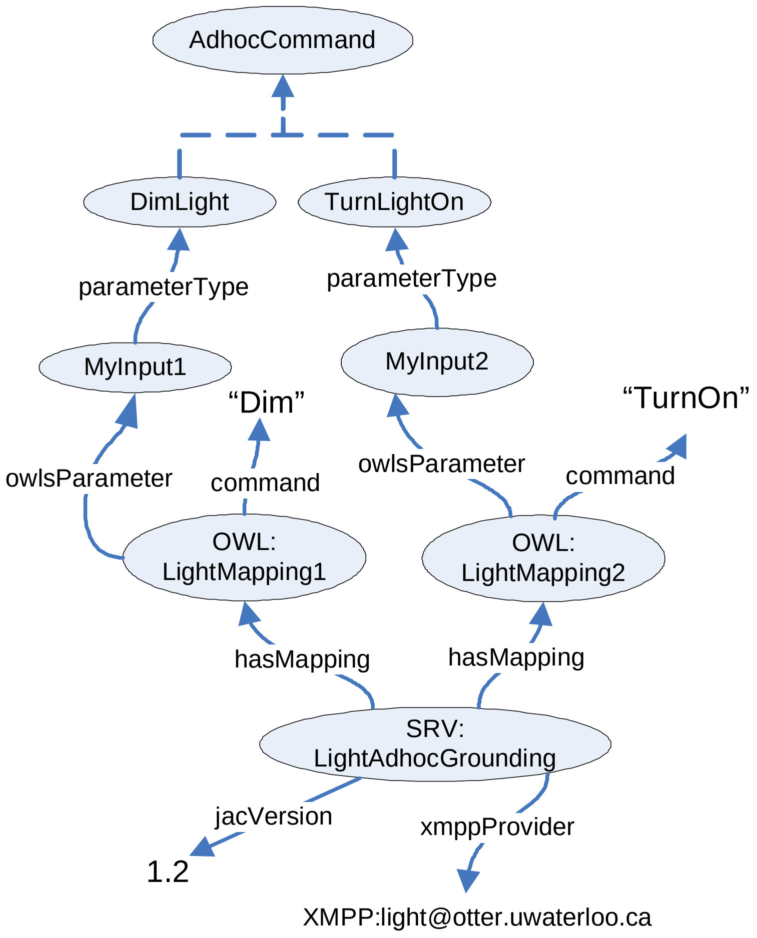


Figure 3.22: Sample AdhocGrounding instance

```
<definitions>

  <types>
     ...types details...
  </types>

  <message>
     <part name=''...'' type=''...''/>
  </message>

  <portType>
    <operation>
         <input message='''' />
         <output message='''' />
    </operation>
  </portType>

  <binding>
    ...binding details...
  </binding>

  <service>
    ...service details...
  </service>

</definitions>
```

Figure 3.23: WSDL document structure

the URI of the WSDL document, the URI of the WSDL operations along with the URI of the service that provides the operations, and finally, a mapping between the conceptual inputs/outputs to the corresponding WSDL input/output message parts. For example, referring to the text-translation service example, assume that the conceptual inputs are represented using two instances of the Input class, InputText and TranslationMode, and one instance of the Output class, TranslatedText. In this case, a WSDLGrounding instance should provide the following mappings.

(i) map InputText to the WSDL URI of *inPart1*

(ii) map TranslationMode to the WSDL URI of *inPart2*

(iii) map TranslatedText to the WSDL URI of *outPart1*.

In addition to these mappings, the grounding instance should capture the following URIs.

(i) URI of the WSDL input and output messages, *msg1* and *msg2*

(ii) URI of the WSDL operation that uses *msg1* and *msg2, transport.*

(iii) URI of the service, WSDL document, and the WSDL version in use.

The SOXGrounding class is similar to the WSDLGrounding class despite some minor changes required to suit our customized and extended version of OWL-S.

It is important to note that the architecture presented in this thesis facilitates autonomous service invocation, since it captures the semantics of the invocation details. However, without a special component that is capable of "understanding" these semantics and invoking the corresponding matching service "on the fly," it still requires human effort to invoke services (manual invocation). The design and development of such a component is beyond the scope of this thesis.

In this chapter, we discussed various aspects of the proposed discovery architecture, including the ontologies used to describe services semantically, how service advertisements are constructed and sent, the mechanism by which services register and publish contextual information, the format and structure of service requests, the details of the matchmaking process used to discover the most suitable services, and the strategy by which services are ranked and returned to the user. Next, we discuss the implementation details of the architecture and the prototype used to assess its feasibility and effectiveness.

Chapter 4

Implementation

4.1 Overview

We have implemented the proposed architecture as well as different services with different invocation schemes to assess its feasibility and effectiveness. In this chapter, various implementation details of our prototype are presented and discussed.

As a pervasive computing platform for our prototype implementation, we use Ejabberd [53], an open-source implementation of Jabber/XMPP, since it is stable and has a strong open-source community due to its wide use. In addition, Ejabberd has many built-in components that provide useful functionalities, such as pubsub, multi-user-chat, and HTTP polling. The entities (e.g., services, users, components) in the environment are addressed uniquely. They exchange information expressed in XML-based messages through a central Ejabberd server, which provides an adequate level of security using protocols such as the Transport Layer Security (TLS) [28], the Simple Authentication and Security Layer (SASL) [27], and the Secure Socket Layer (SSL) [45].

The proposed ontology extensions and additions are constructed using Protégé [57], an open-source ontology editor developed by the medical informatics department at Stanford University. To store and read the shared-ontology data, we use

Jena [22], a Java-based semantic-web framework developed by Hewlett Packard. We use the built-in OWL reasoner in Jena to perform semantic reasoning. Along with Jena, we use the ARQ query engine [20] to query the ontology instances using the SPARQL language. To support knowledge sharing and exchange, the ontology classes and instances are published on the web.[1]

Using the Smack API [32], which is an open-source Java library that provides an implementation of various XMPP functions, we developed the PubSub API to provide basic pubsub functionalities. This API permits Jabber entities to create a new pubsub node, publish values to it, and obtain the value (payload) of a specific node.

The context engine is a Jabber entity based on the Smack API. It relies on the PubSub API to create and fetch the values of pubsub nodes according to the queries and context registration requests. These requests are expressed using custom information-query (IQ) messages, and are sent from the discovery component and services to the engine, which processes them accordingly. An extension to the Smack API was required to support the custom IQ messages.

Similar to the context engine, the discovery component is a Jabber entity based on the Smack API. The discovery component constitutes the core module of the architecture. It is implemented according to the strategies and algorithms presented in this thesis. By extending the Smack API, the component parses service requests expressed in custom IQ messages and sent by Jabber entities. When a request is received, the component processes it and calls a semantic matchmaking algorithm, which coordinates with the context engine and sends SPARQL queries to the shared-ontology database to obtain a list of services that match the request semantically. Finally, the component returns the top matches to the user in a custom IQ message, based on the ranking strategy presented in this thesis.

The XFire software package [15] is used to develop Jabber-based web services that are invoked using the SOX scheme. XFire is a Java-based SOAP framework that supports many transport mechanisms, such as HTTP, Jabber/XMPP and JMS

[1]http://otter.uwaterloo.ca/Services.owl

(Java Messaging Service). JAC-based services, on the other hand, were developed by extending the Smack API to handle ad-hoc command requests and responses.

4.2 Implemented Services

In order to evaluate the feasibility and effectiveness of the architecture, we developed four different services as Jabber components with different invocation schemes.

4.2.1 X10 Light Service

The first service is a hardware-oriented light service developed using Java. This service is capable of controlling a real lamp using an X10 interface [75]. By sending the desired ad-hoc command, which is consequently parsed by the service using the extended Smack API and translated into an equivalent X10 command, the green-colored lamp can be dimmed, turned on, or turned off. The service is composed of two main modules. One module handles authentication and communication with the Jabber server, while the other handles the received ad-hoc commands and sends either the corresponding X10 messages to the lamp or an error message back to the user, depending on the validity of the requested command. Figure 4.1 shows an ad-hoc command sent to the service to turn off the lamp, while the response appears in Figure 4.2.

The description of the service was created using Protégé. The profile of the service is an instance of GreenLight, a subclass of Light, identified as MyGreenLight. It captures the service name, location (DC3326), the supported ad-hoc commands (as instances of the Input class), and the dynamic contextual information (as instances of the LightStatus class) associated with the service. With the support of the engine, the service publishes the status of the light, which can be obtained using the X10 API [72]. An instance of the AdhocGrounding class is used to describe details for invoking the service, including a mapping between the conceptual inputs of the service and its supported ad-hoc commands. Figure 4.3 shows the OWL code

```
<iq type="set" to="light@otter.uwaterloo.ca/Light"
 id="exec1456">

     <command xmlns='http://jabber.org/protocol/commands'
     node='turnOff'
     action='execute'/>

</iq>
```

Figure 4.1: An ad-hoc command to turn off the light

```
<iq from="light@otter.uwaterloo.ca/Light"
type="get" to="alice@otter.uwaterloo.ca" id="xpb7q-8" >

<command xmlns="http://jabber.org/protocol/commands"
   status="completed" node="turnOff"
   sessionid="config:3616Z-700" >

  <note type="info">
        Command has been successfully executed
  </note>

</command>

</iq>
```

Figure 4.2: Response to the turn off ad-hoc command

```
<GreenLight rdf:ID="MyGreenLight">

   <serviceName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
     MyGreenLight</serviceName>

   <hasSCA>
      <LightStatus rdf:ID="MyGreenLight_Status">
        <attrDescription rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >An attribute describing the status of the light</attrDescription>
        <polarity rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</polarity>
        <actualValue rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >pubsub.otter.uwaterloo.ca/home/otter.uwaterloo.ca/engine/
         MyGreenLight/MyGreenLight_Status</actualValue></LightStatus>
    </hasSCA>

    <hasInput rdf:resource="#DimLightInput"/>
    <hasInput rdf:resource="#TurnOnInput"/>
    <hasInput rdf:resource="#TurnOffInput"/>

    <supports>
      <AdhocGrounding rdf:ID="GreenLightGrounding">
        <jacProvider rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        xmpp:light@otter.uwaterloo.ca/Light</jacProvider>
        <hasMapping>
          <AdhocMapping rdf:ID="AdhocMapping_5">
            <command rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
             turn_off</adcommand>
            <owlsParameter>
              <Input rdf:ID="TurnOffInput">
                <parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://otter.uwaterloo.ca/Services.owl#TurnOffLight</parameterType>
              </Input></owlsParameter></AdhocMapping></hasMapping>
        <jacVersion rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        1.2</jacVersion></AdhocGrounding></supports>

    <locatedIn rdf:resource="#DC3326"/>

    <textDescription rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      A service that controls a green light</textDescription>
</GreenLight>
```

Figure 4.3: Light service profile

representing the service profile and grounding instances.

4.2.2 Printing Service

The second service is a software-based printing facility represented as a web service. It accepts the location (URL) of a file/document as an input and sends it to a physical printer in the network. It is invoked using the SOX scheme and can be extended easily to accept the actual file/document as a SOAP attachment rather than a URL. Figure 4.4 shows a print request that consists of a Jabber-based SOAP envelope containing the URL of the file to be printed, while Figure 4.5 shows the response message sent to the user. This service is composed of three main modules. One module manages authentication and communication with the central Jabber server, while another handles the SOAP requests/responses using the XFire API. The third module is the actual implementation of the web service functionalities.

As for dynamic contextual information, the printing service has a load (queue length) that is described using an instance of the PrinterLoad class and published by the service to the pubsub system with the support of the context engine. The actual value of the load can be obtained using the `LPQ` UNIX command or a Simple Network Management Protocol (SNMP) module.

4.2.3 Text Translation Service

The third service, also software-based, is a text translation facility that receives two inputs, a text to be translated (InputText) and a translation mode (TranslationMode), and produces a single output, the translated text (TranslatedText). As mentioned, these are represented using concepts within the ontology to enable capability-based search. Like the printing service, it is developed as a web service, invoked using the SOX scheme, and composed of three main modules. Figure 4.6 shows a request to translate a string from English to French. This request is expressed in a Jabber-based SOAP envelope containing both inputs. Once the service receives it, the service processes it accordingly and returns the result to the user, as shown in Figure 4.7.

```
<iq id="AT988Jq0-13" to="printer@otter.uwaterloo.ca/Printer"
from="alice@otter.uwaterloo.ca" type="get">

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body>
    <m:print xmlns:m="http://otter.uwaterloo.ca/wsdl/printer.wsdl">
    <m:in0>http://otter.uwaterloo.ca/test.ps</m:in0>
    </m:print>
  </soap:Body>
</soap:Envelope>

</iq>
```

Figure 4.4: A SOX-based request to print a document

```
<iq from="printer@otter.uwaterloo.ca/Printer"
type="result" id="AT988Jq0-13" to="alice@otter.uwaterloo.ca/Psi" >
 <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
   <printResponse xmlns="http://otter.uwaterloo.ca/wsdl/printer.wsdl">
   <out>print successful</out>
   </printResponse>
  </soap:Body>
</soap:Envelope>
</iq>
```

Figure 4.5: A SOX response message to a print request

```
<iq id="AT9889Jq0-13" to="translation@otter.uwaterloo.ca/Translation"
from="alice@otter.uwaterloo.ca" type="get">

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soap:Body>
  <m:translate xmlns:m="http://otter.uwaterloo.ca/wsdl/translation.wsdl">
     <m:in0>en-fr</m:in0>
     <m:in1>hello world</m:in1>
   </m:translate>
 </soap:Body>
</soap:Envelope>

</iq>
```

Figure 4.6: A SOX-based translation request

Since this service is software-oriented, we present its dynamic contextual information using an instance of the QoS class. Similar to the previous services, this value is published by the service to a specific pubsub node with the support of the context engine.

As for the invocation details of the service, it is described syntactically using a WSDL document, shown in Figure 4.8, and described semantically using an instance of the SOXGrounding class, which refers to the WSDL document and provides a mapping of the conceptual inputs/outputs to the corresponding WSDL entities, as shown in Figure 4.9. The mapping is very similar to the example presented in

```
<iq from="translation@otter.uwaterloo.ca/Translation" type="result"
id="AT9889Jq0-13" to="alice@otter.uwaterloo.ca" >

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <translateResponse
  xmlns="http://otter.uwaterloo.ca/wsdl/translation.wsdl">
     <out>bonjour tout le monde</out>
  </translateResponse>
 </soap:Body>
</soap:Envelope>

</iq>
```

Figure 4.7: A SOX response message to a translation request

```
<wsdl:definitions
 ........namespace aliases.................
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://otter.uwaterloo.ca/wsdl/translation.wsdl">

  <wsdl:message name="translateRequest">
    <wsdl:part name="mode" type="xsd:string" />
    <wsdl:part name="text" type="xsd:string" />
  </wsdl:message>

  <wsdl:message name="translateResponse">
    <wsdl:part name="out" type="xsd:string" />
  </wsdl:message>

  <wsdl:portType name="TranslationPortType">
    <wsdl:operation name="translate">
      <wsdl:input message="tns:translateRequest" name="translateRequest" />
      <wsdl:output message="tns:translateResponse" name="translateResponse" />
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="TranslationXMPPBinding" type="tns:TranslationPortType">
    <wsdlsoap:binding style="document" transport="http://jabber.org/protocol/soap" />
    <wsdl:operation name="translate">
      <wsdlsoap:operation soapAction="" />
      <wsdl:input name="translateRequest">
        <wsdlsoap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="translateResponse">
        <wsdlsoap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="Translation">
    <wsdl:port binding="tns:TranslationXMPPBinding" name="TranslationXMPPPort">
      <wsdlsoap:address location="translation@otter.uwaterloo.ca/Translation" />
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

Figure 4.8: WSDL document of the translation service

```
<SOXGrounding rdf:ID="JabberTranslationSoxGrounding">

 <wsdlVersion rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">1.1</wsdlVersion>
 <wsdlDocument rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
  http://otter.uwaterloo.ca/wsdl/translation.wsdl</wsdlDocument>
 <wsdlInputMessage rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
     http://otter.uwaterloo.ca/wsdl/translation.wsdl#translateRequest
  </wsdlInputMessage>
  <wsdlOutputMessage rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
     http://otter.uwaterloo.ca/wsdl/translation.wsdl#translateResponse
  </wsdlOutputMessage>
  <wsdlInput>
      <WsdlInputMessageMap rdf:ID="WsdlInputMessageMap_9">
        <owlsParam rdf:resource="#TranslationMode"/>
        <wsdlMessagePart rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
        >http://otter.uwaterloo.ca/wsdl/translation.wsdl#mode
       </wsdlMessagePart>
      </WsdlInputMessageMap>
  </wsdlInput>
  <wsdlInput>
      <WsdlInputMessageMap rdf:ID="WsdlInputMessageMap_8">
        <owlsParam rdf:resource="#InputText"/>
        <wsdlMessagePart rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
        >http://otter.uwaterloo.ca/wsdl/translation.wsdl#text
        </wsdlMessagePart>
      </WsdlInputMessageMap>
  </wsdlInput>
  <wsdlOutput>
      <WsdlOutputMessageMap rdf:ID="WsdlOutputMessageMap_10">
        <owlsParam rdf:resource="#TranslatedText"/>
        <wsdlMessagePart rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
        >http://otter.uwaterloo.ca/wsdl/translation.wsdl#out
        </wsdlMessagePart>
      </WsdlOutputMessageMap>
  </wsdlOutput>
  <wsdlOperation>
      <WsdlOperationRef rdf:ID="WsdlOperationRef_11">
        <operation rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
        >http://otter.uwaterloo.ca/wsdl/translation.wsdl#translate
        </operation>
        <portType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        http://otter.uwaterloo.ca/wsdl/translation.wsdl#TranslationPortType
        </portType>
      </WsdlOperationRef>
    </wsdlOperation>
  <wsdlService rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://otter.uwaterloo.ca/wsdl/translation.wsdl#Translation
  </wsdlService>
  <wsdlPort rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://otter.uwaterloo.ca/wsdl/translation.wsdl#TranslationPortType
  </wsdlPort>

</SOXGrounding>
```

Figure 4.9: The SOXGrounding instance of the translation service

Section 3.4.6.4.

4.2.4 Italian Restaurant Service

The last service is an Italian food restaurant. It is an abstract service with no invocation scheme. It registers with the context engine and publishes contextual information defined by instances of the AvailableTables and AvailableParkingSpots classes.

4.3 Discovery Scenarios

After presenting and discussing the implementation details of the architecture, we present an overview of the services and the prototype implementation in Figure 4.10. To test the effectiveness of the architecture in matchmaking and discovering the most suitable services, we created three copies of the profile instance for each service, with different values of contextual information.

Recall that the location of each service is provided at the time of service advertisement, while the location of the user is assumed to be supplied by location-tracking devices and processed/represented by the context engine. On the other hand, the dynamic contextual information, such as the current status of the light, the queue length of the printer, the number of free tables and parking spots in the restaurant, and the QoS of the translation facility, are published by the services with the support of the context engine. Note that we have not yet developed the discovery applications, which should run on top of the discovery protocol and provide unobtrusive interfaces for users to express their requests. Thus, we constructed the service requests, assuming that they have been formulated by the discovery applications. Likewise, we invoked the services manually, as we have not yet designed and developed the service invocation component.

We tested the discovery protocol with various scenarios. When a user (Alice) located in WaterlooUniv (an instance of GeographicalSpace) requests a Printer service (an instance of Profile) by sending the appropriate XML-based service request, the discovery protocol retrieves the Alice's contextual information (location)

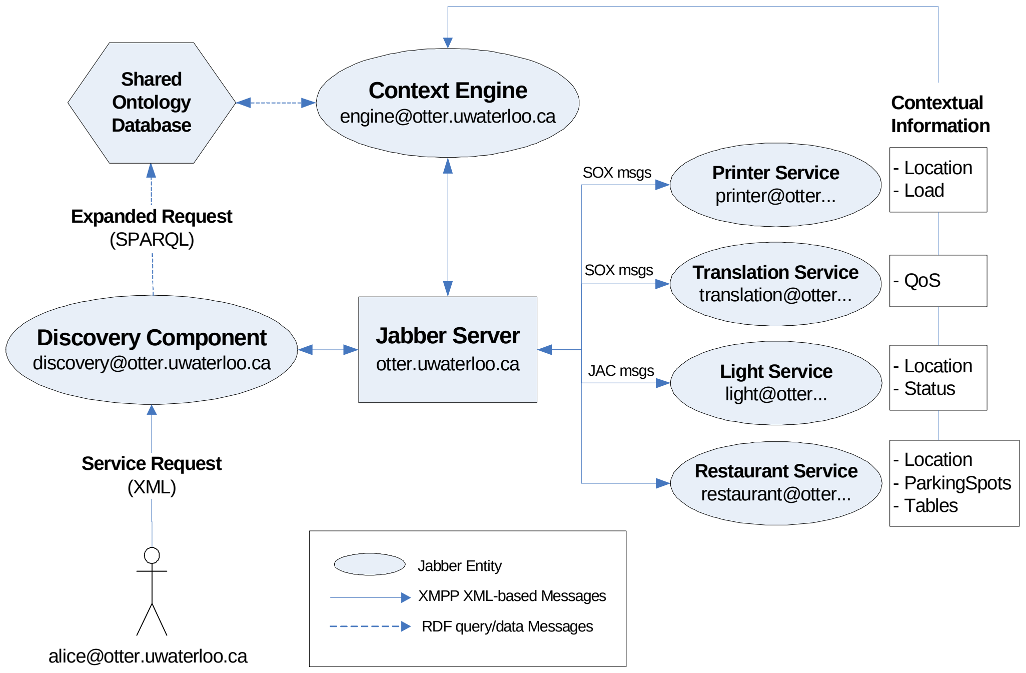


Figure 4.10: Prototype design

with the support of the context engine, expands the request to include LaserPrinter and InkJetPrinter printers that are located in or subsumed by WaterlooUniv (e.g., ShoshinLab, DC3326), and constructs a SPARQL query based on the expanded request to locate matching services. Finally, by ranking the matches according to their dynamic contextual information with the support of the context engine, the protocol identifies the nearest printers with the least load, and sends the URI of the top-ranked ones to the discovery application that Alice is using, in a Jabber message. Through this URI, the application is capable of identifying and "understanding" the invocation details and utilizing the service accordingly.

Similar to the above scenario, when Alice is located in Toronto and she requests a Restaurant service, the discovery protocol retrieves the Alice's contextual information (e.g., location, preferences) with the support of the context engine, expands the request to include ItalianRestaurant profiles that are located in or subsumed by Toronto, and constructs a SPARQL query to locate the matching restaurants. Afterwards, through the ranking strategy, the protocol locates the nearest restaurant with possibly the largest number of available tables and parking spots, and sends Alice the results.

If Alice requests a software service like the text-translation facility, which is advertised without a physical location, using a capability-based service request (e.g, Figure 3.12), the protocol expands the request immediately and converts it into the corresponding SPARQL query to find and return the URIs of the services with the highest QoS value. Then, by inspecting the grounding details, the discovery application can enable Alice to utilize the translation facility accordingly.

For a light service-request, only the location is considered during service ranking, since the status of the light has a polarity of zero. However, we return that information as it might be meaningful for the requesting user.

This chapter has discussed the design and implementation details of our prototype, including the APIs and software components used for the development of the architecture, the details of the implemented services along with their contextual information, and the discovery scenarios used to assess the effectiveness and feasibility of the proposed scheme. Next, we conclude the thesis and discuss our future work plans.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Pervasive or ubiquitous computing is becoming a reality. In that paradigm, service discovery is a fundamental component that enables users or services to request and discover services. Current discovery protocols rely on a syntactic representation of service descriptions and on keyword-based search mechanisms. As a result, when a syntax-based service request is issued, many matching services with a different syntactic representation are not discovered, even though they suit the requirements of the user. Likewise, many irrelevant services are considered matches just because they have a syntactic representation similar to the request. In other words, since they rely on a syntactic representation of information, current protocols suffer from poor precision and recall. Furthermore, they do not incorporate contextual information about the user and services into the discovery phase, and as a result, they are not capable of discovering the most suitable services.

The work presented in this thesis addresses the need for a discovery protocol to support scenarios of pervasive and ubiquitous computing. Through the protocol, using shared ontologies (semantics), software agents or users located in unfamiliar areas are capable of requesting various software- and hardware-oriented services using their handheld devices, and exploiting the most appropriate ones.

The context-aware service-discovery mechanism we propose exploits useful contextual information within pervasive-computing environments to discover the most appropriate and relevant services for the requesting user/agent. We constructed an OWL-based ontology that facilitates context-aware discovery by extending and customizing other ontologies to capture the semantic description of services and contextual information. The proposed discovery scheme relies on this ontology to support knowledge sharing, common understanding, reasoning, capability- rather than keyword-based search, and semantic matchmaking of services. To enhance the overall quality and save users time and effort, the scheme includes a dynamic service-selection mechanism that ranks and filters matching services according to their dynamic contextual information. Our prototype demonstrates the feasibility and effectiveness of the proposed architecture.

5.2 Future Work

As future work, we plan to investigate the following.

- **Large-scale discovery.** The implementation of the architecture proposed in this thesis relies on a central Jabber server. To support large-scale discovery, we plan to investigate the possibility of having multiple Jabber servers, where each server represents a local pervasive environment and has its own context engine and discovery component. Accordingly, these servers can coordinate with each other and exchange context/service information to answer service-requests issued by local and remote users belonging to different environments. For instance, if a local Jabber server fails to find any matches for a service request, it can propogate it to other servers, which might locate a matching service and return its details to the local server. This can be facilitated through a structured peer-to-peer network (i.e., Chord [13], Pastry [50], or CAN [47]), an unstructured peer-to-peer network (e.g., Gnutella [35]), or a directory node that maintains a list of Jabber servers (e.g., LDAP).
- **Discovery preferences.** Currently, in order for the architecture to be as

unobtrusive as possible, the weights of contextual attributes are set equally. We plan to research mechanisms that enable users to specify and store discovery preferences in an easy manner. This includes enabling users to specify the importance (weight) of a specific contextual attribute, and enabling them to specify general preferences. We plan to study the available OWL-based ontologies that model user preferences and incorporate/extend the most suitable ones to achieve the desired discovery goals.

- **Rich service requests**. Currently, service requests are expressed using a flexible, simple, and XML-based Jabber message. It can include a combination of a desired profile, location, inputs, and outputs. We plan to enrich this message format to include custom profile parameters and contextual information to be considered in the matchmaking process. For instance, a user may issue a request for printing services with a PrinterLoad value less than five, a pagesPerSecond value over ten, and a supportsColor instance with a TRUE value.

- **Service advertisement tool.** We have used the Protégé ontology editor to create the profile instances for the prototype services. However, we plan to accomplish this in the future by developing the service advertisement tool as a JSP webpage that queries and stores data in the shared-ontology database using the Jena API.

- **Discovery applications (service request tool**). The discovery architecture presented in this thesis enables users to issue XML-based service requests to find matching services. Software agents can be programmed easily to issue such requests. On the other hand, people require a user-friendly tool to express their service requests and preferences in an unobtrusive manner.

- **Autonomous service invocation.** Recall that the architecture presented in this thesis facilitates autonomous service invocation by capturing the semantics of the service-invocation details. However, a special component is required to let software agents or discovery applications "understand" these semantics and utilize matching services accordingly. Without this component, when a user/agent receives a URI of a matching service, a human effort is required to inspect the invocation scheme and construct the invocation messages.

- **Service notification**. We plan to develop a mechanism that provides users with a service subscription facility as in UPnP. Through this facility, users can subscribe to specific services. Once the description or the status of the service changes, interested user/agents can be notified. Fortunately, the built-in pubsub framework provided by Jabber can be used as a basis for this functionality.

- **Performance evaluation**. Currently, semantic-web development tools are in their infancy. In the future, when the tools become adopted and more stable, we plan to asses the performance of the proposed discovery scheme in terms of the time required to process a service request, perform service matchmaking and ranking, and the effort/time required to utilize Jabber-based services using resource-limited devices.

Bibliography

[1] DJ Adams. JEP-0009: Jabber-RPC, February 2006. URL `http://www.jabber.org/jeps/jep-0009.html`.

[2] Magdalena Balazinska, Hari Balakrishnan, and David Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proceedings of the First International Conference on Pervasive Computing*, pages 195–210, Zürich, Switzerland, August 2002.

[3] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):35–43, May 2001.

[4] James P. Black. The Impress Project, September 2006. URL `http://www.cs.uwaterloo.ca/~jpblack/research/Impress.html`.

[5] James P. Black and Hao Chen. Using Jabber as a Pervasive-Computing Platform for the PUL CogImp Project (Unpublished). October 2005.

[6] Dan Brickley and Libby Miller. FOAF Vocabulary Specification, July 2005. URL `http://xmlns.com/foaf/0.1/`.

[7] Tom Broens, Stanislav Pokraev, Marten van Sinderen, Johan Koolwaaij, and Patricia Dockhorn Costa. Context-aware, Ontology-based, Service Discovery. In *Proceedings of the Second European Symposium on Ambient Intelligence (EUSAI 2004)*, pages 72–83, Eindhoven, The Netherlands, November 2004.

[8] Kevin Chang, Bin He, and Zhen Zhang. Toward Large Scale Integration: Building a MetaQuerier over Databases on the Web. In *Proceedings of the*

Second Conference on Innovative Data Systems Research (CIDR), pages 44–55, Asilomar, California, USA, January 2005.

[9] Harry Chen, Filip Perich, Tim Finin, and Anupam Joshi. SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications, available at http://pervasive.semanticweb.org/ont/2004/06/. In *Proceedings of the International Conference on Mobile and Ubiquitous Systems*, pages 258–267, Massachusetts, USA, August 2004.

[10] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. UDDI 3.0.2 Specification, October 2004. URL `http://uddi.org/`.

[11] Dan Connolly, Frank Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. DAML+OIL Reference Description, December 2001. URL `http://www.w3.org/TR/daml+oil-reference`.

[12] Steve Cuddy, Michael Katchabaw, and Hanan Lutfiyya. Context-Aware Service Selection Based on Dynamic and Static Service Attributes. In *Proceedings of the IEEE International Conference On Wireless and Mobile Computing, Networking and Communications*, pages 13–20, Montreal, Canada, August 2005.

[13] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 81–86, Schloss Elmau, Germany, May 2001.

[14] Anind K. Dey. *Providing Architectural Support for Building Context-Aware Applications.* PhD thesis, College of Computing, Georgia Institute of Technology, November 2000.

[15] Dan Diephouse. XFire, September 2006. URL `http://xfire.codehaus.org`.

[16] Richard Fikes, Pat Hayes, and Ian Horrocks. OWL-QL - A Language for Deductive Query Answering on the Semantic Web. Technical Report 03-14, Knowledge Systems Laboratory, Stanford University, Stanford, California, USA, October 2003.

[17] Fabio Forno and Peter Saint-André. JEP-0072: SOAP Over XMPP, December 2005. URL `http://www.jabber.org/jeps/jep-0072.html`.

[18] Steven D. Gribble, Matt Welsh, J. Robert von Behren, Eric A. Brewer, David E. Culler, N. Borisov, Steven E. Czerwinski, Ramakrishna Gummadi, Jon R. Hill, Anthony D. Joseph, Randy H. Katz, Z. M. Mao, S. Ross, and Ben Y. Zhao. The Ninja Architecture for Robust Internet-scale Systems and Services. *Computer Networks*, 35(4):473–497, March 2001.

[19] Erik Guttman. Service Location Protocol: Automatic Discovery of IP Network Services. *IEEE Internet Computing*, 3(4):71–80, July 1999.

[20] Hewlett Packard. ARQ — A SPARQL Processor for Jena, 2005. URL `http://jena.sourceforge.net/ARQ/`.

[21] Hewlett Packard. Cooltown Project, January 2005. URL `http://www.cooltown.com/cooltown/`.

[22] Hewlett Packard. Jena Semantic Web Framework, 2006. URL `http://jena.sourceforge.net/`.

[23] Joe Hildebrand, Peter Millard, Ryan Eatmon, and Peter Saint-André. JEP-0030: Service Discovery, January 2006. URL `http://www.jabber.org/jeps/jep-0030.html`.

[24] Jerry R. Hobbs. A DAML Ontology of Time, November 2002. URL `Available at http://www.cs.rochester.edu/ferguson/daml/daml-time-nov2002.txt`.

[25] Jerry R. Hobbs. An Ontology of Spatial Relations for the Semantic Web, May 2003. URL `Presentation in the Workshop`

on the Analysis of Geographic References, available at http://gunsight.metacarta.com/kornai/NAACL/WS9/Conf/.

[26] Dan Hong, Dickson Chiu, and Vincent Y. Shen. Requirements Elicitation for the Design of Context-aware Applications in a Ubiquitous Environment. In *Proceedings of the 7th International Conference on Electronic Commerce*, pages 590–596, Xi'an, China, August 2005.

[27] IETF. Simple Authentication and Security Layer (SASL), October 1997. URL http://www.ietf.org/rfc/rfc2222.txt.

[28] IETF. The Transport Layer Security Protocol (TLS), June 2006. URL http://www.ietf.org/internet-drafts/draft-ietf-tls-rfc4346-bis-01.txt.

[29] IETF. Extensible Messaging and Presence Protocol (XMPP): Core, 2004 October. URL http://www.ietf.org/rfc/rfc3920.txt.

[30] Jabber Software Foundation. Jabber Specification. URL http://www.jabber.org.

[31] Michael Jaeger, Gregor Rojec-Goldmann, Christoph Liebetruth, Gero Mühl, and Kurt Geihs. Ranked Matching for Service Descriptions using OWL-S. In *Proceedings of Kummunikation in Verteilten Systemen (KiVS)*, pages 91–102, Kaiserslautern, Germany, February 2005.

[32] Jive Software. The Smack API, June 2006. URL http://www.jivesoftware.org/smack/.

[33] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, and Dimitris Plexousakis. RQL: A Declarative Query Language for RDF. In *Proceedings of the 11th International World Wide Web Conference*, pages 592–603, Honolulu, Hawaii, USA, May 2002.

[34] Omar Zia Khan. Incremental Deployment of Context-Aware Applications. Master's thesis, David R. Cheriton School of Computer Science, University of Waterloo, January 2006.

[35] Patrick Kirk. The Gnutella Protocol Specification, March 2003. URL `http://rfc-gnutella.sourceforge.net`.

[36] Choonhwa Lee and Sumi Helal. Context Attributes: An Approach to Enable Context-awareness for Service Discovery. In *Proceedings of the Symposium on Applications and the Internet*, pages 22–30, Orlando, Florida, USA, January 2003.

[37] Alexander Maedche, Boris Motik, Nuno Silva, and Raphael Volz. MAFRA — A MApping FRAmework for Distributed Ontologies. In *Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management (EKAW)*, pages 235–250, Siquenca, Spain, September 2002.

[38] Paulo Maio, Nuno Bettencourt, Nuno Silva, and João Rocha. Ontology Mapping Negotiation Based on Categorization of Semantic Bridges. In *Proceedings of the International Conference on Knowledge Engineering and Decision Support*, pages 19–27, Lisbon, Portugal, May 2006.

[39] David Martin, Massimo Paolucci, Sheila McIlraith, Mark Burstein, Drew McDermott, Deborah McGuinness, Bijan Parsia, Terry Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia Sycara. Bringing Semantics to Web Services: The OWL-S Approach. In *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, pages 5–21, San Diego, California, USA, July 2004.

[40] Microsoft Corporation. Web Services Referral Protocol (WS-Referral), October 2001. URL `http://www.devx.com/DevX/Link/16185`.

[41] Microsoft Corporation. Web Services Routing Protocol (WS-Routing), October 2001. URL `http://www.devx.com/DevX/Link/16184`.

[42] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. Technical Report HPL-2002-110, Hewlett-Packard Labs, 2002.

[43] Matthew Miller. JEP-0050: Ad-Hoc Commands, June 2005. URL `http://www.jabber.org/jeps/jep-0050.html`.

[44] Soraya Mostefaoui and Beat Hirsbrunner. Context Aware Service Provisioning. In *Proceedings of the IEEE/ACS International Conference on Pervasive Services*, pages 71–80, Beirut, Lebanon, 2004.

[45] Netscape Corporation. The SSL Protocol, November 1996. URL `http://wp.netscape.com/eng/ssl3/draft302.txt`.

[46] Anand Ranganathan, Jalal Al-Muhtadi, Jacob Biehl, Brian Ziebart, Roy Campbell, and Brian Bailey. Towards a Pervasive Computing Benchmark. In *Proceedings of Workshop on Middleware Support for Pervasive Computing (PerWare '05) at the IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, pages 194–198, Kauai Island, Hawaii, USA, March 2005.

[47] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 161–172, San Diego, California, USA, August 2001.

[48] Ricky Robinson and Jadwiga Indulska. Superstring: A Scalable Service Discovery Protocol for the Wide-Area Pervasive Environment. In *Proceedings of the 11th IEEE International Conference on Networks*, pages 699–704, Sydney, Australia, September 2003.

[49] Ricky Robinson and Jadwiga Indulska. A Context-Sensitive Service Discovery Protocol for Mobile Computing Environments. In *Proceedings of the Fourth International Conference on Mobile Business*, pages 565–572, Sydney, Australia, July 2005.

[50] Antony Rowstron and Peter Druschel. Pastry: Scalable Decentralized Object Location and Routing for Large-scale Peer-to-peer Systems. In *Proceedings*

of the IFIP/ACM International Conference on Distributed Systems Platforms, pages 329–350, Heidelberg, Germany, November 2001.

[51] Peter Saint-André. Jabber Enhancement Proposals (JEPs), February 2004. URL `http://www.jabber.org/jeps/jep-0001.html`.

[52] Salutation Consortium. Salutation Architecture Specification, 1999. URL `http://www.salutation.org`.

[53] Alexey Shchepin. Ejabberd Server, 2006. URL `http://ejabberd.jabber.ru/`.

[54] Nuno Silva, Paulo Maio, and João Rocha. An Approach to Ontology Mapping Negotiation. In *Proceedings of the Third International Conference on Knowledge Capture Workshop on Integrating Ontologies*, pages 54–61, Banff, Canada, October 2005.

[55] Evren Sirin. OWL-S API, 2005. URL `http://www.mindswap.org/2004/owl-s/api/`.

[56] Naveen Srinivasan, Massimo Paolucci, and Katia Sycara. Semantic Web Service Discovery in the OWL-S IDE. In *Proceedings of the 39th Annual Hawaii International Conference on System Sciences*, page 109.2, Hawaii, USA, January 2006.

[57] Stanford Medical Informatics. Protege Ontology Editor. URL `http://protege.stanford.edu/`.

[58] Sun Microsystems. Jini Specification, December 2001. URL `http://sun.com/software/jini/specs/jini1.2html/jini-title.html`.

[59] UPnP Forum. UPnP Device Architecture 1.0, December 2003. URL `http://www.upnp.org/resources/documents/CleanUPnPDA10120031202s.pdf`.

[60] W3C. Web Services Description Language (WSDL) 1.1, March 2001. URL `http://www.w3.org/TR/wsdl`.

[61] W3C. DAML-S Ontology, May 2003. URL http://www.daml.org/services/daml-s/0.9/.

[62] W3C. RDF Schema, February 2004. URL http://www.w3.org/TR/rdf-schema/.

[63] W3C. RDQL—A Query Language for RDF, January 2004. URL http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[64] W3C. Web Ontology Language for Services (OWL-S) 1.1, November 2004. URL http://www.daml.org/services/owl-s/1.1/.

[65] W3C. Web Ontology Language (OWL), February 2004. URL http://www.w3.org/TR/owl-features/.

[66] W3C. XML Schema, October 2004. URL http://www.w3.org/XML/Schema.

[67] W3C. Simple part-whole relations in OWL Ontologies, August 2005. URL http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/.

[68] W3C. Resource Description Framework (RDF), June 2006. URL http://www.w3.org/RDF/.

[69] W3C. SPARQL Query Language for RDF, February 2006. URL http://www.w3.org/TR/rdf-sparql-query/.

[70] W3C. The Semantic Web, September 2006. URL http://www.w3.org/2001/sw/.

[71] W3C. XML Namespaces, August 2006. URL http://www.w3.org/TR/REC-xml-names/.

[72] Wade Wassenberg. The Java X10 Project, June 2005. URL http://x10.homelinux.org/.

[73] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265 (3):94–104, September 1991.

[74] Dominic Widdows. Concept Lattices: Binding Everything Loosely Together, September 2005. URL `http://infomap.stanford.edu/book/chapters/chapter8.html`.

[75] X10.com. The X10 Technology, 2005. URL `http://www.x10.com/`.

[76] Feng Zhu, Matt W. Mutka, and Lionel M. Ni. Service Discovery in Pervasive Computing Environments. *Pervasive Computing*, 4(4):81–90, October-December 2005.