CONTEXT-AWARE, UBIQUITOUS SERVICE DISCOVERY AND DELIVERY FOR MOBILE CLIENTS

By

CHOONHWA LEE

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2003
Copyright 2003

by

Choonhwa Lee
To my family
ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Abdelsalam ("Sumi") Helal, for giving me the opportunity to work with him and to study under his guidance. Without his continuous inspiration, support, and encouragement during my years of doctoral study, this research would not be complete. I would also like to offer my sincere appreciation to my supervisory committee members, Dr. Stanley Y. W. Su, Dr. Joachim Hammer, Dr. Michael P. Frank, and Dr. Alan D. George, for their precious time and advice.

This work was supported in part by Motorola, Inc. and by the Department of Education to whom I am grateful.

Also, I appreciate the friendship and encouragement of all the colleagues at the Harris Mobile Networking and Communication Laboratory while working and studying together in the lab. My thanks should also go to many friends who shared all the happy moments and wonderful memories during my stay in Gainesville.

I owe great love to my parents who have always supported and cared for me. I am also grateful to my parents-in-law, my sisters, and my brother for their continuous encouragement. Last, but not least, I would like to thank my wife, Soyoun, and son, Jongmin, for their love and patience throughout my doctorate program.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xi</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Mobile Service Discovery Scenarios</td>
<td>1</td>
</tr>
<tr>
<td>Context-aware Service Discovery</td>
<td>2</td>
</tr>
<tr>
<td>Context-aware Computing</td>
<td>2</td>
</tr>
<tr>
<td>Context-awareness for Service Discovery</td>
<td>3</td>
</tr>
<tr>
<td>Main Contributions</td>
<td>4</td>
</tr>
<tr>
<td>Organization of the Dissertation</td>
<td>7</td>
</tr>
<tr>
<td>2 RELATED RESEARCH</td>
<td>9</td>
</tr>
<tr>
<td>Service Discovery</td>
<td>9</td>
</tr>
<tr>
<td>Service Discovery Protocols</td>
<td>9</td>
</tr>
<tr>
<td>IETF SLP</td>
<td>10</td>
</tr>
<tr>
<td>Sun Jini</td>
<td>11</td>
</tr>
<tr>
<td>Microsoft UPnP</td>
<td>14</td>
</tr>
<tr>
<td>Other service discovery protocols</td>
<td>16</td>
</tr>
<tr>
<td>Comparison of service discovery protocols</td>
<td>17</td>
</tr>
<tr>
<td>Ad hoc Service Discovery</td>
<td>18</td>
</tr>
<tr>
<td>Wide-area Service Discovery</td>
<td>20</td>
</tr>
<tr>
<td>Server Selection Mechanisms</td>
<td>21</td>
</tr>
<tr>
<td>Server-side Approach</td>
<td>21</td>
</tr>
<tr>
<td>Network-side Approach</td>
<td>21</td>
</tr>
<tr>
<td>Client-side Approach</td>
<td>23</td>
</tr>
<tr>
<td>Service Delivery Technologies</td>
<td>23</td>
</tr>
<tr>
<td>Jini ServiceUI project</td>
<td>24</td>
</tr>
<tr>
<td>XML-based approach</td>
<td>24</td>
</tr>
<tr>
<td>Template-based approach</td>
<td>25</td>
</tr>
</tbody>
</table>
3 MOBILE SERVICE DISCOVERY FRAMEWORK ..........................................................26

Overall Architecture ....................................................................................................26
Three-tier Service Discovery Architecture ....................................................................29
  Proximity-based Service Discovery ........................................................................30
  Domain Service Discovery .....................................................................................30
    Advertisement of domain services .....................................................................32
    Domain query routing .......................................................................................33
    Promotion to global services ............................................................................33
  Global Service Discovery ....................................................................................34
Context Awareness Support .......................................................................................36
  Context-aware Service Advertisement .................................................................36
  Context-aware Service Discovery .........................................................................37
  Device Capability Context Information ................................................................38
  User Mobility Support .........................................................................................39

4 CONTEXT-AWARE SERVICE DISCOVERY ............................................................40

Context Attribute and Scenario ..................................................................................41
  Context Attribute ..................................................................................................41
  Context-aware Service Discovery Scenario ..........................................................43
    Guide service .....................................................................................................43
    Movie preview service .....................................................................................44
    Movie theater service .......................................................................................45
    Printer service ..................................................................................................45
    Bus schedule service .......................................................................................46
  Context-Aware Service Discovery Implementation ..................................................46
    Definition of Context Attribute ..........................................................................47
    Context Attribute Processing ............................................................................50
  Implementation of Domain Service Discovery Subsystem Prototype .......................52
    Domain service discovery ................................................................................52
    Domain service advertisement ........................................................................52
    Implementation details ......................................................................................53
  Experiments ...........................................................................................................55
  Context-awareness for Service Discovery .............................................................62

5 GLOBAL SERVICE DISCOVERY PROTOCOL .......................................................64

  Centroid .................................................................................................................64
  Class-based Service Propagation Protocol ...............................................................66
  CSP Query Routing ...............................................................................................71
  Simulation of CSP Protocol ....................................................................................72
    Workload Analysis .............................................................................................72
    Simulation Results .............................................................................................76
6 SERVICE DELIVERY...........................................................................................................81

Thin-client Approach to Service Delivery........................................................................81
μJini Proxy System Architecture.......................................................................................83
μJini Protocol ...................................................................................................................84
Resident Client .................................................................................................................85
VTC Client .........................................................................................................................85
μJini Proxy .........................................................................................................................86
Dispatcher.........................................................................................................................86
Proxy shell.........................................................................................................................87
Jini manager .......................................................................................................................87
Verifier ...............................................................................................................................87
Adapter ..............................................................................................................................88
VTC manager .....................................................................................................................88
MIDlet emulator ..................................................................................................................88
Performance Measurements...............................................................................................91

7 SUMMARY AND FUTURE RESEARCH...........................................................................93

Summary of Contributions ...............................................................................................93
Future Research ..................................................................................................................94

LIST OF REFERENCES.......................................................................................................96

BIOGRAPHICAL SKETCH ...............................................................................................102
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1. Comparison of major service discovery protocols</td>
<td>18</td>
</tr>
<tr>
<td>3-1. Service classification by coverage</td>
<td>27</td>
</tr>
<tr>
<td>4-1. Static attributes versus context attributes</td>
<td>43</td>
</tr>
<tr>
<td>5-1. CD database word length distribution</td>
<td>75</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1. Layers of discovery</td>
<td>3</td>
</tr>
<tr>
<td>1-2. Service Discovery and Delivery Diagram</td>
<td>8</td>
</tr>
<tr>
<td>2-1. Service location protocol</td>
<td>11</td>
</tr>
<tr>
<td>2-2. Jini discovery and lookup protocols</td>
<td>12</td>
</tr>
<tr>
<td>2-3. UPnP Protocol Stack</td>
<td>15</td>
</tr>
<tr>
<td>3-1. Conceptual model of context-aware service discovery system</td>
<td>28</td>
</tr>
<tr>
<td>3-2. An example of domain registry tree</td>
<td>32</td>
</tr>
<tr>
<td>3-3. Integration of domain and global service discovery framework</td>
<td>35</td>
</tr>
<tr>
<td>3-4. Evaluation of context attributes</td>
<td>37</td>
</tr>
<tr>
<td>4-1. Service record and context attribute</td>
<td>41</td>
</tr>
<tr>
<td>4-2. Java interface definitions of local and remote context attribute</td>
<td>47</td>
</tr>
<tr>
<td>4-3. Java class definition of context evaluation result</td>
<td>47</td>
</tr>
<tr>
<td>4-4. Base class definitions for local and remote context attribute</td>
<td>49</td>
</tr>
<tr>
<td>4-5. A sample ServerLoad context attribute</td>
<td>50</td>
</tr>
<tr>
<td>4-6. Context attribute processing</td>
<td>51</td>
</tr>
<tr>
<td>4-7. Reggie implementation architecture</td>
<td>54</td>
</tr>
<tr>
<td>4-8. Jini lookup service performance</td>
<td>57</td>
</tr>
<tr>
<td>4-9. Composition of lookup service CPU time</td>
<td>58</td>
</tr>
<tr>
<td>4-10. Client lookup time</td>
<td>60</td>
</tr>
<tr>
<td>4-11. Service time experienced by client</td>
<td>61</td>
</tr>
</tbody>
</table>
Service discovery and delivery problems have recently been drawing much attention from researchers and practitioners. The SLP, Jini, and UPnP are some of the front-runners in this area. Although they seem to provide a good solution to the problem, there is an unaddressed need for more sophisticated mobile, location- and context-aware service selection and mobile device support for service delivery as well as service discovery.

In this dissertation, we introduce a multitiered mobile service discovery architecture that addresses the dynamics and the added requirements of mobility. We introduce the concept of context attribute as an effective, flexible means to exploit relevant context information during the service discovery process. We also present a Class-based Service Propagation protocol (CSP) for global service discovery and advertisement. The CSP protocol guarantees better overall query performance without requiring excessive resource use. Combined, context attributes and CSP protocol
guarantee the scalability and effectiveness of service discovery in the presence of popular services (services with a very large number of instances). We describe our architecture, concepts, and protocols, and present a performance scalability study of global service discovery.

Service delivery takes place following the service discovery phase to complete the dynamic service acquisition process. The two phases should be bridged by the client device capability information. As a proof-of-concept prototype, we developed the µJini proxy system that allows a service to be discovered by and presented to any class of target devices using a virtual thin-client architecture.
CHAPTER 1
INTRODUCTION

Since *ubiquitous computing* was envisioned by Mark Weiser [Wei91] in the early 1990s, it has been widely accepted as the ultimate technology for the post-PC era. Although it has often been quoted as “pervasive computing,” “invisible computing,” or “anytime, anywhere, any device computing,” depending on people and their orientations, these terms all largely have the same meaning: *computation embedded in the environment is available everywhere to assist users in accomplishing their daily tasks*. Interactions in that ubiquitous world are envisioned to be modeled as service transactions. This includes service discovery, selection, leasing, and rendering. Interactions may also entail the composition of basic services into newer, value-added ones. To realize this vision, an infrastructure should be in place to effectively support service discovery during user mobility and from a wide range of mobile devices.

**Mobile Service Discovery Scenarios**

The very first step toward the mobile and further ubiquitous computing vision is to enable dynamic service discovery. Mobile and wireless computing is permeating the globe, affecting the way we live and conduct business using portable computers such as laptops, personal digital assistants (PDAs), smartphones, and even wearable computers. In becoming mobile, users innocently expect and demand the same computing luxury they used to (and still) enjoy in the fixed computing environment. Unfortunately, network resources (for example, printers, fax machines, and file systems) and applications, collectively called *services*, do not follow the mobile users when they leave their offices
or homes, or when they relocate to another site. Impromptu access to the services in the
new environment is enabled by dynamic service discovery protocols, as illustrated by the
following classical scenarios [Hel03].

• You are sitting in an airport lounge when you realize you need to print a document
  urgently. You use your laptop with a wireless card to search for a nearby printer. You
  get a response from some nearby printer, and fire a print command to that
  printer. Then you walk up to it and pick up your printout.

• You are in a mall in a new city and you feel like having some Chinese food. You
  use your wireless enabled PDA to look for “Chinese Food.” You get back a
  response from the server in some nearby Chinese restaurant. You are allowed to
  reserve a table and also order your Pecking duck before you walk into the
  restaurant 15 minutes later.

Context-aware Service Discovery

Context-aware Computing

In the last decade, enormous research efforts have been concentrated on context-
aware computing, now being considered as yet another key technology to cope with
challenges brought forth by the ubiquitous computing vision. According to Dey et al.,
context is “any information that can be used to characterize the situation of an entity,
where an entity can be a person, place, or physical or computational object” [Dey99a,
p. 114]. A good survey on how context can be beneficial in the ubiquitous environment is
given by Chen et al. [Che00]. The authors classify context into the following four
categories, extending the context definition of Schilit et al. [Sch94].

• Computing context, such as network connectivity, communication costs, and
  communication bandwidth, and nearby resources, such as printers, displays, and
  workstations

• User context, such as the user’s profile, location, people nearby, even the current
  social situation

• Physical context, such as lighting, noise levels, traffic conditions, and temperature

• Time context, such as time of a day, week, month, and season of the year
Enabling context-awareness means to be able to infer users’ situations. By capturing implicit user intentions, it helps to prevent the users from being unnecessarily distracted from their task in the computation-rich environment. Therefore, the benefit of context-awareness becomes more remarkable when it comes to components related to user interaction, including service discovery, delivery, composition, and user interface.

**Context-awareness for Service Discovery**

As in the general context-aware computing research, context information is the key to the ubiquitous service discovery problem. In other words, context-awareness enables the right services to be delivered to the right users (for example, at the right time and place, in the form and delivery mode most appropriate to the mobile device, and so forth). The users’ intention is implicitly understood in the situations in a non-obtrusive manner by the effective interpretation of context. We envision the ubiquitous service discovery process to consist of four phases, captured by the four layers shown in Figure 1-1.

![Figure 1-1. Layers of discovery](image)

The four phases are infrastructure discovery, context discovery, service discovery, and subsequent service delivery phase. The infrastructure discovery cares about lower-level resource discovery, such as system or network resource acquisition. One example is for mobile devices to obtain their IP addresses by stateful DHCP or stateless allocation.
Another example is Bluetooth SDP (Service Discovery Protocol) [Blu99], which is designed to discover lower-level services, for instance, LAN access points within range. Although the context discovery and the service discovery are shown as logically separate layers, our view is that the two layers are closely coupled, as the latter uses the service of the former. Client queries are resolved into references to services, considering their contexts captured by the context discovery layer. Finally, a discovered service is delivered to the client in an appropriate form and mode with regard to the combined context of the service and clients. For instance, a lightweight version of a service can be selected for delivery into a less powerful device, or a thin-client adaptor can be used to deliver a heavyweight service to a resource-poor device [Hel02].

**Main Contributions**

As far as a small, proximity-based service discovery system is concerned, the context-awareness problem has been well addressed by several early research prototypes [Dey99b, Wan92, Wan95], where individual phases are somewhat separately performed. The prototypes have demonstrated particular applications running in small, closed (that is, preprovisioned) environments, where much of the context is primarily derived from the proximity of services and users. These prototypes, however, left other important aspects of the ubiquitous service discovery unaddressed. Most critically, they provided no support for wide-area coverage and did not sufficiently exploit the use of context in dynamic service discovery. Although these limitations are to some extent alleviated in general service discovery frameworks [Gut99a, Sun01b, UPn00], they are not sufficient for the ubiquitous computing vision.

Recognizing this opportunity, we developed a novel approach to service discovery in which we provided support for mobility (local and wide-area coverage) and context
awareness. Our principal goal is to enable services most appropriate for a given context to be discovered by the nomadic or mobile user, in his home network or any foreign roaming network; on a desktop environment or off a mobile device. This goal can be effectively achieved by adopting a new multilayer discovery paradigm that is more suitable for the dynamic and context-sensitive ubiquitous computing world. The four layers of the service discovery process shown in Figure 1-1 need to be formally established and tightly coupled to better serve the users’ needs of dynamic service discovery. Implicitly captured context information plays a pivotal role to glue these layers together and foster their cooperation toward successfully delivering the most appropriate service to the user’s device. To pursue our vision of a multilayer service discovery paradigm, we implemented a prototype of a context-aware service discovery and delivery system, and conducted a simulation study to evaluate a new global service discovery protocol. Our main contributions can be summarized as follows:

Integrated service discovery framework. Several service discovery frameworks already exist that are diversified on grounds of different target devices, target application domains, and underlying networks. One of the most differentiating resultant features among them is their coverage. For example, one discovery protocol is specialized in local services, while another is dedicated to global services. The diversity requires clients who wish to access services anywhere to understand multiple service discovery technologies, but the resource-poor mobile devices cannot afford to host and accommodate multiple heterogeneous and fragmented service discovery protocols. Therefore, we developed a multitiered service discovery architecture that incorporates proximity, domain, and global
service discovery. The clients are provided with a single view of the integrated service
discovery framework.

**Context-aware service discovery.** A superabundance of services in the future
ubiquitous computing world requires that service discovery be refined by means of
context-awareness. Going beyond local areas, the importance of selecting the most
appropriate service(s) gets magnified. This is because users will likely experience a much
broader range of service quality in wide-area networks. Without this screening, the users
with resource-constrained mobile devices will be easily overwhelmed by a large number
of discovered services of unacceptable quality. Current server selection research
addresses this issue (for example, the best service instance selection among identical
service replicas). Current service selection mechanisms, however, require the selection
process to be applied as a second step, occurring after the service discovery phase, in a
service-specific channel separate from the general service discovery network. We have
integrated service discovery with service selection into a single protocol. Our protocol
uses a novel concept we call “context attribute” to enable the distributed (network-based)
and efficient on-the-fly selection. Also, we showed its effectiveness by developing and
demonstrating a prototype of the context-aware service discovery framework.

**Global service discovery protocol.** A global service discovery network is the last
resort, as service search space is gradually expanded (it implies, in general, that proximity
or domain services are preferable to the global ones, as long as they match the query.)
Although a few systems have already been proposed that support wide-area service
discovery, their scalabilities are questionable or they do not address the need of
ubiquitous, dynamic service discovery very well. We developed a new protocol to exploit
the idiosyncrasy of service ubiquity and skewed query pattern in the future. Our protocol ensures efficient resource use by controlling the entire global service advertisement traffic.

**Thin-client-based service delivery.** Context plays a key role in the service delivery phase as well. A service should be delivered in the most appropriate manner to client devices as a continuing process of the service discovery. However, current technologies focus on either the service discovery or delivery problem. For instance, current service discovery protocols, W3C Device Independence WG [Gim01], and W3C CC/PP WG [Kly02] are the separate nonintegrated efforts. We demonstrated the benefits of gluing service discovery and delivery in the course of developing a new service delivery mechanism. Specifically, we have developed a thin-client adaptation architecture to deliver a range of light- to heavyweight services to resource-constrained devices.

**Organization of the Dissertation**

The diagram in Figure 1-2 illustrates entities involved in the dynamic service discovery and delivery scenario. It also indicates the scope of this dissertation. First, a service needs to be advertised to potential clients in local, domain, and further global networks. The advertisement message may contain the service itself and associated descriptive information, including static and context attributes; the context attribute is our new approach to enable context awareness in a service discovery framework. Then, the advertisement is kept by a service directory through which a client can discover desired services. Services cached by the directory are narrowed down to those that satisfy not only a user query (specified explicitly by a client) but also device profile (probably appended to the query by default, that is, no user intervention). Finally, the client accesses discovered services by connecting to corresponding service providers.
This dissertation is organized into the following sections. Chapter 2 surveys related research, such as service discovery protocols, server selection mechanisms, and service delivery methods. Chapter 3 introduces our integrated service discovery architecture. In Chapter 4, we describe how and what context information can be exploited in the course of dynamic service discovery. Context attribute is our proposal to embody context awareness in the service discovery framework. Chapter 5 presents our global service discovery protocol, which achieves efficient services advertisement and discovery in a wide-area network. A simulation study to compare it with previous protocols is also included. Chapter 6 presents a prototype implementation of our thin-client adaptation architecture to deliver a range of light- to heavyweight services to a resource-poor device. Finally, Chapter 7 summarizes the dissertation and suggests future research tasks.
CHAPTER 2
RELATED RESEARCH

Our work on designing and developing a prototype for the context-aware service discovery framework has been influenced by several research projects and emerging technologies. The key technologies that have motivated us to pursue this research are service discovery and subsequent delivery, server selection, and context-aware computing technology. The server selection problem is concerned with selecting service instances of a particular type, for example, selecting the best Web server among identical replicas. In contrast, the service discovery problem focuses on locating instances of all possible service types. One of the major goals of our work is to augment the service discovery framework by incorporating server selection schemes through context information.

Service Discovery

There have recently been enormous research efforts from industry and academia for the service discovery problem. Sun’s Jini [Sun01b], IETF Service Location Protocol (SLP) [Gut99a], Universal Plug and Play (UPnP) [UPn00], and Salutation [Sal99] are some of the front-runners in this area (most of them are for service discovery in local area networks). There are also other protocols designed for small, ad hoc networks as well as wide-area networks.

Service Discovery Protocols

Most service discovery protocols are based on the announce-listen model in which periodic multicast is used for service announcement and discovery [Hel02b]. This soft-
state model is employed for the robustness to failure that is likely to happen in wireless, mobile environments. Starting off from the same goal, they diverge to various protocols with different flavors, features, and audience.

IETF SLP

Service Location Protocol (SLP) [Gut99a, Gut99b] is an effort of the IETF SVRLOC working group to standardize the discovery and selection of network services on IP networks. The SLP defines Service: URL, which specifies a service type, address, and a set of attribute-value pairs. For example, “service:printer:lpr://hostname;ppm=12” is the Service: URL for a line printer service capable of 12 ppm and available on hostname. Using the Service: URLs, a user can browse services available at his site and make use of selected services to perform certain tasks.

As shown in Figure 2-1, there are three agents in SLP: User Agent (UA), Service Agent (SA), and Directory Agent (DA). A UA is a software entity that sends service discovery requests on behalf of a user application. An SA is an entity that advertises service on behalf of a service. As a centralized service information repository, a DA caches advertisements from SAs to respond to requests from UAs later on. An SA advertises itself by registering with DA(s). This registration message contains the URL for the service, lifetime for the service advertisement, and a set of descriptive attributes. The SA should periodically refresh the registration with DA(s) before it expires. The advertisement lifetime is to prevent the network from being left in a transient state, which leads to a robust discovery framework. A UA sends a service request message to a DA to discover the location of a service. Then the DA responds with a service reply message, including the URLs of services matched against the UA’s query. Finally, the UA accesses instances pointed by the returned Service: URLs. It is worth noting that the DA is
optional. In this case, service request messages from UAs are directly sent to SAs listening to the SLP multicast channel. Small networks may be configured to operate in the DA-less mode.

The SLP supports string-based service browsing and query, using service type and descriptive attributes, to help UAs to discover the most appropriate ones. In fact, the service query syntax is a subset of the LDAPv3 search filter, which means better interoperability with other IETF Internet protocols.

**Sun Jini**

The purpose of Jini architecture is to federate groups of devices and software components into a single, dynamic distributed system. Jini technology provides mechanisms for service construction, lookup, communication, and use in a distributed system. The heart of the Jini system is a trio of protocols called *discovery*, *join*, and *lookup* [Sun99]. The *discovery* occurs when a service is looking for a lookup service with
which to register itself. A pair of well-known multicast channels are used during the
discovery process. The join takes place when the service has successfully located a
lookup service and wishes to join it. The lookup occurs when a client needs to locate and
invoke a service described by its interface type (written in the Java programming
language) and possibly other auxiliary attributes. Figure 2-2 illustrates each step of the
interactions among a client, a service provider, and a lookup service in a Jini community.

Figure 2-2. Jini discovery and lookup protocols [Sun99]

Jini connection technology consists of an infrastructure (discovery, join, and
lookup protocol) and a programming model (leasing, transactions, and events), which
address the fundamental issues of how devices connect with each other to form an
impromptu community. These are built on top of Java objects and Java RMI system.
**Lookup service.** Jini lookup service can be viewed as a directory service in that services are discovered and resolved through it. In a Jini community, services register their proxy objects with a lookup service through discovery and join processes, and clients query the lookup service to find services they need. But the Jini lookup service is much more than a simple name server. Clients see a service as an interface, including methods that they will invoke to execute the service, along with associated descriptive attributes. The lookup service maps the interface exposed to the clients to a set of service proxy objects. More specifically, performing a lookup by an interface results in service proxy object(s) being downloaded to the clients, which are actually RMI stubs that can communicate back with the servers. This proxy object enables clients to use the service without knowing anything about it beforehand. Although the service proxy object is a typical scenario of service invocation, that is, accessing services through RMI method invocation, the downloaded service object can be the service itself or a smart object capable of speaking any private communication protocol.

**Leasing.** Access to services in the Jini system is granted on a lease basis: A service is requested for a certain time period and then granted for a negotiated period between the service user and provider. This lease must be renewed before its expiration. Otherwise the resources associated with the service are released. For example, a Jini lookup service grants a lease to a service registration, and the service should continue to renew the lease because a device may leave the community or fail abruptly without having a chance to deregister itself. Against this anomaly, the leasing keeps the Jini system robust and maintenance-free.
Remote events and transactions. Besides the basic service discovery/join and lookup mechanism, Jini supports remote events and transactions that help programmers write distributed programs in reliable and scalable fashion. Remote events enable an object to be notified when desired changes occur in the system. These events can be triggered by newly published services or some state changes of services. For example, a Jini palmtop that registered its interest in printers can be notified by its lookup service when a printer becomes available. Moreover, Jini supports the two-phase commit (2PC) protocol, which makes Jini a perfect framework to build distributed systems where reliability and robustness are otherwise likely to get impaired by partial failures and recovery.

Microsoft UPnP

Universal Plug and Play (UPnP) is an architecture for peer-to-peer network connectivity of intelligent appliances, wireless devices, and PCs of all form factors. In UPnP, there is no central service registry. Services multicast their presence announcements periodically, while control points learn services of interest to them either by passively listening to those advertisements or by actively multicasting discovery request messages searching for devices or services. The advertising message contains a few essential specifics about the service and URLs for more detailed information. By following these URLs, a control point can retrieve XML descriptions that provide detailed information for four subsequent steps: description, control, event, and presentation.

The UPnP’s service discovery protocol is called Simple Service Discovery Protocol (SSDP). As shown in Figure 2-3, it uses HTTP over multicast and unicast UDP which are referred to as HTTPMU and HTTPU, respectively.
According to the latest specification, UPnP features can be summarized in the following five steps that make use of the protocol stack shown in Figure 2-3. These steps are copied verbatim from UPnP specification [UPn00].

1. Discovery: The UPnP discovery protocol is based on SSDP. When a device is added to the network, the device advertises its services to the control points on the network by multicasting advertisement (ssdp:alive) message. Similarly, when a control point is added to the network, the UPnP allows that control point to search for devices of interest on the network by sending out search (ssdp:discover) multicast message. The fundamental exchange in both cases is a discovery message containing a few, essential specifics about the device or one of its services, e.g., its type, identifier, and a pointer to more detailed information.

2. Description: After a control point has discovered a device, the control point still knows very little about the device. For the control point to learn more about the device and its capabilities, or to interact with the device, the control point must retrieve the device’s description from the URL provided by the device in the discovery message. The UPnP description for a device is expressed in XML and includes a list of any embedded devices or service, as well as URLs for control, eventing, and presentation.

3. Control: After a control point has retrieved a description of the device, the control point can send actions to the device’s service(s). To do this, the control point sends a suitable control message to the control URL for the service. Control messages are also expressed in XML using the Simple Object Access Protocol (SOAP). Like function calls, in response to the control message, the service returns any action-specific values.
4. **Eventing:** A UPnP description for a service includes a list of actions the service responds to and a list of variables that model the state of the service at runtime. The service publishes updates when these variables change, and a control point may subscribe to receive this information. The service publishes updates by sending event messages. Event messages contain the names of one or more state variables and the current value of those variables. These messages are also expressed in XML and formatted using the General Event Notification Architecture (GENA).

5. **Presentation:** If a device has a URL for presentation, then the control point can retrieve a page from this URL, load the page into a browser, and depending on the capabilities of the page, allow a user to control the device and/or view device status.

The absence of a service registry attests that UPnP has been designed for small networks. (A control point may cache services available in the network for its own use later on, but it is not a central registry to respond to service queries from others.) It enables seamless proximity networking in addition to control and data transfer among networked devices in the home or office environment.

**Other service discovery protocols**

There are more service discovery protocols worthy of notice, such as Salutation [Sal99], MIT INS (Intentional Naming System) [Adj99], and Bluetooth SDP (Service Discovery Protocol) [Blu99].

The core of the Salutation architecture is a collection of Salutation Managers distributed over the network, which forms a distributed service registry. A service provider registers itself with its local Salutation Manager. When a client asks its local Manager for services, the discovery is performed by the cooperation of the distributed Salutation Managers.

The MIT INS implements a late-binding mechanism to resolve an intentional name, that is, a service name, to network location(s). The INS resolvers form an application-level overlay network throughout which a service announcement propagates. Eventually,
all resolvers see the announcement and keep it to route future requests for the name. A client sends its local INS resolver a message consisting of an intentional name and data. Then the message routing is performed by the INS resolvers according to the service information they keep, hence the late-binding.

Designed for Bluetooth environments, Bluetooth SDP is a protocol to discover lower-level services, for instance, Bluetooth LAN access points within range. It provides a minimal set of service discovery features, and it defines service query and response processes but no service registration process.

**Comparison of service discovery protocols**

The UPnP is designed to accommodate home network or small office network environments, whereas SLP and Jini can cover larger size networks, that is, enterprise networks. This limitation comes from UPnP’s heavy use of multicast, which can be avoided by the service registry in SLP and Jini. The UPnP leverages IP and Web technologies, including TCP/IP, HTTP, and XML. The absence of a central service registry and adopting Web technologies make UPnP more suitable for peer-to-peer network connectivity of intelligent appliances, wireless devices, and PCs. On the opposite side of the coverage lies SLP, which achieves much wider coverage by minimizing the use of multicast through service registry. Also, SLP is well aligned to other Internet technologies in that SLP services are described by the `service: URL` and service queries are expressed by an LDAPv3 search filter.

Each service discovery protocol has chosen different query language: Java class hierarchy and assistant service attribute objects in Jini, text-based (attribute-value pair based) query in SLP, and XML description in case of UPnP. Although there are pros and cons of their design decisions, for example, their query efficiency versus expressiveness,
their service advertisement and query are able to capture only static aspects of context. For example, the *server-load* service attribute can indicate the server machine load at the moment of service announcement. But it cannot do so at the time of query later on because it is dynamically changing. Table 2-1 compares the main features of SLP, Jini, UPnP, and Salutation.

Table 2-1. Comparison of major service discovery protocols

<table>
<thead>
<tr>
<th></th>
<th>SLP</th>
<th>Jini</th>
<th>UPnP</th>
<th>Salutation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main Entities</strong></td>
<td>Directory Agent (DA), Service Agent (SA), and User Agent (UA)</td>
<td>Jini Lookup Service (JLS), client, and service</td>
<td>Control point and devices (services)</td>
<td>Salutation Manager (SLM), Transport Manager (TM), client, and server</td>
</tr>
<tr>
<td><strong>Service Repository</strong></td>
<td>DA</td>
<td>JLS</td>
<td>No</td>
<td>A set of SLMs</td>
</tr>
<tr>
<td><strong>Service Announcement</strong></td>
<td>Service registration</td>
<td>Discovery/Join protocol</td>
<td>Periodic advertisement</td>
<td>Registration with a local SLM</td>
</tr>
<tr>
<td><strong>Service Discovery</strong></td>
<td>Resolved by DA</td>
<td>Resolved by JLS</td>
<td>Active discovery request or passively listening to advertisements</td>
<td>Query to a local SLM is resolved by cooperation among SLMs</td>
</tr>
<tr>
<td><strong>Access to Service</strong></td>
<td>Service type (service protocol and server information)</td>
<td>Service proxy object based on RMI</td>
<td>SOAP action and state variable query</td>
<td>Service session Management</td>
</tr>
<tr>
<td><strong>Service Description and Scoping</strong></td>
<td>Text-based service type and attribute matching</td>
<td>Java interface type and attribute matching</td>
<td>XML description</td>
<td>ISO 8824 ASN.1 description of Service, Functional Unit, and Attributes</td>
</tr>
<tr>
<td><strong>Service Registration Lifetime</strong></td>
<td>Lifetime in service registration message</td>
<td>Leasing mechanism</td>
<td>CACHE-CONTROL header field in alive message</td>
<td>No</td>
</tr>
<tr>
<td><strong>Service grouping</strong></td>
<td>Scope</td>
<td>Group</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Event notification</strong></td>
<td>No</td>
<td>Remote Events</td>
<td>GENA</td>
<td>Availability checking (periodic &amp; automatic)</td>
</tr>
<tr>
<td><strong>Note</strong></td>
<td>Authentication security feature</td>
<td>Java-centric architecture</td>
<td>IP/HTTP/XML based</td>
<td>Transport independence</td>
</tr>
</tbody>
</table>

**Ad hoc Service Discovery**

The extreme of impromptu peer-to-peer connectivity support is found at ad hoc service discovery protocols, such as DEAPspace [Nid01], Service gossip protocol.
[Lee03], and MOCA [Bec99], where each mobile device maintains its own service registry. They facilitate ad hoc collaboration on the spot without any infrastructure support, thereby making them an ideal choice for highly dynamic, small wireless networks.

The IBM DEAPspace [Nid01] addresses the service discovery problem in wireless single-hop ad hoc networks. Each device in DEAPspace maintains a view of all the services present in the network, and periodically exchanges its view of the world, that is, the full list of its service database entries with its neighbors. The periodic broadcast is scheduled in a proactive way: When a device finds its own services absent in messages broadcast by its neighbors or about to expire, it schedules its broadcast sooner than usual. This way it achieves prompt responsiveness to changes in the environments, that is, timely service discovery.

Designed for wireless multihop ad hoc network, the Service gossip protocol [Lee03] attempts to reach the convergence of service information propagation. It does this by multicasting delta messages containing differences between services that a node knows and the ones others seem to know. In other words, each node gossips what it knows, more specifically its knowledge about the services minus the network’s knowledge. So the individual nodes complement one another’s limited knowledge to attain the global view of the whole network. By suppressing repeated gossips of the same information, the incremental advertisement algorithm is able to effectively reduce network traffic for service discovery. The distributed and cooperative mechanism of the Service gossip protocol enables the efficiency and robustness that a discovery system for a highly dynamic ad hoc network requires.
Wide-area Service Discovery

The applicability of the aforementioned service discovery protocols is limited to enterprise size networks because of their adoption of the administratively scoped IP multicast [Mey98]. Wide-area service location is a challenging problem. Not much has been done to extend the utility of service discovery into the global Internet. Even though a few protocols for wide-area service discovery have already been proposed, including the wide-area extension to SLP [Ros97], Berkeley SSDS (Secure Service Discovery Service) [Cze99], and the Centroid [Wei96], none of them provides a clean solution to the problem.

The wide-area extension to SLP, called WASRV, attempts to solve the problem by introducing BA (Brokering Agent) and AA (Advertising Agent). Wide-area services are advertised by AAs on service-specific global multicast channels. It is a BA that caches these advertisements to resolve requests from its local clients. However, its scalability is questionable since the global multicast channel is not available, and, even if available, it would be easily flooded with global advertisement traffic.

Berkeley SSDS [Cze99] supports wide-area service discovery through a hierarchy of SDS servers (that is, their service registries) and lossy aggregation of service descriptions. Service advertisements are aggregated by the Bloom filter [Blo70], as they propagate up along the hierarchy. Although their aggregation method provides good global scalability, the SSDS imposes limitations on service descriptions that are to be aggregated. More specifically, it restricts the number of service attributes to a few to which the aggregation process is applied.

The Centroid, an aggregation method used for Whois++ index service [Fal96, Wei96], is a collection of every word that appears at least once in a service description. It
is propagated up along the hierarchical tree of index servers to guide a query toward services. The Centroid at the top layers eventually converges to a large, fixed set of words to achieve the global scalability. But the convergence will not occur until the very top layer of the tree, which means little gain from the aggregation method.

**Server Selection Mechanisms**

With the explosive growth of the Internet, the server selection problem has become a popular research topic to select the best among replicated servers. To enable properties required for Internet services, such as scalability, load-balance, and fault-tolerance, the servers are usually placed on geographically replicated locations, especially on strategic points of the network. The server selection problem is to select the best replica among those, and the selection should be made transparently to clients, that is, ideally without involving the users with it. Even though, because of the popularity of the Web, most works in this research area are focused on the best Web server selection problem, various approaches show a full spectrum, ranging from server-side to network-side and to client-side approaches in terms of the selection places. With respect to layers where the selection is made, they can be classified as DNS, network routing layer, and application-layer approaches.

**Server-side Approach**

The HTTP redirect [Fie97] is a simple load-balancing mechanism. When a Web server is heavily loaded, client requests are redirected to more available Web servers that provide identical contents using HTTP redirect command.

**Network-side Approach**

According to the DNS round-robin scheme [Bri95], a FQDN (Fully Qualified Domain Name) is mapped to a set of IP addresses, which results in a load-balancing
effect among them. However, a selected IP address does not necessarily mean the most available server since the mapping is performed in round-robin fashion.

DNS name to IP address resolution phase is a perfect place to implement server selection logic since it can be hidden from clients and does not require any change to client programs. Among DNS geographical proximity proposals are SONAR service [Moo98], DNS LOC resource record [Dav96], and GL resource record [Cos01].

Cisco DistributedDirector routers [Cis02] keep track of a set of server replica addresses that provide an identical service. They achieve the geographical locality of service requests by resolving a client request to the IP address of the closest server. The idea of anycasting and its network-layer support was originally proposed by Partridge et al. [Par93], and it was officially adopted by IPv6 [Hin98].

In Application-layer anycast [Bha97], a resolver keeps track of the status of a set of servers. When a client presents a URL to it, it returns the IP address of the best server, based on the current load and network characteristics to the servers.

Although the Application-layer anycast mechanism enables sophisticated server selection, it is likely fine-tuned to a specific type of service. As a result, it lacks generality to be shared by a variety of applications. To overcome this limitation, the server selection schemes are being generalized by recent RSerPool (Reliable Server Pooling) IETF drafts [Tue01a][Tue01b]. A client requests a service by sending its corresponding name to a name resolution server residing somewhere on the network. The name resolver translates the name to IP address(es) and returns the list of appropriate services registered by that name. Then the client connects to one of the returned services. The name-to-service translation and server status monitoring are performed by the
RSerPool framework for reliable server pooling. It is more concerned with common communication-oriented aspects rather than particular application-specific aspects.

**Client-side Approach**

The Smart client [Yos97] provides an interface to a generic networked service. When a client connects to a network service, an applet to mediate an access to the service is downloaded to the client. The applet includes references to mirror sites and connects to the best server based on current load and network latency to replicated servers.

Various server proximity metrics have been proposed to select the nearest server [Dyk00]. The proximity of a server and a client can be defined by network hop-count, geographical proximity, round-trip time, bandwidth, or other network characteristics.

**Service Delivery Technologies**

A wide spectrum of client devices, ranging from tiny, resource-poor devices to powerful workstations, are usually involved in the ubiquitous computing environments. It naturally follows that a dynamically discovered service should be accessible from across diverse target devices, which is often termed as *service mobility*. Furthermore, it must be represented in the most appropriate way for the device that a user happens to use at the moment.

Representing services across an array of user devices, which have heterogeneous rendering capability and interface modality, is no less important than an effective service discovery mechanism. Bickmore et al. summarized four classifications of possible approaches to displaying WWW pages on small screen devices [Bic97]: *device-specific authoring, multiple-device authoring, client-side navigation, and automatic re-authoring*. The device-specific authoring approach is to author a set of Web pages for each target device class. The Jini ServiceUI project [Ven01] can be seen as one example of this
category. In multiple-device authoring, a service GUI is described by some general, intermediate language from which mappings to specific target device classes are defined. “Device Independence Principles” W3C draft [Gim01], “A Device-Independent Representation for Services” [Rom00], and UIML [Abr99] belong to this category. According to the client-side navigation approach, small screen users zoom and pan around a service GUI that does not fit into their displays. Our µJini system, a Jini proxy system presented in Chapter 6, also supports panning of a smartphone display over service GUIs. But our approach is different from others in that client devices are shielded from service implementation details as well as rendering details, as services are executed on the network-side proxy and provided to the client devices via VNC [ATT02]. Finally, the automatic re-authoring approach tries to re-author the document pages for a given target device through a series of transformations.

**Jini ServiceUI project**

A Jini client downloads the front-end of a service implementation that includes the service proxy and GUI. In the Jini ServiceUI project [Ven01], more than one GUI object is attached to a service in the form of service attributes, so that the most appropriate one can be selected, according to the rendering capability of client devices. For example, a Swing GUI implementation would be selected for a desktop user and a Java AWT implementation for a PDA user.

**XML-based approach**

Roman et al. [Rom00] propose a way to represent a service in a device-independent way using XML. In their approach, a service is described by an XML document in a device-independent way, while the device-specific presentation style is specified in an XSL file. This XSL file is associated with a class of client devices to map GUI
components to specific graphical components on the devices. By combining service
descriptions in XML and device-capability in an XSL file, UI objects, for example,
HTML pages, are generated to achieve the device-independent service representation.
The same approach has been taken by W3C for device-independent Web access and
single authoring [Gim01]. Similar to this, UIML [Abr99] is an appliance-independent UI
language, which uses style sheets to map user interfaces described in XML to appliance-
specific markup languages.

**Template-based approach**

Icrafter [Pon01] proposes an infrastructure to enable service UI selection,
generation, and/or adaptation, based on service and device capability. It uses abstract,
declarative markup languages, such as HTML, VoiceXML, MoDAL, and SUIML for
GUI representation. When a service description is discovered, the most suitable service
template is selected from potential candidates out of a template database. From this
template, a UI object is created based on the service description and devices information.
CHAPTER 3
MOBILE SERVICE DISCOVERY FRAMEWORK

In the era of ubiquitous computing, computing devices are everywhere. They are interconnected through a ubiquitous computing infrastructure to help users perform their everyday tasks. Interactions between them are modeled as services. Each of those devices provides services to others so that more value-added services can be composed from primitive ones. In short, our vision is that ubiquitous computing will flourish by enabling the scalable and manageable explosion of services, and therefore the effective service discovery framework is the first step toward this vision.

Overall Architecture

One of the most important criteria to be met by the effective service discovery framework is intelligence to provide the most appropriate service to mobile users by exploiting any meaningful contextual information. User mobility commands support for wide-area service discovery as well as local service discovery, which sets our horizontal coverage to be the Internet. With respect to the layers our proposed architecture relates to, its focus is placed on lower layers, for example, nonapplication-specific context, such as user location, server load, device capability, and network condition. In contrast, Web service discovery is more concerned with high-level, application-related aspects, for example, finding a supplier offering the best price.

Table 3-1 shows three categories of services according to their intended coverage: proximity service, domain service, and global service. The proximity services mean services found through the closest service registry, while the domain services are ones
discovered within an administrative domain boundary. The global services include any other services from the Internet. Notice that the relevant context for service discovery and selection varies across the three classes. In the case of proximity services, physical location or distance will likely be of a primary concern to clients, while communication or computation-oriented context is usually key to effective global service discovery.

Table 3-1. Service classification by coverage

<table>
<thead>
<tr>
<th>Service class</th>
<th>Coverage</th>
<th>Primary selection criteria</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proximity</td>
<td>Local area</td>
<td>Proximity</td>
<td>Nearby printer</td>
</tr>
<tr>
<td>service</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domain</td>
<td>Domain</td>
<td>Proximity and service features</td>
<td>Area guide service</td>
</tr>
<tr>
<td>service</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global</td>
<td>Global area</td>
<td>Service quality, user memory, or availability</td>
<td>Media service</td>
</tr>
<tr>
<td>service</td>
<td></td>
<td></td>
<td>Storage service</td>
</tr>
</tbody>
</table>

As users search for services, their queries would end up with much more services than they can manage or need except for the proximity service case. This is because in the future ubiquitous computing world where services are to be discovered from anywhere, there would be an abundance of matched service instances beyond local areas. We argue that this deluge of services must be dealt with by the infrastructure since user devices’ limited resources will be easily overwhelmed by those services. The users first connect to a nearby proximity service registry to make their service discovery requests. As described in detail later, an important component of the registry is BA (Brokering Agent) that functions as a broker to match and recommend appropriate services among a possibly large set of service instances. We note that the proximity service registry is the base-level component of our discovery system, which is a contact point for the users. Domain service registries are organized in a hierarchical fashion for beyond-the-
proximity service discovery. One node of the domain hierarchy (typically the root node) is designated as the GSR (*Global Service Registry*) node that comprises a global hierarchical tree. It is an entry point to the global hierarchy through which some proximity/domain services are exported to the globe and global services may be imported for local clients. In other words, it is a gateway to a wide-area service discovery network.

Figure 3-1. Conceptual model of context-aware service discovery system

Figure 3-1 shows the overall view of our context-aware, integrated service discovery architecture. Proximity services are closest to a user in the figure, indicating, if there exist services that satisfy the user’s query, that they are the most likely candidate services, unless explicitly specified otherwise. Domain services would be the next choice, in case the query is not satisfied by services from the proximity. And finally the global service discovery tree is explored. The design decision of the three-tier architecture has
been made from our understanding that the primary, relevant context in the three classes is largely determined by the coverage. However, the three discrete classes are hidden from the user, since he interacts with our service discovery framework via the BA. Factors concerning the discovery include various contexts, such as location, network condition, server computing load, user device profile, user service profile, and user-specified query. It is also shown that some of the proximity and domain services may be promoted to global services through a GSR node.

Our architecture is refined by adding context-awareness support to the multitier service discovery framework. Context-awareness is enabled via context attribute evaluation by the BA. The context attribute is a special kind of service attribute, as described in Chapter 4. It provides a flexible, general way to exploit relevant context information, including spatial and temporal context of all parties involved, such as clients, servers, and network condition. In short, the framework makes use of various but relevant contextual information during the discovery process for qualified discovery results.

**Three-tier Service Discovery Architecture**

Suppose that a mobile user visiting a new place needs a service on that local network. First, the user will attempt to discover it using a small-area service discovery protocol such as Bluetooth SDP, since services found nearby are likely to be superior to distant ones in most cases. If not found, the user has to search bigger areas and may end up with a global discovery network as a final resort. Currently, little well-established infrastructure support exists for this expanding discovery scenario. That is, the user is being left directly exposed to more than one service discovery protocol. He must deal with multiple discovery protocols, which is not usually possible on small mobile devices.
Recognizing this problem, we have developed a system architecture that provides a unified view of the multiple service discoveries. Our service discovery subsystems for each coverage class are described below.

**Proximity-based Service Discovery**

A constituent element of our discovery system is the proximity-based service discovery framework that typically consists of a service registry, services, and clients. They are within the reach of one another, and service discovery activities are centered on the service registry.

Based on general service-repository-based discovery architecture, our model introduces a brokering entity called a BA, which is added to the service registry. It can be either a component of the registry or a separate but co-located entity. The BA handles inter-registry communication and context attribute evaluation. The former is for domain and global service discovery, as further explained in the following sections, and the latter is to filter disqualified services considering relevant context. Since the BA takes care of all of these, the client does not need to know what is happening behind the scene, while benefiting from seamless discovery and context-awareness.

Finally, we note that the proximity service registry becomes a node of the hierarchy of domain service registries.

**Domain Service Discovery**

Although a proximal community is often self-sufficing so the need for services may be satisfied by the community itself, that is, a group of service providers in close vicinity, it is not always the case; nonlocal services may be either necessary or more desirable in some cases. As an example, assume that services in a room area are registered with the room’s service registry. A printer in the next room might be the choice of a visitor in case...
there is no printer service in the current room. This example clearly illustrates the need of intercommunication of service registries.

The coverage limitation of the proximity-based service discovery can be overcome by forming a hierarchy of service registries, which is typically based on location context [Jos99]. An administrative authority is usually in charge of setting up the hierarchy of service registries within its jurisdiction. More specifically, each site maintains a service registry serving the area, and the registry becomes a child of the parent registry whose jurisdiction subsumes the child’s. A portion of local services may propagate up along the tree, thereby being advertised beyond their home registry’s coverage to a larger organizational domain or area. All the services need to do for the upward propagation is to register themselves with their service registry node and to declare themselves as a domain service. On a discovery request from clients, a local proximity service registry is first searched. Its parent registry is then tried, which is repeated until the search request reaches the root registry for the administration domain. By this search scheme, some contextual information related to the client and desired services may lose accuracy to a certain extent as it travels up beyond their origin sites. But this incremental search scheme ensures that discovered services will be the best among services available in the client situations, if they are not absolutely ideal ones.

Figure 3-2 illustrates a sample hierarchy of on-campus service registries. Suppose that each (sub-)domain has an area guide service for visitors. When a visitor on the fourth floor of the CSE building makes a discovery request for a guide service, it is implied from his location context that he seeks a floor map service for the fourth floor, unless stated otherwise. If available, it would be able to provide the best guidance service. A
CSE building guide service found in the parent node may otherwise be somewhat helpful.
The last resort would be the campus node. A service discovered there may not provide
any map for the fourth floor, but some other useful information regarding the CSE
building and its neighborhood may be provided. This way the accuracy of context
information is gradually lost as it gets away from the origin in which the request was
made.

These kinds of services are termed *domain services* in that they are intended to be
visible within a certain domain. Service advertisement and query routing for them should
be properly handled by the tree itself to provide the view of domain coverage.

![Figure 3-2. An example of a domain registry tree](image)

**Advertisement of domain services**

Once a service provider indicates that it is intended to be used by clients within a
domain, the service advertisement is propagated up the domain registry tree. It continues
up to the root, unless it is prohibited by the evaluation of the *context attribute*, as
described in Chapter 4. The service may be published to the world at the root node, if
allowed, as explained in the “Promotion to global services” section. Suppose that a guide service is being registered with the fourth floor service registry in Figure 3-2, and it is marked as a domain service. (It could be a global service. In this case, it is handled in the same way as domain service propagation.) The advertisement will be forwarded to the campus registry via the CSE Bld registry. Every registry on the path from the fourth floor to the campus registry caches it for the purpose of resolving queries from local clients or its sub-tree later on. One may raise a concern that upper nodes will be flooded with services from lower nodes. But it would not be so severe since only a portion of services is to be designated as domain or global services. More importantly, we expect the domain size to be kept to a proper extent. A huge domain will be divided into several manageable subdomains to be beneficial from the incremental domain search scenario.

**Domain query routing**

Service query routing is straightforward since a query will run into services on its way upwards that have been cached as a result of prior domain (and global) service advertisements. A service registry forwards to its parent the queries that are not resolved by itself. Services discovered from upper nodes are copied to the client registry which will finally return them to the clients. The cached services can then be shared among other local clients. Also, it makes the clients see a uniform interface regardless of whether the services come from a proximity, domain, or global discovery network.

**Promotion to global services**

Both domain and global service advertisements propagate toward the root node of the domain registry hierarchy, starting off a base proximity registry. It is further exported to a global discovery network from the root, if it is marked to be a global service. This promotion is handled by a GSR entity residing at the root node. To the domain, it is
viewed as a proxy that publishes services to a global area on its behalf. Yet another important reason to distinguish domain and global services in our architecture is service mobility. Since service itself could also be mobile, service mobility across service registries will cause frequent service advertisement updates. Our domain hierarchy (and GSR entity together) contains the frequent updates by intradomain movements within the domain, which generates no update traffic in a global service discovery network. The GSR entity also translates service descriptions between the two hierarchies if they employed different languages for service description.

**Global Service Discovery**

Previous efforts for wide-area service discovery can be classified into two groups. The first approach is to extend the general service discovery protocol to build a certain form of a global discovery network. Berkeley Secure SDS [Cze99] and SLP WASRV [Ros97] fall into the first group. We have developed a new global service discovery protocol, the Class-based Service Propagation (CSP) protocol, that belongs to this group. One basic assumption is that there would be a morass of services everywhere in the future ubiquitous computing world, and general queries looking for anonymous services with certain functionalities would predominate over specific queries (for example, a general query specified in popular terms versus a specific query with unique service id). Accordingly, the emphasis is placed on the aggregation ratio of service information, which ensures the improvement of overall system performance. A detailed description of the CSP protocol is given in Chapter 5. The second group builds on the existing directory systems, whether centralized or distributed, which includes UDDI [UDD02] for Web services, an LDAP-based system such as Jipang [Suz01], and USDP [Bis01]. Although their architecture can rely on the strength of the existing directory systems, for example,
LDAP [Hod02b], service replication and update protocol among directories are yet to be defined and evaluated.

Figure 3-3 illustrates how the components described thus far are threaded together to provide a single, unified view of the disintegrated service discovery hierarchies. We note that a domain hierarchy comprises base proximity service registries, while the global hierarchy is made up of root nodes of domain trees. Global services are advertised into the global discovery network by GSR-capable domain roots. The GSR-capable root means a root node augmented with a GSR entity to import and export global services on behalf of the domain. Although a GSR entity is typically hosted by the domain root, multiple GSR entities can be deployed at strategic points of the hierarchy where the demand for global services is high, especially for big domain networks.

Figure 3-3. Integration of domain and global service discovery framework
Every service registry is basically BA-capable, meaning that it is capable of sensing and processing relevant context to refine service discovery results. The figure also indicates that a user interacts with the closest service registry through a UA, and services may propagate up the domain hierarchy or even up along the global hierarchy.

**Context Awareness Support**

As previously discussed, context capturing and processing should be handled by a service discovery infrastructure without involving mobile devices and users. A context attribute, which is a viable solution to the context-awareness problem, is a special kind of attribute that is part of service announcement messages. It is also called a dynamic attribute in that its actual value is dynamically determined at the time of evaluation, compared to a static attribute that has a fixed value, as in current service discovery protocols. Containing logic to capture context information specific to a particular service type, the context attribute can be used for the purpose of service advertisement as well as service discovery.

**Context-aware Service Advertisement**

Context attributes can be used to indicate service providers’ intentions as to the boundary to which their services are to be advertised. This type of context attributes is evaluated at the time of service registration. On registration requests from a service, our domain service registry passes the service advertisement on to the parent registry, if it has an attached advertisement context attribute. Depending on the evaluation result by the parent, the service may be further propagated or dropped at that point. In other words, the context attributes prescribe preconditions that must be met to further propagate. This way service providers can control the reach of their service advertisements so that their resources can be dedicated to the targeted clients by avoiding being disturbed by
unintended customers. Otherwise, clients cannot find that the services are not for them until they are somehow rejected by service access control logic.

**Context-aware Service Discovery**

In a domain or especially in a global service discovery subsystem, it is likely for a site to be inundated with a large number of service instances of the same type from the globe. Although they can first be screened by user-specified static attribute matching, a user query will end up with still many unmanageable candidate instances. Therefore, it is imperative that a service discovery framework be able to support an intelligent brokering mechanism for further winnow. The need is amplified in the case of resource-constrained mobile devices since it is too painful for such devices to go through all candidate services to find usable ones.

Figure 3-4. Evaluation of context attributes
Figure 3-4 shows how the context attributes are used in connection with a global service discovery scenario. At a server site, a GSR entity responsible for global service import and export publishes services to a wide-area discovery network. The advertisement messages include text-based service descriptions, such as service type, server address, and static attributes. On the other end, they are retrieved by the client-side GSR. Upon service requests from clients through their UA, the associated BA searches the global discovery network for services satisfying a static-attribute-based query made by the clients. This search is performed through the GSR. Using candidate service information by the static attribute matching, the BA downloads associated objects, such as service proxy and context attribute objects, from source GSRs. Then the candidates are ranked and further screened through the evaluation of associated context attributes. Since this two-step discovery by static attributes and then context attributes introduces non-negligent cost, the downloaded objects (and evaluation result) are cached in GSRs for future requests from other clients at the site. We note that context attributes for domain services are processed in much the same way.

**Device Capability Context Information**

A full array of devices, including powerful workstations, PCs, PDAs, smartphones, and wearable computers, usually lives in the mobile computing environments. It enriches services available to mobile users but, at the same time, their disparate capability brings a new challenge to light: Mobile users should be able to use any type of devices that they happen to carry at the moment.

Client device capability is yet another invaluable context information to be considered in the process of mobile service discovery. The device capability is presented to the client BA through his UA along with service queries. The device capability
description may include device classes, computing and communication resources, and I/O
devices. Then the BA attempts to match the capability as well as the query itself. As a
result, mobile users may see a different set of service instances even with the same
service query, as they switch to a different device.

The importance and effectiveness of device capability, as context information for
successful service discovery and delivery, are attested by our prototype of a thin-client-
based service delivery system presented in Chapter 6.

**User Mobility Support**

Mobile clients use their desktop computers at work, car navigation systems while
driving, and smartphones while walking. They will expect virtually the same set of their
favorite services, regardless of wherever they are with whatever devices. But they must
not be bothered to specify all the detailed query terms to make a discovery of services
that they once used. Our *user service profile* enables this service mobility. It consists of a
set of frequently used services (that is, an active service set) and associated rules to re-
evaluate the list in a foreign network. The users’ UA re-evaluates the service list
downloaded from their home profile service by the help of the associated BA to ensure
that they are presented with the best service instances available in the new environments.
Although existing service discovery protocols have adopted different service description models, their service advertisements and queries are largely limited to capture only static aspects of context. For example, a *server load* attribute can indicate the server machine load at the moment of service announcement but not at the time of service query later on because it is dynamically changing. Service matching, based on the declarative, static service descriptions, provides minimal service discovery and filtering, leaving the rest of the work to users’ manual selection. Users have to blindly try discovered candidate instances one-by-one until they find an instance with satisfactory quality of service. It would be a painful process if their mobile devices do not have enough computing and network resources needed for the manual selection. The problem gets even worse when a global service discovery network is searched. It is quite possible for them to be stormed by an unmanageably large number of service instances returned as a result of their queries, most of which turn out to be useless.

Existing service discovery frameworks do not sufficiently exploit context information for dynamic service discovery. Some information, such as service quality involving several factors, is inappropriate or impossible to be handled by the static, declarative service descriptions. For example, it is difficult for the static descriptions to express how far away service providers and clients are. How can the discovery framework tell if a particular service instance is better than any others in a particular location at the moment?
We introduce the concept of context attribute to incorporate the context-awareness into a general service discovery framework. It allows us to benefit from implicitly captured context information which is currently underutilized.

**Context Attribute and Scenario**

**Context Attribute**

The *context attribute* is a special kind of attribute that is part of service announcements. It is also called a *dynamic attribute* in that its actual value is dynamically determined at the time of lookups, compared to a *static attribute* that keeps a fixed value set at the time of announcements. Figure 4-1 shows the context attribute for a Jini-based context-aware service discovery system.

<table>
<thead>
<tr>
<th>service proxy object</th>
<th>static attribute</th>
<th>static attribute</th>
<th>...</th>
<th>context attribute</th>
</tr>
</thead>
</table>

Figure 4-1. Service record and context attribute

Services are first matched against a user query, based on the static attributes, which produces a candidate service set. Then through the context attribute evaluation, the candidates are refined to a smaller set of qualified services to be returned to clients. In other words, services are ranked according to the evaluation results to ensure that the clients are given the best service instance(s). The context attribute draws from client-side server selection mechanisms, for example, Smart client [Yos97], but it is generalized for service discovery in our architecture. The context attribute embodies service-specific selection logic hidden from clients. Also, it may prescribe conditions that should be met to further propagate throughout the service discovery network. Some examples of the context attribute are as follows:
• **Distance to server:** The context attribute could describe different metrics depending on specific service implementations, for example, geographical distance to the servers, distance in logical, organizational view, or network distance. The network distance may be measured in terms of hop-count, round-trip time, bandwidth, and so forth.

• **Server load:** Server load could be advertised as a static load attribute. But it requires too frequent updates to keep up with the load changes on the server machine, which may be problematic in a wide-area discovery network.

• **Service channel:** The context attribute may examine service channel conditions to recommend better service instances with respect to QoS.

• **Service advertisement constraint:** Some services may set a certain range within our domain hierarchy for their advertisements to be propagated. For example, a service wants to serve *foo.com* domain and to remain unknown beyond that to avoid being bothered by unintended clients.

It should be noted that clients are unaware of the existence of context attributes so they are not at all involved in the context evaluation. Service authors provide appropriate context attributes for their services, and our context-aware service registry’s BA simply evaluates them without any knowledge of their internals. (Context attribute is required to implement a predefined Java interface, as explained later.) The rationale behind this is that the service authors know what selection criteria matter the most to their services.

Also, the BA can be relieved of the burden to know the selection criteria of all types of services. The usefulness of the context attributes is highlighted especially for domain and global service discovery, since gain by the choice of the right service is magnified due to the diversity of service implementations, devices, and networks and network uncertainty. The context-attribute evaluation is usually a cooperative effort of the servers and clients, which may cause communication in between them during the evaluation process.

Benefits from exploiting context information for service discovery through the context attribute, which is a key feature of our context-aware, enhanced service discovery architecture, are summarized as follows:
The context attribute provides a general, flexible means to enable sophisticated service selection. Service authors are allowed full flexibility to express selection criteria specific to their own services.

The context attribute is an effective indicator of service quality that is able to capture various kinds of service qualities, including communication-related and application-related aspects.

The evaluation cost of context attributes is amortized over multiple clients. To justify the non-negligible overhead of late evaluation, the evaluation result is cached so that it can be reused for future requests for the same service made by other clients.

The context attribute is transparent to clients. Its existence and evaluation happening behind the scenes are completely hidden from the clients.

Context-aware Service Discovery Scenario

In this section, we present one service discovery scenario on a new student’s first day on the University of Florida (UF) campus to illustrate the merit of context attributes over static attributes. Table 4-1 summarizes five example services used in the scenario.

Table 4-1. Static attributes versus context attributes

<table>
<thead>
<tr>
<th>Service</th>
<th>Static attribute</th>
<th>Context attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guide</td>
<td>Manual selection</td>
<td>Closeness detection based on domain name</td>
</tr>
<tr>
<td>Movie preview</td>
<td>Manual selection</td>
<td>Network bandwidth and delay detection</td>
</tr>
<tr>
<td>Movie theater</td>
<td>Random selection</td>
<td>Physical distance understanding</td>
</tr>
<tr>
<td>Printer</td>
<td>Constant load information pushing</td>
<td>Load information pulling when necessary</td>
</tr>
<tr>
<td>Bus schedule</td>
<td>Flag attribute</td>
<td>Advertisement controlled by time of day</td>
</tr>
</tbody>
</table>

Guide service

A new student arrives on the UF campus to start his studies. At the entrance of a residence hall, he picks up his PDA to discover area guide services. Assume that there are
three guide services registered with a service registry: dorm guide, campus guide, and Gainesville city guide.

**Context attribute.** A domain context attribute attached to the guide services determines the closeness between the service registry and service instances based on the DNS domain name. Since the service registry is in the dorm.ufl.edu domain, the dorm guide is given the highest score as a result of the evaluation. The student gets back a service instance list sorted in the order of dorm guide, campus guide, and city guide service.

**Static attribute.** The user has to browse the list to figure out which service instance he needs, relying on descriptive static attributes. Unfortunately, it may take too long to find that it does not provide detailed information about the residence hall, if he first chooses the campus guide service.

**Movie preview service.**

He finished his check-in process. Now he wants a movie for the rest of the day. He needs to check movie previews from movie-ad sites, using a MPEG player program. They are likely out-of-state sites (that is, remote services) maintained by nationwide movie distributors.

**Context attribute.** The best site in terms of network bandwidth and delay can be determined by ping context attributes, which check the network condition to advertised sites.

**Static attribute.** He will pick up one at random. If he selects a service with intolerable QoS, he will have a long delay and the jitters.
Movie theater service

At last, he made up his mind as to what movie to watch. Now he needs to search for local theaters that advertise themselves to the service registry.

Context attribute. The nearest one is recommended by a location context attribute based on ZIP codes. Assume that the registry is preconfigured with its own ZIP code, or the context attribute can somehow acquire it. Then the service registry, more specifically the location context attribute, will be able to figure out approximate distances to the theaters from its own ZIP code and the theaters’ ZIP codes.

Static attribute. He has to make a random selection of ZIP codes (that is, part of service descriptions displayed on his PDA). But he does not know what his ZIP code is since it is his first day on campus.

Printer service

The student wants to print out the direction to the theater using a nearby printer. Note that some printers are being heavily used by housing staffs to process student check-ins. According to the movie schedule from the theater service, he knows that he does not have much time for the movie. He needs to rush.

Context attribute. A load context attribute is used to figure out which printer is least loaded. The load attribute connects back to its service for current queue length, which returns up-to-date load information in response. Then the service registry is able to present to the user the list of printer services sorted in the ascending order of queue lengths.

Static attribute. A declarative, static load attribute is associated with a printer service. The attribute should be updated immediately as soon as the printer load changes although nobody requests the service. This results in the waste of valuable network
resource and processing power. Also, the user has to make a manual selection based on the advertised load information.

**Bus schedule service**

The student decides to take a bus. The access point device at a bus stop in front of his residence hall hosts a service registry. Each bus running through the bus stop advertises itself to the registry.

**Context attribute.** The registered bus services are either shown or hidden, depending on the evaluation results of their *in-service* context attributes. In other words, buses will not be returned to the user according to the context attribute evaluation results if they already stopped running for the day.

**Static attribute.** Each bus service may have an *in-service* flag attribute to indicate whether or not the bus is still in service for the day, and the attribute is to be changed every night and every morning.

**Context-Aware Service Discovery Implementation**

Jini is chosen as a base system to build and demonstrate our context-aware service discovery system, more specifically Sun’s Jini reference implementation, named *reggie*, version 1.2. As with other service discovery protocols, it does not do much for context-awareness support. As to where to put the intelligence to capture context information, we made a decision that context-awareness should be supported by the infrastructure (that is, the BAs associated with service registries), since it is often the case in which resource-constrained mobile devices cannot afford it. A basic assumption of our approach, therefore, is that a client interacts with the closest service registry so that the context captured by the registry can approximate that of the client.
All changes we made are kept within the Jini lookup service, and the standard Jini APIs are not affected. Our modifications are therefore transparent to ordinary clients, and our augmented service registry can support services with either ordinary static services or with static and context attributes. Services with context attributes are given special treatments by our enhanced service registry, but the clients are kept unaware of them.

**Definition of Context Attribute**

Extending the Jini naming space `net.jini.lookup.entry`, we added several Java classes to define the context attribute, including `LDContextEvaluator`, `RDContextEvaluator`, `AbstractLDContextEntry`, `AbstractRDContextEntry`, and `ContextEvaluation`. Parts of these class definitions are shown in Figures 4-2, 4-3, and 4-4.

```java
package net.jini.lookup.entry;

public interface LDContextEvaluator {
    ContextEvaluation evaluate();
}

public interface RDContextEvaluator extends java.rmi.Remote {
    ContextEvaluation evaluate() throws java.rmi.RemoteException;
}
```

Figure 4-2. Java interface definitions of local and remote context attribute

```java
package net.jini.lookup.entry;

import java.io.Serializable;

public class ContextEvaluation implements Serializable {
    long validUntil;         // duration + current time.
    long duration;          // period during which this evaluation remains valid.
    long contextIndex;   // indicates service quality.
    ....
}
```

Figure 4-3. Java class definition of context evaluation result
A context attribute is either a local or remote object. The local context attribute and remote context attribute implement the *LDContextEvaluator* and *RDContextEvaluator* interface, respectively. If the evaluation can be performed by a Jini lookup service alone, it is a local context attribute. For example, if a context attribute measures network hop-count toward its service instance, it is a local object evaluated solely by the lookup service. The evaluation process does not involve the corresponding service instance. As an example of a remote object, we can think of a context attribute that probes the current load on the server machine. This context attribute will be a stub object that makes an RMI call back to its service instance.

All context attributes are required to implement either of the two interfaces, so our lookup service can evaluate them by simply invoking their *evaluate()* methods without understanding their internals. The *evaluate()* method returns a *ContextEvaluation* instance that contains *duration*, *validUntil* and *contextIndex*. The *duration* field indicates a time period during which the evaluation result is likely to remain unchanged. This allows the evaluation result to be cached in our lookup service for future requests from local clients at the site. The *validUntil* field is an absolute time when the cached result expires. The third field, *contextIndex*, represents a comparative quality index of services in question in a numeric form, for example, a normalized index out of 100. It is important to know that this normalization is defined on a service type-by-type basis, as each Jini service interface is autonomously defined by a community of all parties involved in the service type. This ensures that relative superiority can be indicated by a normalized number among service instances of the same type. Service selection criteria specific to a service type will be defined as part of the service standardization, and the index value
will be valid across the instances of the service type. Therefore, the definition of `contextIndex` frees our lookup services (more specifically BA) from having to understand the normalization semantics of every service type for comparison. We believe that this approach is well aligned to the Jini design philosophy to foster the autonomy for individual service implementations.

```java
package net.jini.lookup.entry;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.server.RemoteObject;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.NoSuchObjectException;
import net.jini.core.entry.Entry;
import net.jini.entry.AbstractEntry;

public abstract class AbstractLDContextEntry extends AbstractEntry
        implements LDContextEvaluator {
    public ContextEvaluation val;
    ...
}

public abstract class AbstractRDContextEntry extends AbstractEntry
        implements RDContextEvaluator {
    public Object rObj;
    public ContextEvaluation val;
    ...

    public AbstractRDContextEntry() throws RemoteException {
        super();
        UnicastRemoteObject.exportObject(this);
        setStub();
    }

    private void setStub() {
        try {
            rObj = RemoteObject.toStub((Remote)this);
        } catch (NoSuchObjectException e) {
            System.err.println("… error : " + e);
        }
    }
}
```

Figure 4-4. Base class definitions for local and remote context attribute
Figure 4-4 shows the base classes for the local and remote context attributes. They extend the `net.jini.entry.AbstractEntry` class that is a base `Entry` type for Jini service attributes. By subclassing it, service authors are relieved of the internal details of context attributes. Both contain a `ContextEvaluation` type `val`, which caches their previous context evaluation results. Although not shown in the figure, the constructor of the `AbstractRDContextEntry` class exports itself to the RMI runtime system so that a lookup service can make a connection back to it for context evaluation. Finally, Figure 4-5 shows a sample remote context attribute, the `ServerLoad` attribute.

```java
import java.rmi.RemoteException;
import net.jini.lookup.entry.AbstractRDContextEntry;
import net.jini.lookup.entry.ContextEvaluation;

public class ServerLoad extends AbstractRDContextEntry {
    public ServerLoad() throws RemoteException {

        private long getSystemLoad() {
            // get current load and return it
        }

        public ContextEvaluation evaluate() throws RemoteException {
            long sysLoad = getSystemLoad();
            return new ContextEvaluation(1*60*1000, MAX_LOAD – sysLoad);
        }
    }
}
```

Figure 4-5. A sample ServerLoad context attribute

**Context Attribute Processing**

The context-awareness added to the original Jini lookup services is transparent to clients, since context attributes, provided by the service authors, are evaluated by our lookup service (and service providers in case of remote attributes). Therefore, the clients make a discovery request, using standard Jini interface and static attributes without
worrying about whether they are seeing original lookup services or our enhanced lookup services. In other words, original Jini APIs are kept untouched.

As shown in Figure 4-6, our lookup service considers \textit{CANDSETSIZE} times the number of service instances a client requests. The \textit{slookup()}, the original \textit{reggie}’s lookup method, is first performed based on the static query. Then the method goes through each service instance to see if any context attribute attached to it needs to be re-evaluated. Again, the previous evaluation result is cached in the \textit{val} field of the attribute. A thread is assigned to each context attribute so that the evaluations can be performed in parallel. Since each attribute may take different evaluation times, we need to set a timer. When this timer expires, any ongoing evaluation is aborted. After that, it sorts the candidate set in the descending order of the \textit{contextIndex} value. Service instances with context attributes are favored over those without them. It is possible that some instances do not have any context attribute attached to them, even if they are all the same service type. Before returning top \textit{maxMatches} instances to its client, context attributes are removed from their \textit{Entry} list to make the context awareness processing invisible to the client.

```java
ServiceMatches lookup(ServiceTemplate tmpl, int maxMatches) {
    int candidateSetSize = maxMatches * CANDSETSIZE;
    candidates = slookup(tmpl, candidateSetSize);
    for (each instance in candidates.items) {
        if (it has an instance of AbstractLDContextEntry or AbstractRDContextEntry) {
            if (previous evaluation expired)
                evaluate the context attribute;
        }
    }
    sort the candidate set according to contextIndex
    remove context attributes from each service;
    return top maxMatches instances;
}
```

Figure 4-6. Context attribute processing
Implementation of Domain Service Discovery Subsystem Prototype

We have also developed a prototype of the domain service discovery subsystem by building a hierarchy of Jini lookup services. The context processing logic for service discovery shown in Figure 4-6 has to be revised to enable discovery across registries. Also, inter-registry advertisement and lease management must be supported.

Domain service discovery

If not enough candidate services are discovered in a local lookup service (the size of candidates is less than candidateSetSize in Figure 4-6), the lookup service continues to search its parent registry which, in turn, passes the query to the grandparent. (This upward recursive search goes on until necessary candidates are discovered.) In general, the incrementally expanding search scheme enables the discovery subsystem to consider closer candidate services first that are more likely to share the same context with the original lookup service. Those services imported from ancestor nodes are stored in the local lookup service, and are processed and ranked by the same context evaluation logic for local services. They are cached in the lookup service until their lease granted by source lookup services expires.

Domain service advertisement

Services should be able to propagate to parent lookup services along the hierarchy. Similar to the LDContextEvaluator and RDContextEvaluator, we introduced the LAContextEvaluator and RAContextEvaluator interface to evaluate the service advertisement context. Two corresponding abstract base classes, AbstractLAContextEntry and AbstractRAContextEntry, were added as well. The advertisement context attributes are to be evaluated at the time of registration. After a service registers with itself, our
service registry passes the service on to its parent registry, if it has an attached
advertisement context attribute. Depending on the context attribute evaluation result by
the parent, the service may be allowed to further propagate or get dropped at that point. In
other words, the context attributes prescribe the preconditions that should be met to
further propagate the services. This way service providers can control the reach of their
service advertisements so that their resources can be dedicated to their targeted clients by
avoiding being disturbed by unintended customers. Otherwise, clients will not find out
that discovered services are not intended for them until they are later rejected by the
access control logic of the services.

When a service is propagated upwards, the chain of pointers should also be kept for
the purpose of service lease management and service updates. A node has a responsibility
to timely renew the lease of the services propagated to the parent. The node is the lease
holder of the services from the parent’s perspective. The pointer chain formed for lease
renewal and cancellation along the hierarchy is also used for service attribute
modification.

**Implementation details**

Figure 4-7 depicts the architectural diagram of the *reggie* version 1.2. The figure
also shows how a client and a lookup service interact with each other. The arrows
indicate RMI communication. The client components are dynamically downloaded to the
client when a client program discovers and associates itself with the lookup service. The
client sees the lookup service only via Jini standard interfaces, such as

`net.jini.core.lookup.ServiceRegistrar`, `net.jini.core.lookup.ServiceRegistration`,
`net.jini.core.lease.Lease`, and so on. As shown in the figure, these are implemented by
client-side proxy classes (*RegistrarProxy*, *Registration*, and *ServiceLease* class) that
transform Jini API parameters (*ServiceItem*, *ServiceTemplate*, *ServiceMatches*, and *Entry* class) into internal representations (*Item*, *Template*, *Matches*, and *EntryRep* class) for message transfer efficiency. Also, the proxy classes make corresponding method calls on the *Registrar* interface. The *Registrar* interface defines a private RMI communication protocol between the various client-side proxy classes and the registration server implementation. Most of our modifications for context-awareness are confined to the *RegistrarImpl* class with an exception of the *RegistrarProxy* class where a few changes were made. Since none of the Jini standard APIs that clients interface with was changed, our prototype is backward compatible with standard Jini implementations.

![Figure 4-7. Reggie implementation architecture](image)

The important changes we made to enable context-awareness for service discovery and advertisement are summarized as follows:

Our changes internal to the method implementations of the *ServiceRegistrar* interface include:
For the ServiceRegistration interface, the following method implementations are changed to add, modify, or set the attributes of services that were propagated upwards along the domain hierarchy. When a registered service item’s attributes are updated by the service provider, the same changes are made to the registration with the parent by following the parent pointer.

- void addAttributes(Entry[] attrSets) throws UnknownLeaseException, RemoteException
- void modifyAttributes(Entry[] attrSetTemplates, Entry[] attrSets) throws UnknownLeaseException, RemoteException
- void setAttributes(Entry[] attrSets) throws UnknownLeaseException, RemoteException

To properly maintain the lease chain, we had to change the following method implementations belonging to the Lease interface. When either is invoked, the same method is also called at the parent along the chain.

- void cancel() throws UnknownLeaseException, RemoteException
- void renew(long duration) throws LeaseDeniedException, UnknownLeaseException, RemoteException

Experiments

We conducted a series of experiments of our prototype context-aware lookup service based on Jini reference implementation version 1.2 from Sun. The experiments were designed to evaluate the prototype in terms of discovery performance and benefit, which used a single lookup service and consisted of two parts: service registry-side and
client-side performance experiments. We used a Sun Ultra-60 workstation with 512M RAM, running J2SE v1.3.1_02.

The benefits from the context-awareness are evident. But, at the same time, it introduces processing overhead to the lookup service that must be scalable to support a large number of service registrations. Jini service attributes (that is, Entry objects) are packaged as serialized bit streams (that is, MarshalledObject) so service matching can be made via bitwise comparison. The matching mechanism allows fast processing by lookup services without reconstituting the bit streams into real objects. This is why Jini can support only exact matching for service lookup. But in our case, the MarshalledObject objects for context attributes should be reconstituted back into objects to invoke evaluate() methods on them. Additional processing burden and network traffic will be introduced by deserializing the marshaled objects and the following evaluate() method calls. The first set of experiments is designed to address this scalability concern.

Our lookup service running on a Sun Ultra-60 machine is populated with 10,000 service instances of the same type. The services used for experiments have two attributes: one is an instance of net.jini.lookup.entry.ServiceInfo class and the other is an instance of one of net.jini.lookup.entry.Location, Domain, Load, or Ping class. Note that the ServiceInfo and Location instance are static attributes and the other three are context attributes. The Domain context attribute looks into the DNS domain name of the registry, which may be checked for service providers who want to publish their services within a certain domain boundary. The Load attribute connects to its service provider to ask the current load on the server machine. Finally, round-trip time is measured by the Ping attribute that may be found useful by network latency sensitive services. The Domain and
Ping attributes are local context attributes, while the Load attribute is a remote context attribute.

For each of these previous services, a client on the same machine makes four discovery requests to the lookup service with 10,000 instances of each type, asking for 1, 10, 100, and 1,000 instances each time, although the queries are satisfied with all 10,000 instances. For each query, we measure CPU time taken by our augmented reggie using a Java profiling tool [Ejt02]. The evaluation result caching feature is disabled for these experiments.

![Graph showing CPU time for different number of service instances for different attributes.]

Figure 4-8. Jini lookup service performance

Figure 4-8 plots the CPU time in logarithmic scale on both the x- and y-axis for each service type. The lookup service performs well for the Location static attribute, while processing time for the three context attributes rises drastically beyond the 10 instances, as the number of returned service instances increases. But it should be noted
that the simultaneous context evaluation of 100 or 1,000 instances is an extreme setting for scalability evaluation. Such a bursty evaluation is unlikely to happen in a real situation unless a client browses the whole service in the lookup service (in this case, context attribute processing would be bypassed) for the following reasons: Candidate services are first selected by static attribute matching, and then context attributes are evaluated for those services in the candidate set. In addition, the cache of previous evaluations will reduce the number of service instances that need to be evaluated. Finally, as expected, the local context attributes (Domain and Ping) take less time than the remote attribute (Load).

Our second experiment set is to analyze contributions to the CPU time by the Ping context attribute. Again, caching is disabled for these experiments. Sun’s reggie uses com.sun.jini.thread.TaskManager to manage a pool of threads for Jini discovery and event handling. We use another TaskManager instance for threads to evaluate context attributes. Figure 4-9 breaks down the CPU time for the Ping attribute evaluations.

Figure 4-9. Composition of lookup service CPU time
The plot shows relative time spent on three methods. The TaskManager’s TaskThread calls our ContextTask.run() that, in turn, executes the Ping.evaluate() method. The TaskThread also manages other tasks, such as Jini discovery and event task. Since ContextTask time is part of TaskThread time and Ping.evaluate() time is part of the ContextTask time, each line in the graph subsumes its underlying line. As the number of returned services increases, a larger portion of the CPU time is taken up by TaskThread (62% and 85% of the CPU time at “ping-100” and “ping-1000” point, respectively).

ContextTask.run() uses about 58% of the CPU time for 100 instances and 35% for 1,000 instances. At the “ping-100” point, 23% of the CPU time is spent on Ping.evaluate() method, while 35% of the time is wasted mainly for thread synchronization to access the shared data structure, that is, reggie’s service map. (It is the time that threads waste on the com.sun.jini.thread.ReadersWriter.writeLock() method invocation.) Less than 4% of the CPU time is used for multithread execution and switching (that is, the time spent on the java.Object.wait() method). In the case of the “ping-1000” point, the numbers change to 13% and 22%. And 50% of the CPU time is used for concurrent thread execution.

To measure time observed by clients, we ran another set of experiments. For three services, including compute, dictionary, and ftp service, we measured lookup time and service time. The compute service offers a computation service that employs the Load context attribute in the previous experiments. The dictionary service returns the meaning of a word asked by clients, which is similar to dictionary Web services. Since latency is the most important criterion for this type of interactive services, it uses the Ping attribute used in the previous experiments. Finally, the ftp service uses a new Bandwidth attribute that examines available bandwidth between the lookup service and the ftp service.
provider by rehearsing a 100K packet. For each service type, we populated 8 service instances with different service quality, ranging from 1x, 2x, 4x, 8x, 16x, 32x, 64x, to 128x. Figure 4-10 shows time for the lookup() method by a client under these experiment settings.

Figure 4-10. Client lookup time

The x-axis represents different context evaluation miss ratios. For example, “miss-0” means that cached context evaluations are valid for all candidate services produced by static attribute matching. In addition, the case of “static attribute only” is shown for comparison. As the miss ratio increases, the client experiences moderate increases of lookup time for compute and dictionary services. But for ftp service, the lookup time changes drastically. This is because average context evaluation time and its variation differ across service types and instances of the same type. Exercising a 100K packet over the 1x bandwidth link takes a much longer time. This is why we need a timer for context
evaluation, as previously discussed. Also, we can see that links with better bandwidth for
the “miss-75” case than for the “miss-50” case have happened to be chosen.

Figure 4-11 shows time for the first round of service uses. For example, service
time for ftp service is the time to fetch 1M data. The x-axis represents how many cached
evaluations are valid at the time of client requests. For example, “cand-50” means that
context evaluation results are available for 4 out of 8 services. Service selection is made
among those cached services without evaluating expired ones. For static attribute, the
service time comes close to the average of all 1x through 128x 8 instances since the
instances are selected at random fashion. The graph shows that service time can be
dramatically reduced by service recommendation via context attributes at “cand-25”. The
service time continues to improve as the candidate set size grows toward the perfect case
of “cand-100”.

![Figure 4-11. Service time experienced by client](image)
Context-awareness for Service Discovery

Various server selection mechanisms show a full spectrum ranging from server-side to network and to client-side approaches. However, most of the mechanisms lack generality to be used by a variety of applications since they are fine-tuned to a specific type of service or a specific application domain. To overcome this limitation, the server selection schemes are being generalized by recent RSerPool IETF drafts [Tue01a, Tue01b]. The RSerPool proposal is more concerned with communication-oriented aspects than service-oriented aspects, which positions itself as an additional entity separate from a general service discovery framework. It contrasts our approach, which is a more general service discovery framework that is capable of supporting various server selection mechanisms through context attributes. By integrating the two into a single framework, our architecture provides a single-step service discovery and selection.

Infrastructure support for a value-added service discovery is also found at the Selection and Sort extensions to SLP [Zha02]. However, their approach is to rank service instances according to a static-attribute-based sort key list, which is specified by clients. Clients must understand what attributes for the service type matter the most to be able to specify the selection criteria. Besides, the selection process is performed solely by service registries without any help of services, which may constrain context information that can be captured. The idea of the context attribute draws from client-side server selection mechanisms, for example, Smart client [Yos97]. But it is generalized for general service discovery in our work, while the Smart client targets the Internet services hosted on a cluster of workstations. Another noteworthy work is media service discovery [Xu01] where the service discovery framework is used to cache end-to-end quality of services.
Unlike others, our approach is to be able to capture dynamic aspects of context to enable much more sophisticated selection mechanisms. Consequently, our approach achieves both the specialty of server selection mechanisms and the generality of service discovery protocols.

We implemented the context-awareness support on Jini lookup services with resource-constrained mobile devices in mind. However, it can be brought down to client devices if they have adequate resources to handle the context attributes. In this case, more accurate context information would be able to be captured, and there would be no concern about lookup service scalability. But a disadvantage is that context evaluation results cannot be reused among clients.
CHAPTER 5
GLOBAL SERVICE DISCOVERY PROTOCOL

Our new global service discovery protocol, called Class-based Service Propagation (CSP) protocol, is based on the Centroid that is an aggregation method used for Whois++ index service [Fal96, Wei96].

**Centroid**

The directory mesh for Whois++ index service is composed of Whois++ servers as base nodes and index servers that summarize their child Whois++ servers or index servers. A layer of the mesh summarizes the next lower layer. As such an aggregation method, Whois++ index service introduced the Centroid, which is a collection of every word that appears at least once as attribute values. The Centroid is propagated up along the tree to provide query routing information. The Centroid at very upper layers eventually converges to a large set of words, which means global scalability.

![Centroid Example](image)

**Figure 5-1. A Centroid example**
Figure 5-1 shows an example of the Centroid which summarizes *color* and *fruit* attributes. The attribute values are aggregated per attribute. That is, the Centroid is generated by taking a union of all the words that appeared as the attribute values for a trade-off between storage need and query performance. In this example, the Centroid saves one instance of *Green* and *Banana*, but it introduces false information, that is, *Yellow Papaya*. If the Centroid is looked up for *Yellow Papaya*, it will return a hit although such a record actually does not exist. This is called a *false hit*.

Despite the one record saving, it does not seem cost-effective for every single word to be propagated all the way to the top layer of the mesh in the same way, regardless of how many times it appears in the sub-tree. Seldom-used words contribute little to query routing, while taking up the same storage space as others. They need to be discriminated in favor of more frequently used ones at the upper layers of the index tree for overall system performance improvement.

We envision that there would be a morass of services everywhere in the future ubiquitous computing world. In such a world, the efforts to update the globe on every dynamically changing service will not be paid off. In other words, it is wasteful to advertise full service information to the opposite end of the world where it will likely not be used. Also, queries tend to be specified using popular, common attributes and their well-established values rather than specific query terms. (Clients usually do not know of all the details of services in advance according to dynamic service discovery scenario.) The mobile clients will likely not seek specific service instances but anonymous services providing desired (perhaps standardized) functionalities, for example, queries to look for a color ps printer rather than a printer with a bizarre name. Therefore, these general
attributes need to be favored over specific ones to improve overall service discovery performance without generating much traffic. The vision of service ubiquity leads us to the idea of our Class-based Service Propagation (CSP) protocol as a new aggregation scheme. Its primary design goal is an acceptable query performance without using excessive resources for service advertisements and updates (a trade-off between query routing costs and service advertisement traffic and the storage of descriptive service information). Our new aggregation scheme forces less-contributing words to fade faster than others, thereby enabling quick convergence to a manageable size set.

**Class-based Service Propagation Protocol**

It is well known that word usage in English follows Zipf distribution [Nie03]. That is, a few words are extremely often used, a medium number of words are frequently used, and a huge number of words are rarely or almost never used. Our CSP approach is trying to exploit this word frequency pattern. In general, to be globally scalable, information accuracy likely degrades through a certain means of aggregation or condensation, as it propagates further. Examples of the aggregation include a hierarchical routing protocol (for example, Internet interdomain and intradomain routing protocol) and the Bloom filter used by Berkeley Secures SDS. Noticing the Zipf distribution of service queries and the inefficiency in the Centroid indexing scheme, we devised an efficient aggregation method to employ the concept of classification, that is, different handling on the word frequency basis. Service information is classified as the first and second class. The first class information does not degrade at all or it undergoes TTL-based degradation and lazy update propagation, while the second class experiences aggressive aggregation. Dominant, first-class attributes should be further propagated to attract more queries traveling on the network. In contrast, the protocol forces a faster convergence for the
second-class information through attribute value string splits. To describe our protocol in
detail, we need the following parameter definitions:

- **Class profiling threshold**, denoted as \( pt \), is the class divider to classify service
  information according to the number of its occurrences. Words that constitute
  attribute values are classified as the first class, if they occurred more than \( pt \) times
  at a node. Otherwise, they belong to the second-class information. It is important to
  know that any node in our global service hierarchy is the root of our domain
  hierarchy that advertises collectively its domain services.

- **Minimal word length**, denoted as \( mw \), is the longest string length that is not subject
  to the attenuation process described below. Namely, it indicates the largest string
  size that is not further broken down.

- **Attenuation unit**, denoted as \( au \), is the unit by which a string is cut off at the end as
  it passes through a node. It can be in the range of \( 1 \leq au \leq mw \).

Service attribute values, whose frequency exceeds the \( pt \) threshold value, are
treated as the first class. Otherwise, they are classified as the second class.

The second-class information undergoes forced aggregation through the gradual
string diminution. A string, \( s \), is split into two substrings of length \(|s| - au\) and of length
\( au \) at each hop, as it propagates. This attenuation process makes the string, except for the
leading \( mw \) part, gradually disappear. This way a query will be able to see more detailed
information as it gets closer to potential matches. The attenuation unit parameter governs
the aggregation rate of the second class.

The first-class information propagates up toward the root without any degradation
or by a TTL-based lazy degradation scheme (that is, minor changes to their frequency are
postponed until the accumulated changes break up an update barrier for further
propagation). The first-class information provides an accurate, fast path for query routing
to promote the global scalability. Two possible strategies for the first-class propagation
are as follows:
• Zero degradation: Once raised to the first class, the service information is never degraded. It makes first-class information updates in the middle of a propagation path unnecessary. The path can grow downwards or shrink upwards only at the bottom end.

• TTL-based degradation: The TTL value for the first-class information is defined to be \( \text{ceil}(\log(\text{frequency})) \), which detains update propagations until it becomes a new TTL value. This way frequent update traffic regarding the first-class information can be reduced. The TTL is decreased by one at each hop, and the first class information is downgraded to the second class when the TTL value becomes zero.

![Diagram of first-class propagation](image)

Figure 5-2. Zero degradation of the first class

Figures 5-2 and 5-3 illustrate the zero degradation and TTL-based degradation propagation, where the class profiling threshold \( pt \) is 2. The notation \( S_1\{a_1=v_{11}, a_2=v_{21}\} \) indicates an instance of service type \( S_1 \) whose attribute values for \( a_1 \) and \( a_2 \) are \( v_{11} \) and \( v_{21} \), respectively. As shown in the figures, the first-class and second-class information takes separate propagation paths. In Figure 5-2, changes to the first-class information are updated only for the ones occurring at the bottom of the path and crossing the \( pt \) threshold value. Otherwise, they need not tally up. For instance, service information from
the right-most leaf node does not affect the first-class path. Figure 5-3 shows that the first-class information is demoted to the second-class information, when it uses up its momentum for upward propagation, that is, the TTL value defined to be $\text{ceil}(\log(\text{frequency}))$.

![Diagram of TTL-based degradation of the first class](image)

Figure 5-3. TTL-based degradation of the first class

The second-class attributes below the profiling threshold $pt$ value take a different propagation path. Attribute value strings are split into substrings at each hop along the tree, if they are longer than the minimal word length, $mw$. This attenuation process forces service information to quickly converge to a much smaller size set than the original Centroid, while keeping minimal query routing information (that is, the fragments of $au$ size substrings) for rarely sought services, hence global scalability. It follows that a smaller set size means less service update traffic for the second-class attributes. While the split process allows significant gains in terms of storage and update traffic, there is a
higher chance of false hits introduced by the attenuation. Although query performance
degradation due to the false hits will be alleviated by the ubiquity and abundance of
services in the future and our first-class path, the attenuation rate would be the most
important determinant of specific query performance. Therefore, attenuation parameters,
such as attenuation unit \((au)\) and minimal word length \((mw)\), should be determined
considering tolerable performance for specific queries.

Figure 5-4 illustrates how strings are aggregated by the second-class attenuation
process, where \(mw\) is set to 2 and \(au\) also to 2. Strings are split off by \(au\) characters at the
end, as long as they are longer than the minimal word length \(mw\). The aggregate size is
reduced from 26 at the bottom to 24 at the second layer and to 16 at the next layer. We
note that it is an extreme example to explain the attenuation process. These parameter
values aggregate service information too aggressively, and increased false hits will cause
severe query performance degradation, as shown in the next simulation section.

Figure 5-4. An example of service information attenuation
CSP Query Routing

Given a client query, our global service hierarchy routes it to service registries that contain matching service information. Service information cached by each node provides an answer to which of its sub-trees leads to service instances matching the query. For this, a registry has to keep a separate aggregate for each link to its children. However, the service aggregates are not always able to provide the correct answers because of false hits, which require backtracking to the last branching node.

Figure 5-5 illustrates how cached service information is used for query routing.

Figure 5-5 illustrates how cached service information is used for query routing. Service aggregates are shown to the left of corresponding nodes, and a query to the right along a query path. Suppose that we have a query looking for a service with a “sore throat” attribute string. The top node of the sub-tree in the figure concludes that matching services may exist in its left sub-tree, since the query can be covered by “so,” “re,” “th,” “ro,” and “at” from its left-child aggregate. The query is forwarded into that direction and
a match is eventually found at the left-bottom node. An example of a false hit is a query of “song” for which the root node will point to the left child. But its left child node will find that it was a false hit.

**Simulation of CSP Protocol**

We conducted a simulation study to compare the performance of our CSP protocol and the base Centroid since both are using string-based attribute-value pairs for service descriptions. Performance comparison, which was between the Centroid and Berkeley SSDS to employ the Bloom filter as their aggregation method, was reported in [Hod02a].

**Workload Analysis**

The objective of our simulation is to validate our design decision of CSP protocol. Our primary concern is a balance between query performance and service advertisement traffic. Through the simulation study, we tried to evaluate our CSP protocol’s gains and losses, especially to verify if it achieves an acceptable query performance without using excessive resources for service advertisements.

As our workload, we use the freedb CD database (http://www.freedb.org) as in [Hod02a]. The record for a CD consists of several text-based attributes and associated values. Among them, *YEAR* and *GENRE* attributes are thought of as general attributes, popular for discovery queries (perhaps standardized attributes and their well-established values), and the rest of the attributes as specific ones. Examples of the specific attributes include *DISCID*, *DISC TITLE*, and *SONG TITLES* (*TITLE0*, *TITLE1*, *TITLE2*, *TITLE3*, *TITLE4*, and *TITLE5*), which have unique string values. We compare our CSP performance with the Centroid’s in terms of query cost and storage used for service advertisements. We note that the storage need is proportional to the volume of information exchanged among nodes for service advertisements and updates.
It is a well known fact that word usage in any natural language follows Zipf distribution. Other examples of the distribution are Web-page access and book check-out from a library [Nie03]. The distribution is a straight curve when plotted in logarithmic scale on both the x- and y-axis, which means that (1) a small number of popular words are excessively used, (2) a moderate number of words are frequently used, and (3) the rest of the words are rarely or almost never used. We analyzed our workload consisting of more than 4,000 CD records to find that it shows a perfect Zipf distribution, as depicted in Figure 5-6.

The graph shows curves for 0 to 4 time splits, where both CSP attenuation parameters of \( mw \) and \( au \) are set to 3. The x-axis represents word popularity, meaning word ranking according to occurrence frequency. For instance, the first x value is the most frequently used word, which appears 5,935 times in the case of “no split.” The 7,536\(^{th} \) frequently used word, that is, the last x value, appears only two times. The base straight line corresponds to the “no split” case. As words are split repeatedly, the curve shifts upwards, which means a chance of a huge gain that our CSP protocol can exploit. The split will increase a false hit rate, but it does not necessarily mean the same rate of query performance degradation. A long string is likely to be made up of distinguishing and common parts. The common substrings contribute little to query routing if they show up from a majority of child nodes. We will have to try them one by one. As an example, consider two strings, “pavement” and “movement,” coming from two different child nodes, and assume that “pave” and “move” are unique. If the two are split in the middle, “pave” and “move” will be the determining elements of the query of “pavement” or
“movement,” while the common substring, “ment,” does not provide any guidance, as long as only the two strings are concerned.

Figure 5-6. CD database word distribution

Words used to describe the CD records are of length 1 to 71. Single character words, including alphanumeric and special characters, appear 14,039 times out of total 127,479. The most common length, 4, happens 25,872 times, which constitutes about 20% of the total. The longest of length 71 appears only once. Table 5-1 shows the word length distribution of the original workload (no split), and 95% of them are less than length 10.
Table 5-1. CD database word length distribution

<table>
<thead>
<tr>
<th>Word length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10 - 71</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>14039</td>
<td>12481</td>
<td>16631</td>
<td>25872</td>
<td>16853</td>
<td>11964</td>
<td>9908</td>
<td>9887</td>
<td>4352</td>
<td>5492</td>
</tr>
<tr>
<td>Cumulative percentage</td>
<td>11.01</td>
<td>20.80</td>
<td>33.84</td>
<td>54.14</td>
<td>67.36</td>
<td>76.74</td>
<td>84.52</td>
<td>92.27</td>
<td>95.69</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Also, the assumption our CSP protocol made about the general-to-specific query ratio is shown to be true by the Zipf distribution curves. As presented in the following section, specific query performance is sacrificed in favor of service advertisement traffic. However, CSP protocol achieves improved overall system performance in return.

Figure 5-7 shows the effect of the splits in terms of an aggregated service database size. The graph plots how much of the service aggregate of random 2,040 services can be reduced by different attenuation rates. The legend aggregation-n indicates the case of attenuation parameters of $mw = n$ and $au = n$.

Figure 5-7. Service database size reduction by splits
Simulation Results

The global registry hierarchy for all our simulation cases is a complete binary tree comprised of 255 nodes. In addition, the class profiling threshold $pt$ is fixed to 16 because we need much larger scale simulation to see its effect. Figure 5-8 is a simulation result, when 2,040 services are randomly populated on the complete binary tree.

![Simulation Results Diagram]

Figure 5-8. Database size at each layer

The plot compares average service aggregate size at each layer for different aggregation rates. (The $aggregation-n$ indicates the case of the attenuation parameters of $mw = n$ and $au = n$.) In the case of the Centroid, the aggregate database grows exponentially, which means that top-tier nodes of the tree will be overwhelmed by massive traffic. It may be thought of as CSP protocol with $mw = \infty$. A huge amount of storage and traffic is spent to index every word appearing in the tree. However, the cost will not be paid off considering a morass of services and dominant general queries in the
future ubiquitous computing world, as discussed earlier. In summary, our CSP protocol forces further service aggregation, enabling better storage efficiency.

Figure 5-9 presents query performance evaluation results for different aggregation parameter values of $mw$ and $au$: (a) indicates query routing performance in terms of the average number of hops that a query travels until a match; (b) shows the total space use throughout the tree; and (c) tries to compare the efficiency across different settings. Total 2,040 CD information is randomly distributed over the 255-node tree, and 255 queries are also randomly fed into the network. The general-to-specific query ratio is 1:1, meaning that half the queries are to look for a CD using the YEAR or GENRE attribute as a search key. The other half, that is, specific queries, are specified in the DISCID, DISC TITLE, or SONG TITLE 0 to TITLE 5 attributes.

Graph (a) confirms that the minimal word size ($mw$) 2 is too short so it overly aggregates service information, resulting in severe query performance degradation for the given workload. In the upper part of the tree, query match results are almost always true, thereby forwarding the queries to every child node (becoming equivalent to an exhaustive search). But the cases of $mw$ 3 and 4 (for both $au=1$ and $au=mw$) achieve comparable query performance with the Centroid. Graph (b) shows storage savings by CSP protocols. Graph (c) presents protocol efficiency defined to be the product of average hops and storage usage, a quick and rough comparison metric. It is important to know that the protocol efficiency should be appraised from a global scalability perspective. Space usage at the upper parts of the tree should be considered rather than the total storage (see Figure 5-8), and traffic near the root is also an important criterion (refer to Figure 5-11).
Figure 5-9. Query cost versus attenuation rate

Figure 5-10 compares the query performance of CSP protocol with that of the Centroid for different general-to-specific query ratios. Other simulation conditions are the same as in the previous simulations, that is, total 2,040 services populated over the 255-node tree and 255 queries. Again, the aggregation-n indicates the case of the attenuation parameters of $mw = n$ and $au = n$. Except for the aggregation-2 case, the CSP protocol
shows comparable performance with the Centroid’s. The CSP query takes a bit longer response time, which is attributed to the increased false hit rate caused by CSP’s attenuation. As long as query performance is concerned, the Centroid obviously outperforms our CSP. By using more resources to keep the service index, the Centroid can provide nearly optimal query performance from our CSP’s point of view. However, our protocol design goal is a balance between query performance and storage and advertisement traffic.

![Figure 5-10. General-to-specific query ratio versus cost](image)

We also ran additional simulations to measure CSP query performance for different service populations of 4,080, 2,040, 1,020, 510, and 255 services. The query performance was largely consistent regardless of the service populations. (The CSP query latency slightly increased, as the service population grows.)
To compare service update performance, 127 new service insertion requests are issued over the 255-node tree with 2,040 services preloaded. As shown in Figure 5-11, CSP protocol generates much less traffic than the Centroid, especially for the top tier of the tree, which is critical to be globally scalable. Again, the aggregation-$n$ indicates the case of the attenuation parameters of $mw = n$ and $au = n$. The x-axis value $d$ indicates the amount of update traffic between depth $d$ and $d+1$. Finally, we note that service deletion showed similar traffic usage.

![Figure 5-11. Update traffic at each layer](image-url)
CHAPTER 6
SERVICE DELIVERY

User mobility inherent in the future ubiquitous computing environments dictates service mobility support. For instance, a user should be able to access the same service on his desktop computer at work, on his auto PC on the road, or on his smartphone on the street. That is, services should be able to be presented to a variety of target devices in an appropriate way to the device capability, which entails service instance selection, service transformation, or some sort of terminal adaptation. In addition, provided a “note-taker” service delivered to his PDA is able to support dual modalities of pen and voice interfaces, he will likely prefer the voice command interface on the street. But he will probably use the pen interface to edit his voice notes back in his office. The diverse user device capability and interface modality are a main force to drive device-independent service delivery research, and they are primary context information to be taken into account for service delivery.

Thin-client Approach to Service Delivery

To deliver services to client devices independently of the target device capability, we have developed a proxy system, called the µJini proxy system [Hel02a]. This prototype is to be incorporated into the BA of our three-tiered service discovery framework presented in Chapter 3. Built on top of Sun’s Jini and AT&T’s VNC (Virtual Network Computing) [ATT02], the prototype deals with service discovery and delivery. More specifically, it functions as a Jini proxy with regard to service discovery, and it serves client devices as VTC (Virtual Thin Client) servers for service delivery as well.
From a service user’s perspective, our µJini architecture provides the same functionality as the combination of Jini surrogate architecture [Sun01a] and Jini ServiceUI project [Ven01] does. Jini surrogate architecture also introduces a proxy to represent to a Jini community resource-constrained devices that cannot afford Jini protocol. Unlike the Jini ServiceUI project that requires multiple GUI implementations (one for every potential target device class), our “emulation through VNC” approach is able to deliver one service running on the network proxy to any type of client devices. Therefore, a single service implementation is enough for various target device classes. The Jini surrogate architecture plus the Jini ServiceUI project require that a service implementation follow the discipline of the surrogate architecture for small, non-Jini devices with extra burden to implement a corresponding full-blown version once again for Jini-capable devices. In contrast, our VNC emulation is an “all-in-one” approach; a single service implementation would be enough regardless of whether it is a full-blown Jini implementation or surrogate implementation. Also, we do not need to care about specific graphic toolkits available on target devices, as long as they meet a minimal requirement to run our VNC client program. The µJini proxy system chooses the most appropriate implementation, considering service implementations and client devices altogether, when multiple choices are given. If any matching service implementation is not available, it resorts to the VNC emulation.

To bring various services to resource-constrained mobile devices, our µJini system employs three different approaches. The simplest and most straightforward mode is to have a service code executed entirely on the client side (Client executable mode). In case of fat services or if resources needed by the services are not available, we have to resort
to two adaptation approaches. These approaches rely on the proxy to execute any kind of fat services, either MIDlets [Sun02b] or J2SE services, through the network-side thin-client system, which sends only screen updates to the client. The first adaptation enables MIDlets to be executed by our MIDlet emulator on the proxy side (VTC MIDlet emulation mode), while the second is for J2SE services (VTC J2SE emulation mode). The client should not be involved in the decision of the delivery methods. Our μJini system is capable of selecting the best under the given service invocation conditions, and it minimizes the service’s dependency on device capability to achieve device-independent service representation.

A Java interface, VTCable, must be implemented by any service object that may be invoked through the VTC emulation modes. The interface is an empty marker interface to indicate that an object implementing it is allowed to run on the μJini proxy.

μJini Proxy System Architecture

As shown in Figure 6-1, the μJini proxy system architecture is made up of four main components: μJini Protocol, Resident Client, VTC (Virtual Thin Client) Client, and μJini Proxy.

The Resident Client resides on a client device and takes care of interfacing between the client and the proxy server as well as handling the VTC client/server communication through a VNC channel. The μJini proxy contains many different pieces and performs the Jini service discovery on the network side for the client as well as allocating a thin-client server for the client to access to use the discovered service. The VTC is split into two pieces, VTC Client on the client side and VTC Server on the μJini proxy side. Most of the processing is pushed to the proxy server side so the low-resource client device can
operate more efficiently. The µJini Protocol allows the client and proxy to talk using a common library, and a separate VNC channel is used for the VNC (Virtual Network Computing) server connection. The VNC is an open-source, thin-client system developed by AT&T Laboratories Cambridge [ATT02].

Figure 6-1. Overall µJini system architecture

**µJini Protocol**

Because small mobile devices are commonly resource-constrained, the µJini proxy has been introduced to offload heavy Jini requirements from client devices. Jini requires J2SE serialization and RMI, which are not supported by J2ME (CLDC/MIDP) for small devices [Sun02a, Sun02b]. We have therefore defined the µJini Protocol API which allows the client device and proxy server to communicate using a common set of library APIs. The protocol is simple enough to fit into small devices, but supports all necessary messages for µJini proxy discovery, Jini service discovery, service invocation, lease
management, and client device description (that is, client device capability context information) passing. Its full description can be found in Nordstedt’s MicroJini [Nor01]. Any µJini Protocol session begins with passing the client device profile to the µJini proxy, similar to CC/PP device profile passing [Kly02].

**Resident Client**

The Resident Client is a MIDlet program, in the case of small J2ME devices, running on the client devices. It provides a graphical user interface to the user for speaking the µJini Protocol to find and request services from the µJini proxy server. It also takes care of instantiating a VTC viewer and cleaning up resources after a service session is complete. In the case of downloaded services, that is, the client executable mode, the Resident Client will interact with the proxy, download the MIDlets from a specified http server, and load the code onto the client device using native operating system functions. Otherwise, it will start a VTC viewer for VTC emulation service delivery.

**VTC Client**

The VTC on the client side implements the main encoding schemes the VNC rfb protocol defines [ATT02]. These include Raw, Hextile, RRE, and CoRRE encoding schemes. In addition, we have added our own protocols to send updates as a PNG image, using the freely available PNG image format [PNG01], since J2ME provides built-in support for PNG. We support PNG 8-bit color, PNG 8-bit grayscale, and also PNG 2-bit grayscale images.
μJini Proxy

Figure 6-2 shows the components that comprise the μJini proxy. It functions as not only a Jini proxy to translate the μJini protocol to full Jini protocol, but also a thin-client adaptor that executes heavyweight services on behalf of the client device. Each of the components is briefly described below.

Figure 6-2. Components of μJini proxy

Dispatcher

As shown in Figure 6-2, the Dispatcher is the central piece of the μJini proxy, which mediates all the interactions among components. First of all, it communicates with client devices, speaking the μJini protocol. Upon requests from the clients, it performs service discovery on behalf of the Resident Clients and provides MIDlet or J2SE service emulation services through the VNC channel. More specifically, the Dispatcher contacts the Jini Manager to perform service discovery from Jini lookup services in the local
network. Also, it makes a MIDlet or Jini service emulation request to the VTC manager, if necessary.

**Proxy shell**

The Proxy shell provides a simple administrative shell interface so that administrators can check the system status, such as active Jini lookup services, VTC server pool usage, and current client list.

**Jini manager**

The Jini manager provides the rest of the proxy with Jini interface service. Basically, it keeps track of active Jini lookup services in the local network. It translates the µJini Protocol messages into Jini representation to query these lookup services.

**Verifier**

The Verifier checks if discovered service objects are MIDlets that are downloadable to a J2ME device. When it determines service delivery mode, it must take into consideration the discovered service object, client device capability, and the delivery mode the service provider has set. First, the Verifier inspects the service object to see if it is J2SE objects or MIDlet(s). In the case of MIDlets, the verification process proceeds to check if the delivery mode set by the service providers allows downloading to the client. Note that the service providers’ wish precedes the Verifier’s decision on delivery mode. Furthermore, it looks into program/data space on the device to ensure that there is enough room for the MIDlets. If it passes all these checks, the Verifier finally declares the delivery mode to be “download” mode. Otherwise, the delivery mode is set to “vtc” mode (MIDlet or J2SE service emulation mode).
Adapter

The Adapter executes the decision on the service delivery mode made by the Verifier. If the discovered service turns out to be J2SE Jini objects, it should be executed on the proxy side and the client gets its display through the VNC channel. If the object is MIDlets but not downloadable, it is executed by the MIDlet Emulator, a component of the µJini VTC server. If downloading is allowed, the MIDlets are downloaded to the client and executed locally. Accordingly, it provides emulation services for standard Jini services and MIDlets that do not fit into client devices.

VTC manager

VTC manager provides client devices with the emulation service of MIDlets and J2SE Jini services. The VTC manager maintains a pool of VTC servers from which one server instance is allocated to a user who needs the emulation service. When the user session is over, the server instance is put back to the pool.

The implementation of J2SE Jini service emulation is relatively easy. A VNC desktop is dedicated to a JVM on which a Jini service proxy is started. Then its display is sent to a client through a VNC channel and the, in response, user input is forwarded to the service proxy via VNC event messages. For MIDlet emulation, we have developed the MIDlet emulator which is built on CLDC/MIDP reference implementations from Sun [Sun02a][Sun02b]. As described below, input events and output displays are intercepted and diverted to the other side.

MIDlet emulator

The performance of the MIDlet emulator is not a primary concern for our current proof-of-concept version. For rapid prototype development, we built our µJini MIDlet
emulator on Sun CLDC/MIDP reference implementation and reuse them as much as possible.

One of the primary design principles of our MIDlet emulator is to provide a consistent view to users regardless of whichever mode a service is presented in. The emulator is designed to hide any difference between local execution mode and emulation modes by the proxy. However, in our current implementation, users are exposed to a long MIDlet installation phase of the local execution case, which is inevitable due to the limitation of current J2ME technology.

Figure 6-3. MIDlet emulator

To create an illusion that the emulated service is running on the client device, the proxy intercepts and diverts input events to the emulator and output display back to the client. The user inputs are relayed to the emulator running on the µJini proxy side, and resultant emulator display is redirected to the client device through the VNC channel. The flow of user inputs and display updates is as follows:
(1) **VNC server pool**: On startup, the µJini proxy initializes a set of VNC servers used for the emulation service, which avoids the lead-time to bring an underlying VNC server up on receipt of client requests.

(2) **The µJini protocol**: When a new client requests a service to be presented via the MIDlet emulation mode, the µJini proxy assigns a free VNC server to the client and starts a MIDlet emulator instance over the VNC desktop.

(3) **KVM event relay**: The Resident Client on the phone intercepts user key strokes to send them to the µJini proxy which, in turn, passes them to a corresponding MIDlet emulator instance. The decision of the event relaying was made for the sake of easy event processing. The limitation of simultaneous socket connections, which is 2 on the client device (Motorola i85 smartphone), is another reason for the decision. The Resident Client needs one connection, and the other one must be given to the VNC rfb display update channel. By being able to share the Resident Client’s connection for event transfer, the client does not need to interfere with the VNC rfb protocol (that is, no need to understand VNC event representation and processing). Input KVM events are sent via µJini protocol messages.

In the case of smartphones, only the LCD screen part of the emulator display is mirrored to the real phone screen through the VNC channel. Our input event handling mechanism threads two separate pieces together, that is, the user interface part on the client device and the MIDlet service running on the network proxy side, so as to provide MIDlet emulation service. More specifically, the Resident Client intercepts keypad input events and delivers them to the emulator in the J2ME KVM event representation. The µJini proxy simply passes incoming events on to an associated emulator through an open
socket connection. These events are injected to the point where the original emulator used to translate native operating system events (X11 mouse events in our prototype) to corresponding KVM events.

(4) **Display update**: A MIDlet running on the emulator reacts to the event. A resultant display update (VNC rfb update message) is sent to the VTC client on the client device.

Consequently, it looks to the client as if the MIDlet is locally being executed on his device.

**Performance Measurements**

We present basic performance measurements for the two delivery options of our µJini proxy system prototype implementation. Detailed experimental measurements and performance data are reported in MicroJini [Nor01].

There is a huge difference in the time between a downloaded service and VTC emulation service usage. A service application (that is, a J2ME MIDlet) run locally must first be downloaded to the client device and then installed before it can be executed. The low bandwidth wireless networks that are common today, along with the long process of installing an application on a resource-constrained device, add up to a long time for the user to wait to run the service. A service application that is run as a thin-client, however, only needs to wait for the VTC server to start the service and for the client to create a connection to the server. The application can be started in seconds rather than minutes for a downloaded service.

In Figure 6-4, the performance data averages are shown for the total time to start the services after discovery. We have used four J2ME MIDlets on the Motorola i85
smartphone which are *iCoffee, Shopping, PhoneClient*, and *MapIt*. The comparison shows the large difference in time for a user to start a service when downloading compared to when using the VTC emulation mode.

![Comparison of Local Execution Startup and VTC Startup](image)

**Figure 6-4. Service startup time comparison**

We have also implemented an 8-bit color PNG protocol, an 8-bit grayscale PNG protocol, and a 2-bit grayscale PNG protocol to complement existing VNC encoding schemes. These new protocols enhance the overall performance of the µJini proxy system when the VTC emulation mode is invoked as the service delivery option. The performance data of the VTC MIDlet emulation mode for standard and enhanced VNC encoding schemes can be found in MicroJini [Nor01].
CHAPTER 7
SUMMARY AND FUTURE RESEARCH

In this chapter, the main contributions of the dissertation are assessed along with some of the future research tasks regarding the context-aware, ubiquitous service discovery and delivery system.

Summary of Contributions

The main contributions of this dissertation are an architecture, a concept, and a protocol whose aim is to take service discovery from the realm of enterprise networks into a ubiquitous and global scale. By mobilizing service discovery through a scalable infrastructure, a wide class of exciting pervasive computing application will be enabled.

Our first contribution is a three-tiered architecture for service discovery that allows services defined within a proximity (a small network) to remain within the proximity boundary, to become accessible from a domain of multiple proximities (for example, an enterprise), or even to become a global “anywhere” service. The architecture uses a novel concept we call context attributes and a new protocol that extends IETF’s Centroid indexing service to guarantee the scalability of global service advertisements while minimizing overall service query performance degradation.

We introduced context attributes, which are an effective means to capture relevant, dynamic context information related to the condition of the mobile client, the service, and/or the end-to-end network in connection with dynamic service discovery. Since it provides a framework for any service-specific selection logic, our context attribute extends the role of service discovery to include server selection capability. By tightly
coupling service selection with service discovery and advertisement, it becomes possible to easily achieve higher scalability, especially in the face of service proliferation (for example, widespread popular and similar services).

We have designed a Class-based Service Propagation protocol (CSP) to guarantee the efficiency of global service discovery, based on the three-tiered architecture. The CSP protocol minimizes storage overhead and service advertisement traffic. We compared the CSP protocol for wide-area service discovery with the IETF Centroid approach through simulation. The simulation result verified the superiority of our CSP protocol over the standard Centroid.

By achieving scalable service discovery and advertisement (through context attributes) and by minimizing the cost of storage and global service advertisement traffic (the Centroid-based CSP protocol), it becomes feasible to design and implement truly ubiquitous service discovery systems.

Service delivery takes place following the service discovery phase to complete the dynamic service acquisition process. Since discovered services may not fit into resource-poor mobile devices, the two phases should be bridged by client device capability context information. Our unique approach to the problem is a thin-client adaptation architecture to deliver a range of light- to heavyweight service to resource-constrained devices. As a proof-of-concept prototype, we developed the µJini proxy system that allows a service to be presented to any class of target devices using virtual thin-client technology.

**Future Research**

It has been shown through a prototype implementation and simulation study that our context-aware, ubiquitous service discovery and delivery architecture properly
addresses the needs of a future service discovery system. However, it can be further enhanced and extended through the following planned tasks:

**Ad hoc service discovery proxy:** To address the requirements and characteristics of mobile services in wireless ad hoc, peer-to-peer networks, we developed a new ad hoc service discovery protocol in a separate work [Lee03]. A partially ad hoc network may have a few fixed nodes that are connected to the wired network in which services are advertised and discovered by the mobile service discovery framework. The next step would be to bridge the two worlds via a proxy residing on the fixed nodes.

**CC/PP support:** The µJini proxy system enables resource-constrained mobile devices to indicate their capability to a service registry. For this, we defined a set of basic device capability attributes as part of the µJini protocol. Our proprietary device capability attributes need to be replaced with W3C CC/PP standard attributes [Kly02] (that is, CC/PP equivalent Java attributes for service discovery and delivery).

**Service matching rule for context attributes:** The current context attribute can be viewed as just a container to hold service matching and recommendation logic. As a Java object, there is virtually no limit to the capability to express sophisticated service selection and context handling logic. It allows service providers flexibility to write their context attributes in the most appropriate way to their service implementations. However, the service instance ranking through the context index normalization may not be as powerful enough to process complicated context handling as a general rule language. It is an interesting idea to adopt a rule language to describe context attribute logic to be evaluated and executed by a rule engine in service registry.
LIST OF REFERENCES


BIOGRAPHICAL SKETCH

Choonhwa Lee received his Bachelor of Science and Master of Science degrees in computer engineering from Seoul National University, South Korea, in 1990 and 1992, respectively. He was then employed by LG Information and Communication Ltd., South Korea, where he was involved in various computer network and telecommunication projects. In 1999, he started his doctoral studies in the Computer and Information Science and Engineering Department (CISE) at the University of Florida in Gainesville. His research interests include mobile computing and networks, pervasive computing, Internet protocols, and network security.