Regions: A Scalable Infrastructure for Scoped Service Location in Ubiquitous Computing

by

Kathryn Flores Benedicto

Submitted to the Department of Electrical Engineering
and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering and Bachelor of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999

Author ....
Department of Electrical Engineering
and Computer Science
May 14, 1999

Certified by ....
Karen R. Sollins
Research Scientist
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Regions: A Scalable Infrastructure for Scoped Service Location in Ubiquitous Computing

by

Kathryn Flores Benedicto

Submitted to the Department of Electrical Engineering
and Computer Science
on May 14, 1999, in partial fulfillment of the
requirements for the degrees of
Master of Engineering and Bachelor of Science in Computer Science

Abstract

Until recently, most efforts in service location have focused on finding local services. However, service location is also useful in large-scale networked environments containing numerous, possibly non-local services. Regions address this need for scalable service location. Regions are groups of services that provide scoped service location by allowing user agents to find services within a region that have certain types or attributes. Regions and the region infrastructure were developed to support ubiquitous computing applications that integrate heterogeneous sets of networked devices and services dynamically. We have designed a scalable architecture for regions which meets the functional requirements of this application domain. This architecture incorporates hierarchy, flexibility, type models, caching, and a model for services. We have developed a prototype implementation of the region infrastructure using Sun Microsystems' Jini technology. We have used this implementation to analyze region performance and examine some of the factors affecting performance.

Keywords: service location, ubiquitous computing, Jini(tm), network scaling

Thesis Supervisor: Karen R. Sollins
Title: Research Scientist

Acknowledgments

I would like to thank Dr. Karen Sollins for the patience, support, and encouragement she has given me over the past three years. I am grateful to have had the opportunity to work for her. I especially appreciate her advocacy and sympathy in helping me battle the adminstrative and bureaucratic demons lurking in Course 6!

I would also like to extend my appreciation to Rose Manela, my lifelong friend. She inspired me to apply to MIT because I have been copying everything she does since I was two.

My sincere gratitude goes to Charles Santori, whose warmth and sense of humor have been nothing less than a lifeline during my college years.

This work is dedicated to my mother, Anna Benedicto, and to the memory of my father, Perfecto Benedicto. The opportunities I have today are the fruits of their hard work and struggles. I will be forever thankful for their support and love and the faith they have in me.

This work has been funded in part by the Defense Advanced Research Projects Agency (DARPA) under contract number 66557.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Two trends are changing the face of computing today. One is the proliferation of networks small and large, wired and wireless. The other is the trend towards ubiquitous computing. This is manifested in the popularity surge of mobile personal information and communication devices, such as cellular phones, PDAs, laptops, and pagers. It is also evident from the number of devices and appliances with embedded computers, such as cars, VCRs, and microwaves. Taken together, these two trends give rise to promising new networks inhabited not only by desktop PCs and printers, but also by devices such as fax machines, PDAs, stereos, microwaves, cellular phones, VCRs, and perhaps even cars and wristwatches.

But what is the best way to exploit these new networks of heterogeneous devices and the services they provide? The applications that will be in the best position to take advantage of these networks will be adaptive applications. They will have a greater awareness of the networked environment in which they operate. They will be able to choose dynamically the networked devices and services that are best suited for the task at hand, such as an e-mail application that uses a visual interface in the user's office and a voice interface in the user's car.

To enable this "greater awareness" of services and devices in the network, an infrastructure must be in place that manages information about services in the network. It must allow services to advertise themselves easily, with little or no human intervention. It must organize services into groups that help define administrative domains, provide scoping, and make large numbers of services easier to handle. It must help users find the services in a group that meet their needs. Lastly, it must scale well in order to tap the potential of a global network of services and devices.

To this end, we propose the concept of regions and the supporting region infrastructure. Regions are scalable groups of services that can be searched using various criteria, such as service type or service attributes. The region infrastructure stores and manages information relating to the services and internal structure of regions. It also manages the transfer of this information between regions, and provides protocols for services to join and leave regions automatically.

The need for regions is best understood in the context of our broader research goal, which will be explained in the following section.

1.1 The Big Picture

Our long-term goal is to create a distributed framework for ubiquitous computing that combines regions with the adaptive applications described above. The applications will be created dynamically by intelligent agents called *catalysts*. These catalysts can discover the services and devices in their physical and/or virtual environment and use them as building blocks to create an application that meets a high-level goal. The result is applications that make full use of the services available in their surroundings, and that can adapt to changes in the set of services available in the network.

These ideas are best illustrated by an example. Consider a homeowner who wishes to install a home security system. Instead of purchasing a pre-built, pre-configured, stand-alone security system, the homeowner could purchase individual components such as alarm sirens and motion detectors and then connect these to his home network. The homeowner could then employ a catalyst whose high-level goal is to build security systems that deter intruders, detect break-ins, and notify the authorities if necessary. The catalyst would discover the new security components, but it would also be aware of services and devices already on the home network, such as PCs, telephones, and controllers for house lights. The catalyst could then construct a home security application that flashes the house lights to deter burglars, uses the motion detectors to sense intruders, sounds the siren to alert neighbors of a break-in, and uses the telephone to call police.

This scenario poses several questions about the infrastructure that must be in place to support it. Three of these questions will be addressed by this work. The first is: how does the catalyst learn about the devices and services in the home network? Secondly, how can it tell that a particular service meets its needs? Finally, how does the catalyst know to stay within the bounds of the home network? Regions are one way to answer to the questions posed here.

1.2 Regions

Regions focus on the problem of sharing information about services. In the example above, the security system catalyst needs to learn about the services in the home network. Using the region system, the catalyst can do this by querying the `Home` region, which is set up by the homeowner to correspond to all services in the home network. Of course, the catalyst must also find out which services are relevant to its purpose. For this reason, regions can be queried for services by desired service type or service attributes. In this way, the catalyst can search for a service of type `Telephone` or an `AlarmSiren` service with an attribute `Volume=110db`.

Regions also help to define scopes by organizing services into groups. When the security system catalyst builds its application, its operations must be restricted to services in the home network's scope only, even if the home network is interconnected with other homes in the neighborhood. It would not be desirable for the home security application to flash the lights of every house on the block!

The flip side of this coin is that regions enable controlled sharing of information across administrative boundaries. If for some reason the homeowner decides that he *does* want to share some of the services within his home, the region infrastructure allows his neighbors to obtain information about those services.

The region infrastructure also includes support for building services that can interact with regions. Its general framework for services includes support for attributes, as well as a protocol for automatically joining, leaving, and sending updated information to regions. It provides a layer of abstraction between regions and the software or hardware providing the service, such as the motion detectors in the home security system. This standardized interface between regions and services encourages development and automates participation in the region system. This opens the door for many new region-enabled services to be developed, further enhancing the value of the region infrastructure.

1.3 Summary

The current trends toward ubiquitous computing and networking are prompting new models of computation which feature many lightweight computing elements interconnected via a network. Our eventual goal is to create a model that harnesses these trends by using catalysts to integrate a wide variety of available devices and services into a single application that meets a high-level goal.

Regions are a piece of the information infrastructure that makes this happen. They group services together, creating natural administrative boundaries and resulting in scopes in which users and catalysts can operate. They can be used to create domains of trust. They match up users with the services they desire by providing lookups of services based on types and attributes. They allow controlled sharing of service information and service use between regions. They support the rapid development of services and facilitate their inclusion into the region infrastructure. By providing these capabilities, regions will contribute to the creation of powerful new networks of heterogeneous devices and and allow them to be used to their fullest potential.

1.4 Coming Attractions

Chapter 2 covers related work, most notably Jini, SLP, DNS, and the Millenium, Inferno, and Infospheres projects.

Chapters 3 and 4 discuss the functional requirements of regions and services and the architecture we designed to satisfy them. Chapter 4 ends with a series of high-level examples that illustrate how the region architecture works.

Chapter 5 describes our prototype implementation of the region infrastructure and the prototype's various components. It gives an example of the prototype system in use, and also contains a discussion about Jini's impact on our implementation.

Chapter 6 delves into the factors that affect the performance of the region infrastructure. It uses a performance analysis of our implementation as a vehicle for discussing issues of efficiency, design tradeoffs, and hidden costs that apply to all region implementations.

Chapter 7 outlines the direction of future efforts and concludes with some ideas for possible future uses of the region infrastructure.

Chapter 2

Related Work

Current trends in ubiquitous computing are driving the development of the region infrastructure. Here, we describe those trends and mention some of their offshoots. We also touch briefly on three infrastructure projects that share some of our goals: Millennium, Inferno, and Infospheres.

There exist many examples of naming and scoping systems that have similarities to regions or attempt to solve some of the same problems. They are not listed comprehensively here. However, three of these systems were heavily influential in the design of the region infrastructure: DNS, SLP, and Jini. We describe them and discuss their contribution to the region infrastructure.

2.1 Trends in Ubiquitous Computing

Recent developments in ubiquitous computing and related fields such as home networking have been the motivating force behind the region infrastructure. New models of computation are being explored in which computation is distributed among numerous, lightweight, networked devices. The popularity of PDAs, mobile communications devices, and appliances with embedded computers has contributed to this movement. New standards for home, personal, and ad-hoc networks are being developed, such as Bluetooth [4] and HAVi [8]. Research is underway that explores the possibilities of ubiquitous computing systems for everyday use, such as a project at Xerox PARC

[18].

As the new computing paradigm gains ground, the need has arisen for supporting infrastructure. The region infrastructure described here attempts to address part of this need. The Millennium, Inferno, and Infospheres projects, which have various degrees of overlap with the region infrastructure, also aim to fill this need. These three projects are described briefly below.

2.2 Millennium

Millennium [3] is a new research effort at Microsoft. Its goal is to build a distributed operating system. Applications that run in this operating system will have their computations distributed automatically and will also have ready access to network resources. The application itself will not need to know the details about the distributed environment or the location of machines and network resources. Stated goals of the Millennium project's distributed operating system include: seamless distribution of computation, worldwide scalability, fault-tolerance, the ability to self-tune, the ability to self-configure by automatically assimilating new network resources and machines, security, and resource controls [2]. Although Millennium's approach of developing a distributed operating system is different from the region infrastructure approach, many of the goals are similar, especially worldwide scalability, self-configuration, location-independent availability of network resources, and resource controls. Millennium is a new project, so its strategy for meeting these goals has not yet been worked out in detail. However, in light of the many shared goals, Millennium deserves attention as it continues to evolve.

2.3 Inferno

Inferno [1] [17], developed by Lucent Technologies, is an operating system that represents resources on the network, as well as system resources, as files in a hierarchical

[1]Inferno and InfernoSpaces are registered trademarks of Lucent Technologies.

file system. This file system metaphor is called a *namespace*. It simplifies the interface between applications and network resources, and also allows applications to access resources without knowing their location. Network resources can be added by grafting on additional namespaces. Namespaces also allows arbitrary groupings of resources by creating a directory containing those resources. InfernoSpaces makes the Inferno technology available to non-Inferno platforms by providing a software development kit that runs on a virtual machine.

Several key ideas in the Inferno namespace model are applicable to regions. One is the ability to create arbitrary groups of network resources. Another is the ability to combine groups dynamically and in a hierarchical fashion through grafting. Location and platform independence are also features of Inferno that are desirable in regions. However, Inferno does not have a model for determining the attributes of network resources, as regions do. In addition, Inferno dictates the model for how clients use network resources. (Clients use an I/O model of reading and writing to virtual files in order to communicate with network resources.) This is outside the scope of regions.

2.4 Infospheres Project

The Caltech Infospheres Project [12] investigates compositional systems and the surrounding issues of compositionality, scalability, dynamic reconfigurability, and high confidence [5]. One interesting product of this research initiative is the Infospheres Infrastructure. This is an architecture and tool set that brings together applications distributed in a network into a virtual network called an Infosphere. The Infosphere can then be harnessed to perform a particular high-level task depending on the applications it contains. The Infospheres Infrastructure software includes tools for building the distributed applications, messaging models for communication between them, and tools for Infosphere management.

The Infospheres Project's focus on the composition of distributed applications to meet a high-level goal is much like our own overall research aim. It approaches the problem from a parallel/distributed computing perspective, though. It uses a model of distributed objects within a virtual network, in contrast to the three-part model used in SLP (Section 2.6) and also in regions (Section 3.2.1), as we shall see. The Infospheres Infrastructure uses formal specifications of interfaces to locate objects that meet a certain set of requirements. Regions, on the other hand, rely on both attributes and a simple interface model to find the services that meet a user's needs. The Infospheres Project also emphasizes low-level issues such as messaging between objects. Such issues are outside the scope of regions, which focus more on service discovery.

2.5 DNS

The Domain Name System (DNS) [11] is relevant because it addresses a central problem in the region infrastructure: the sharing of information across many independently-administered domains. The DNS enables hosts in one Internet domain to obtain name and addressing information about hosts in a different domain. The DNS must provide this service to a large number of domains which are administered independently. In order to preserve this autonomy and maintain high availability and reasonable performance, the DNS works in a decentralized, distributed fashion.

Several lessons were drawn from the DNS for the design of the region system. One lesson is the use of hierarchy to manage the large number of domains. While regions have more flexibility than the strict hierarchy required by the DNS, it is likely that in many application areas, regions will naturally decompose into subregion structures with a large degree of hierarchy. This hierarchy can be exploited for the same benefits it provides in the DNS. Another lesson learned from the DNS is its model for performing queries. Queries received at a higher-level domain are handed down to subsequent subdomains until an authoritative answer is found. Although the region infrastructure does not use the exact same model for performing queries, it does use information flow between regions and subregions to answers queries at a high-level region. The third lesson incorporated from the DNS is its heavy reliance on caching to ensure reasonable performance. Caching plays a more complex role in the region infrastructure because the rate of region information changes can vary greatly between different regions, or even within the same region over a period of time. This is not the common case for the DNS; changes in the DNS occur at a relatively steady rate and are relatively infrequent. However, caching is just as crucial in the region infrastructure as in the DNS, especially for the kind of multiple-result feature-matching queries that regions provide.

2.6 SLP

Service Location Protocol (SLP) [7] is an important conceptual precursor to the region infrastructure. SLP enables users to find networked services with a desired set of characteristics in a particular (usually local) area. For example, a user could use SLP to find the nearest color printer, or to find a printer in a certain building with a certain resolution. SLP relies on a three-part model of user agents which seek services, service agents which provide them, and directory agents which mediate the interaction between the user agents and the service agents. It allows the user agents to query the directory agents for services with a particular feature set. Also, it features a protocol for services to register automatically with local directory agents upon joining the network. This protocol also sends automated updates to the directory agent if changes should occur at the service. Additionally, it includes security mechanisms for authenticating the senders of registration and update messages. The automated registration and update protocol, along with the three-part model used by SLP, appear in later work such as Jini [15], and also in the region infrastructure. Lastly, SLP has a scoping mechanism which limits the set of services considered when a directory agent answers a user agent's query. Each directory agent is associated with a named scope, and every service agent that is part of a scope must register with all directory agents associated with that scope. While this scoping mechanism does not scale up to the proposed size of the region infrastructure, the idea of querying for services within a limited scope is in the same vein as the region concept.

In addition to the poor scalability of the scoping mechanism, SLP has other weaknesses which preclude it from being used to implement regions directly. SLP was designed for finding local services, i. e. services in the same building or on the same campus as the user. Regions, on the other hand, are more general, may contain more users and services, and are not necessarily local. The difference in design goals accounts for the poor scalability of scopes up to regions and the lack of support for hierarchy and nested scopes. Another issue with SLP is the use of keepalive messages and multicast in the automated registration protocol, which can put a potential burden on available bandwidth. Finally, SLP does not have typed scopes, and it does not explicitly support the ability for the same service to register different feature sets within different scopes.

2.7 Jini

Sun Microsystems' Jini [2] [15] technology provides an architecture and tools for facilitating ubiquitous computing. The goal of Jini is to form impromptu heterogeneous networks of devices and instantly make their services available on the network, all with little or no configuration or human intervention. The Jini framework provides protocols for devices to discover nearby Jini networks and join them. It also has mechanisms that allow users to locate services on a Jini network based on the desired service type and the attributes associated with the service. Furthermore, it has support mechanisms, such as leases, transactions, and events, that address the special requirements of distributed systems.

Jini's approach bears a resemblance to that of SLP. It relies on a similar model of users, services, and directories that manage the information and interaction between the users and services. Like SLP, it allows services to be grouped together in different ways. Also, both Jini and SLP have protocols for services to automatically join, leave, and update information within these groups, and these protocols employ both unicast and multicast.

However, Jini has some advantages over SLP. It allows for more flexibility in the

[2] Jini and Java are registered trademarks of Sun Microsystems, Inc.

way a service's attribute sets are organized. Also, unlike SLP, Jini extends to the interaction between the user and the service. A user downloads an object or applet that provides an interface to the service, and runs it on a Java virtual machine. Two benefits of this approach are that it eliminates the need for the client application to know about low-level details such as the communication protocol used by the service, and it exploits Java's platform independence, which is especially useful on the client side. A third benefit is that this model of interaction with services is amenable to both human and machine users. This reflects the fact that Jini was designed with service composition as one of its eventual goals.

As-is, Jini is not entirely suitable for representing regions directly. One issue is that Jini does not explicitly support typed groups. Another issue is that Jini supports only exact-match and wildcard lookups for services. It may be desirable for a region infrastructure to provide more general query support, with options such as value-range and predicate queries. Security measures have yet to be incorporated into Jini. Also, Jini does not provide the level of support for hierarchy that is required by the region infrastructure. While it is possible to connect Jini service directories (called *lookups*) in a hierarchical manner, one must still query sublevels directly in order to learn about the services they provide, rather than just querying a top-level lookup once. Finally, Jini's protocol for services to discover lookups relies on the use of a single, well-known multicast channel. This use of multicast, along with Jini's insufficient support of hierarchy, introduces potential scaling problems for regions, especially those that include a large number of services and users, which are not necessarily grouped by proximity in the network.

To be fair, Jini's scaling problems in the context of regions arise because its design goals do not completely match those of regions. Jini is primarily meant for services and users in the same local area. In this context, the use of multicast is appropriate, and the groups of services are not large enough to require scaling management strategies such as hierarchy.

Jini does meet the goals of the region infrastructure on a small scale, and we believe it can be used as a building block to solve the region infrastructure problem on a large scale. For this reason, we use Jini as the underlying foundation for the region infrastructure. Section 5.7 discusses the impact of Jini on the design of the region infrastructure, and the ways in which the design leverages Jini's strengths while minimizing its weaknesses.

2.8 Summary

The development of the region infrastructure is motivated by the increasing shift from traditional computing models using full-size, dedicated computers to a new model in which the computing elements are lightweight, networked, and far greater in number. This trend is reflected in recent developments in home networking and related fields.

This paradigm shift is creating the need for infrastructure that can combine networked objects in useful ways and make them widely available and easy to locate and use. The region infrastructure is one approach to this problem. Other approaches include Millennium, a new research initiative into distributed operating systems; Inferno, a system which presents network resources as files in a hierarchical file system; and Infospheres, which has produced software for tying together distributed applications into a virtual network.

While many examples of naming and scoping systems exist, the region infrastructure approach derives several of its ideas from three primary sources: DNS, SLP, and Jini. The DNS solves the similar problem of sharing information among a large number of independently-administered domains; its query model and use of hierarchy and caching provide important lessons for the region infrastructure. SLP deals with the problem of service location; its key contributions are its model for managing service and user information and interaction, its use of scopes, and its protocols for automatic service registration within a scope. Jini expands on many of SLP's ideas, solves the region problem at a local level, and provides software that serves as the foundation for the region infrastructure.

Chapter 3

Regions

Regions group services together so that the services can be searched by type and feature. What may not be apparent from this seemingly-simple concept is that regions are powerful enablers. For example, applying the region concept to a home turns it into a home network with easy-to-use, easy-to-configure services. Applying the region concept to a laptop, pager, PC, cellular phone, and PDA turns them into a mobile personal information network in which the devices can exchange information freely and be composed into services more powerful than those provided by each device separately.

This chapter delves into the details of the goals and other requirements of the region infrastructure. It then outlines an architecture for the region infrastructure, and discusses how this architecture meets the needs of regions.

3.1 Functional Requirements

Regions have scoping and service information sharing as their main function. But in addition to these goals, other requirements must be met in order for regions to work in a distributed, ubiquitous computing world. Among these are scalability, independent administration, robustness, and security. All of these requirements are described below.

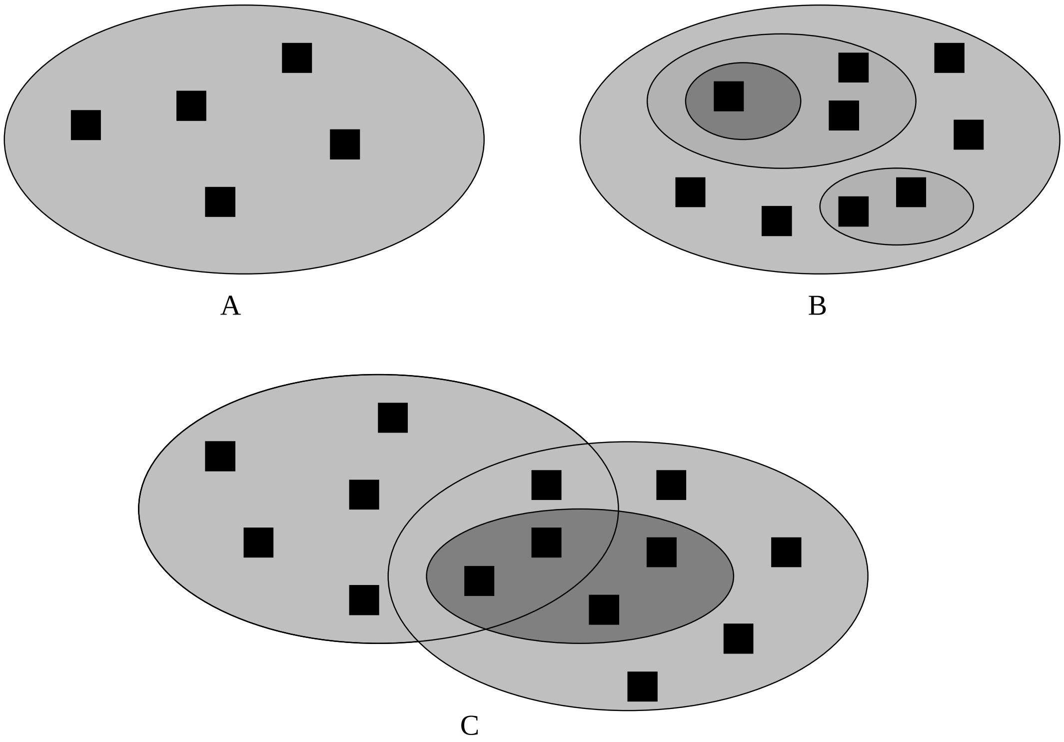


Figure 3-1: Examples of regions. The ovals represent regions and the black squares represent services. A is a region with no nesting. B shows nested regions. C illustrates arbitrary overlap in regions.

3.1.1 Scoping

A region's basic goals are to group services together and share information about those services with user agents and with other regions. Thus, a region must be able to organize a set of services into a group and act on behalf of that group. Furthermore, a region must act as a scoping mechanism for user agents that want to learn about and use its services.

Services may be grouped together by any number of criteria. For example, they may be grouped by physical proximity, ownership, or security level. The region infrastructure should not try to enforce a certain set of criteria; rather, regions should be general enough to support the grouping of services by arbitrary criteria.

Also, regions must provide flexibility in their internal organization. For many users, the model of a region as a simple, flat grouping of services may be satisfactory. However, a region with many services might be more manageable if it is broken up into several subregions that make up a larger region. For example, a region consisting of all the services in a corporation could be decomposed into smaller regions by division, department, etc. This illustrates the importance of supporting nested regions. Regions also require flexible organization in the case of shared, publicly-available services that can be incorporated into any region. Several region owners might want to share a service by including it in their respective regions. For example, officemates might own regions consisting of their individual telephones and PCs and a shared fax machine. In this scenario, a single service may be part of multiple regions that are not necessarily nested. Thus, regions must support arbitrary overlap in addition to flat and nested organizations.

3.1.2 Sharing Service Information

Besides grouping services together, the other major purpose of a region is to facilitate the sharing of information about services. This goal requires a two-pronged approach. One part of this approach is to enable services to join a region and provide information about themselves easily. The other part is to allow user agents to query a region for services that meet the user agents' needs.

In order for services to join regions and share information easily, some kind of protocol is required. This protocol should allow services to locate and join one or more regions of interest, and leave those regions at a later time if desired. The protocol should communicate the service information to the region as part of the process of registering with the region. It should send updates to the region when the service information changes. It should have provisions for dealing with distributed failure (discussed further in 4.2.3). Finally, this protocol for service registration should be automated. Only the service information and its list of regions should need to be configured; the protocol sends the appropriate registration, deregistration, and update messages when this configuration changes. Automating the protocol and minimizing the required amount of configuration is crucial. It opens the door for the development and deployment of many new services which can easily plug themselves into a region and become available to any user agent in the region infrastructure.

In addition to the service protocol mentioned above, regions must provide user agent query support in order to share service information effectively. User agents must be able to query a region with a desired set of characteristics and obtain a list of one or more services with those characteristics. These desired characteristics may include attributes and their values, service types and instances, and functions or functional interfaces. Along with this rich set of characteristics, regions should also support a broad range of queries. User agents should be able to look up services using exact-match, wildcard, predicate, and value-range queries.

3.1.3 Scalability

One common aspect of the other systems on which regions are based, such as Jini and SLP, is that they aim to solve the grouping/scoping and service location problems on a local scale [7, 15]. They tend to adhere to assumptions about the number of services in a group and the services' proximity in the network or in the physical world. Regions are more general than this—and thus more powerful. They are intended to operate in a world with large numbers of services, and in which the services grouped together in a region may in fact be widely-dispersed geographically or topologically. One example is the use of regions to manage the networked services of a worldwide corporation. As a result of this design goal, scalability is absolutely essential to the region infrastructure, to deal with both the large number of services and the potentially large distances between them.

3.1.4 Independent Administration

The world in which regions operate is too big to assume that all regions will be administered by a single owner, or even a handful of cooperating owners. A better model is one similar to the DNS model [11], in which administrators manage their regions independently of each other, but service information can be shared across regions. This model of autonomous regions lends itself to a distributed, decentralized mode of operation. While it may contribute to complexity in some respects, it carves up the region world into more manageable chunks, and it obviates region owners' dependence on a central authority (in which this model differs from DNS). It also avoids having a single point of failure, and can help isolate regions from problems in other regions.

3.1.5 Typing

Regions need a type model, especially in the context of interoperation with catalysts. A catalyst is just a specialized user agent; it synthesizes applications through composition of services in its environment, which is a region or set of regions. To this end, it is helpful for a catalyst to be able to have some expectations about the region in which it operates. It should have an idea of what kinds of services and attributes it might find in the region, without having to perform a comprehensive search. This facilitates the synthesis of applications. Hence, regions should be typed.

What does it mean for a region to have a type? In keeping with the purpose of region types as guidelines to catalysts, region types are specified as requirements and restrictions on the attributes, attribute values, service types, and functional interfaces in a region's services. For example, a region of type `Kitchen` might be required to have services of type `Microwave`, `Dishwasher`, and `Toaster`. Region types may also include attributes and attribute values that are common to all instances of a type, much like non-static and static field variables in a class. In addition, region types follow the usual rules for inheritance and other features commonly found in type models.

3.1.6 Robustness

Regions need to be robust. In particular, the various parts of the region infrastructure should be able to detect and recover from distributed failure. The service protocol should account for the possibility of the service being cut off from its containing region at any point during registration, update, and deregistration. The user agent may want to have a policy in place for retrying failed queries or detecting when a region is up or down. Most importantly, regions, which obtain information from their subregions and services, should be prepared to cope with losses, delays, and reordering of information messages sent by those subregions and services. Furthermore, they should have a method for restoring and maintaining a fairly consistent view of the world.

3.1.7 Security

A region allows user agents and other regions to obtain information about it and the services that it contains. It also permits the use of those services. Clearly, this has security implications. We will not attempt to be exhaustive about the security requirements of regions here, nor will we emphasize security mechanisms in our architecture; that is a subject for future work. However, we will touch on some important points in this area.

Authorization and access control are key parts of the security picture. Regions may wish to impose restrictions on which services may join them, which user agents may query them, and what service information user agents are allowed to see. Likewise, regions and services may wish to impose restrictions on which user agents may actually use the service.

Authentication is also important, both for access control as described above, and for the information updates that pass between regions, their subregions, and their services. A region's knowledge of its services comes from directly-contained services and from subregions which contain services. This information is conveyed in the form of join and update messages from services and subregions. As a consequence, falsified update messages can wreak havoc with a region's view of the world.

The region type model also introduces some trust issues. User agents and services may interact differently with a region depending on its type. Services, in particular, can expose a different interface and attribute set depending on the region in which they are. The behavior of catalysts can also be influenced by the type of the region in which they operate. This leads to the question of where a region's type and inheritance information is stored, and whether or not this information can be trusted. Digital

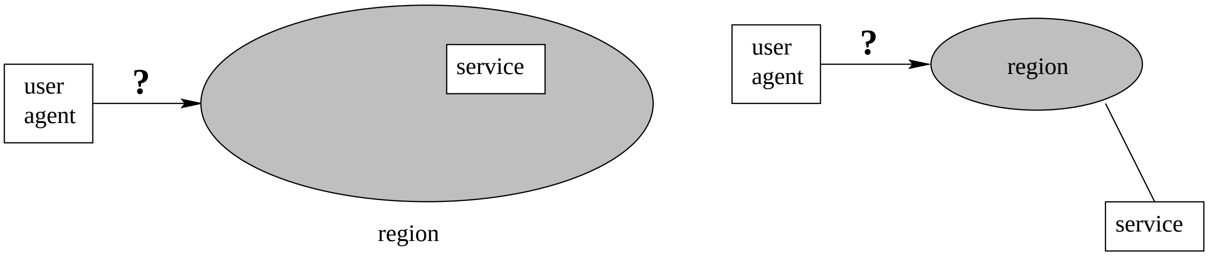


Figure 3-2: Left: The three-part model. Regions act as intermediaries between user agents and services. Right: Same, but depicted as a tree in which the service is a child of the region.

signatures are one possible strategy for making such information trustworthy.

3.2 Architecture

Here we describe a design for the region architecture that meets the above requirements. The design uses a three-part model for regions, which is extended with hierarchy to provide more flexible region structures. It employs a Region Manager, which maintains service, structure, and region type information and answers queries about services in the region. Lastly, the architecture for regions also includes URN resolution services and a region type model.

3.2.1 The Three-Part Model

The most basic regions are based directly on the three-part model described in SLP [7] and later in Jini [15]. This model consists of a user agent, a service, and an agent, called a Region Manager, that represents the region.

User Agent

A user agent is a client that uses one or more services. One example of a user agent is a word processing program that uses a printer service to print documents. In general, a user agent initially does not know which service to use, but it does know what characteristics the service should have. User agents use the region infrastructure to match them up with services that have this set of desired characteristics. They query a region for services with the desired characteristics. The region responds with a list of service entries describing the matching services and providing handles to them. The user agent can then use the services.

Service

A service is an object that performs some set of functions for user agents. One example is a printer service. The functionality of a service is specified by its type, its attributes and their values, and a functional interface. In the case of the printer service, its service type may be `Printer`, and its attributes and values may include `ColorType=BLACK_AND_WHITE` and `Resolution=600dpi`. The functional interface might include the functions `printSingleSided()` and `printDoubleSided()`; these functions could be called by user agents.

Services are described more fully in Chapter 4.

Region Manager

The Region Manager is an agent that embodies the region. It manages the information about how the region is organized and which services it contains. It keeps track of services as they join, leave, and change their attributes. The Region Manager also acts as a intermediary between user agents and the region's services by providing service information to user agents in response to their queries.

The Region Manager's role is analogous to the Directory Agent in SLP [7] and the Lookup Service in Jini [15], but its capabilities and the information it contains are a superset of those in SLP and Jini. The additional capabilities are for managing the more complex internal structure found in regions. These capabilities are described later in Section 3.2.3, which discusses the Region Manager's role in greater detail.

3.2.2 Structure and Hierarchy in Regions

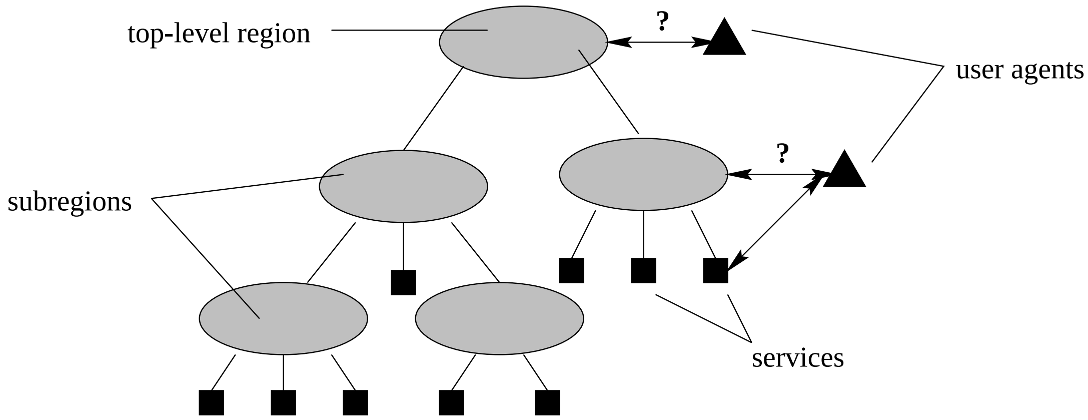


Figure 3-3: Region based on extended three-part model with hierarchy.

The basic three-part model described above, with its flat collection of services, is insufficient for most regions because of the scaling and administrative requirements of regions. To accommodate these requirements, the region infrastructure uses hierarchy to extend the three-part model and create regions with more complex and flexible structures. In this extended model, not only may regions contain services, but they may also contain other regions. Regions take on a role similar to services: they may join and leave other regions, and when their information changes, they must update their containing regions. This enables regions like the one in Figure 3-3 to be created.

The new model for structuring regions requires more flexibility than a strict hierarchy provides, however. One way in which it differs from a strict hierarchy is that it allows a region to be contained in two or more regions that are not necessarily nested. In other words, a region can have multiple parent regions. This provides for arbitrary overlap of subregions as well as services. It also means that multiple paths may exist to the same service or subregion within a region hierarchy. Also, a region may directly contain subregions, services, or a mixture of both. Figure 3-4 illustrates the variety of regions that can be built with this flexible model.

In this new model for regions, what does it mean for a region to "contain" a service? We take the position that a region contains a service if and only if the service belongs to the region directly, or belongs to any of the region's subregions, ignoring access control for the moment. A similar definition applies for containment of one region inside another. We also take the position that a region cannot contain itself

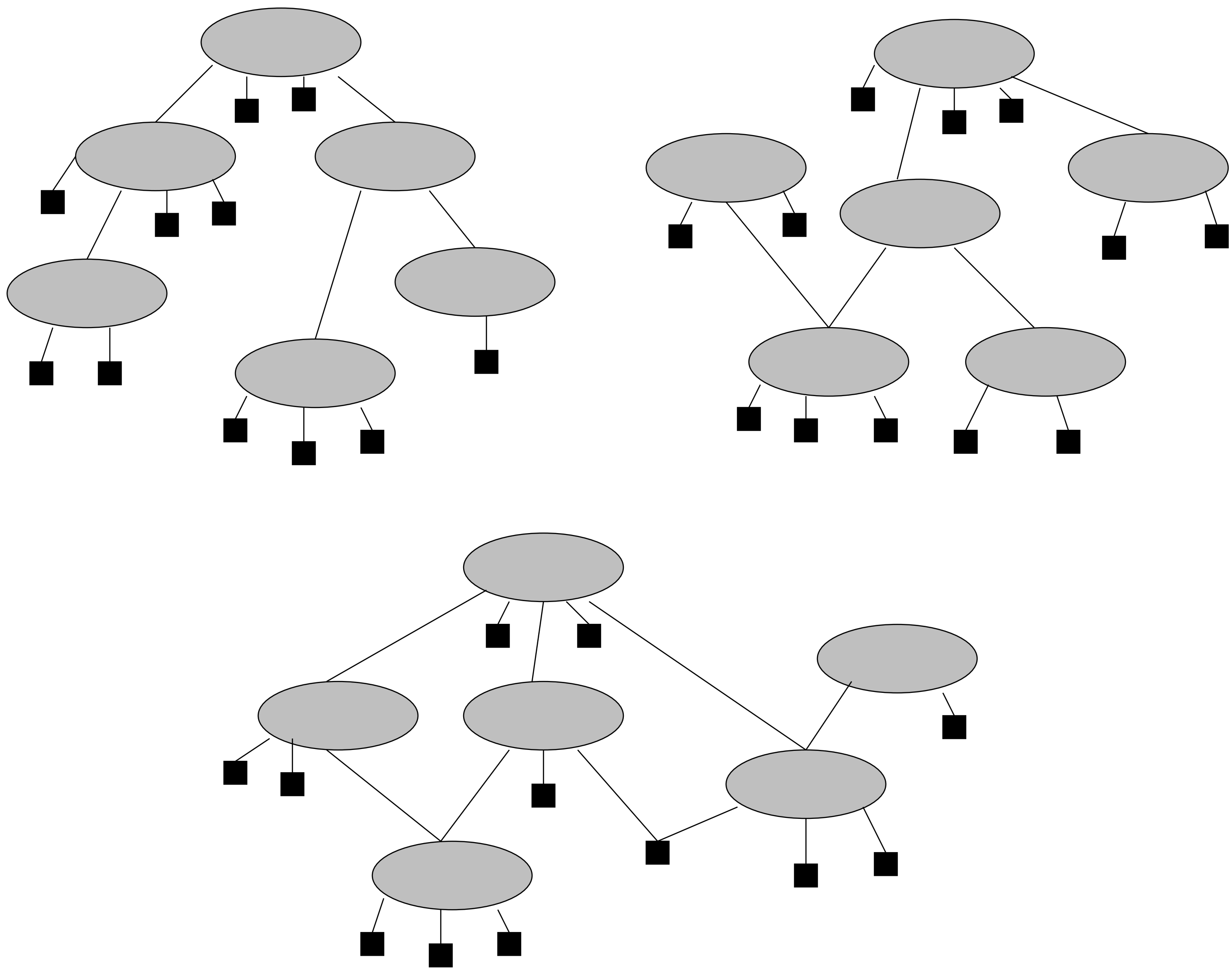


Figure 3-4: Examples of structured regions. Note that both services and subregions can be shared by two or more parents. Also note that two of the region graphs show more than one top-level region.

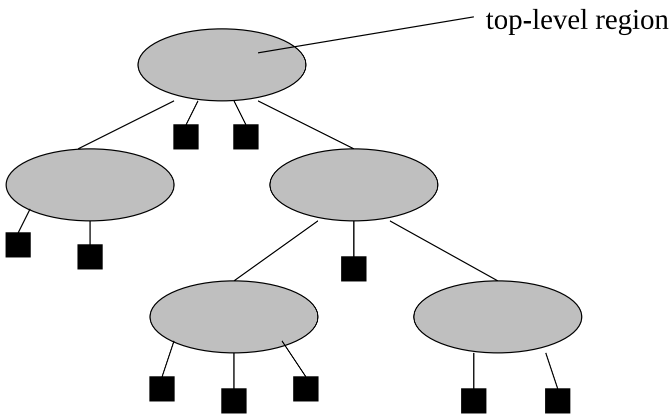


Figure 3-5: Top-level region, subregions, and services. The top-level region contains four subregions. All ten services shown in the figure are contained by the top-level region.

directly or indirectly, because such a relationship is not useful for service location. By applying this notion of containment to Figure 3-5, one can see that all services shown in the figure are contained by the top-level region, and that the top-level region contains four subregions. This containment model is especially useful as it applies to services. A region possesses, or can obtain, information about all of the services it contains. As a result, a user agent can search among all of a region's services and all of its subregions' services with a single query.

It is important to note that containment of a service or a subregion does not imply control over it or the ability to get complete information about it. It is possible to build regions in which containment relationships reflect delegation of authority. But the region infrastructure should also support controlled sharing between regions that do not necessarily trust each other. For this reason, a region at any level can choose to hide some of its service information from its parents. An example of this would be a region corresponding to a business center of the kind commonly found in business hotels. Such a region might contain PC, printer, telephone, fax, and copier services. The manager of the business center might include the business center region in her personal region, in order to perform administrative tasks. A hotel guest might include the business center region in her personal region for the duration of her stay, in order to make use of its services. Although both use the business center as a subregion, the business center region may permit the manager to access administrative services that are hidden from the hotel guest.

The model for region structure, which incorporates both flexibility and elements of hierarchy, satisfies several of the functional requirements of regions. It provides a scoping mechanism that allows nested scopes and the arbitrary overlap of regions. It contributes to scalability by permitting regions with a large number of services to be organized into a hierarchy of subregions, each of which manages only a small number of services directly. It facilitates the sharing of service information by allowing a region to include a subregion that it may not necessarily control, obtain information from that subregion, and share it with user agents. At the same time, it preserves the ability of regions to be administered independently, and bolsters security by allowing subregions to hide service information from parents.

3.2.3 The Region Manager

We now turn to the question of how the individual regions manage their interactions with user agents, services, and each other. These interactions are handled by the Region Manager. The Region Manager stores information about the structure of the region and the services it contains. It then uses this information to perform its management tasks. These tasks are: to keep track of regions and services as they join, leave, and modify their attributes; to perform region type-checking when the region is created and when its membership changes; to respond to user agent queries; to send and receive information updates from other region managers; and to cache information when appropriate.

First and foremost, the Region Manager stores information about the services it contains directly. This information includes the attributes, attribute values, types, functional interface, identifier, and handle for each service.

The Region Manager also uses information about services it contains indirectly. In addition to the service information listed above, it also stores those services' parent regions, i. e., the regions that contain them directly. Information about indirectly-contained services may be cached in the Region Manager, or may be obtained directly from a subregion's Region Manager. Caching is discussed in Section 3.2.5.

Along with service information, the Region Manager stores information about the structure of the region in the form of the interconnections between the subregions. This information, together with the service information, is sufficient for reconstructing the complete subtree consisting of the region, its descendents, its services, and the subregion to which each service belongs. This is enough information for the Region Manager to perform its management tasks.

One of these management tasks is keeping track of services as they join and leave the region and its subregions, and updating service information when services change their attributes or attribute values. Likewise, the Region Manager must keep track of subregions as they are added and removed, and how these changes affect the set of all services contained by the region.

A related task is type-checking of services and subregions. Type-checking occurs at the time the region is created, in order to ensure that none of the services or subregions violate the requirements imposed by the region type. However, type-checking is a continuous process because regions are dynamic; their services and subregions change. The type model is described further in 3.2.6.

One critical task of the Region Manager is its caching of service and structure information and its handling of information update messages passed between it and other Region Managers. The Region Manager uses these messages to update its knowledge about subregions and their services. Caching and update messages are described more in 3.2.5.

Of course, management tasks aside, the fundamental role of the Region Manager is to help user agents find the services they need within the scope of the region. The Region Manager manages information about the services it contains so that it can respond to user agent queries for services.

The Region Manager contributes to the fulfillment of several functional requirements. It helps manage the sharing of service information by tracking changes in subregions and services, and by supporting user agent queries. It enforces region typing through type-checking of subregions and services. Also, the natural place to put mechanisms for robustness and security is in the Region Manager.

3.2.4 Service Queries

A central role of the region infrastructure is to help user agents find services according to the kind of service the user agent wants and the features it should have. This role is filled by the Region Manager, which supports service queries from user agents. These queries specify the characteristics that the service must have. In response to these queries, the Region Manager searches among its local services as well as those of its subregions in order to find one or more services that fit the description.

Queries for services use three criteria: service type, identifier, and attributes. The service type refers to both the type of a service object as well as the types of any functional interfaces it may implement. The service identifier is unique for each service and provides a way to request a particular service instance. The attributes allow the user agent to specify the service features it needs. The user agent can specify which attribute types should be present; in addition, it can specify the values that the attributes should have.

Service queries may be expressed in a number of ways. The query format may include exact-match queries, wildcards, predicate-based queries, and value-range queries on the service type, identifier, and attributes. Ideally, the Region Manager should support this broad array of queries. For now, our architecture and implementation use a query scheme in which either a wildcard or an exact-match value is given for the service ID, type, and attributes.

One issue which should be mentioned at this point is the specification language for service attributes. What attributes are important for characterizing a particular service, such as a printer? What format or schema should be used for expressing these attributes? Although these questions are not addressed here, they are nonetheless significant. Some clues to the answers may be found in [9] and [10].

Another key feature of region query support is that a user agent need only query one region in order to find out about its subregions' services as well as its own local services. Internally, the Region Manager may rely on a combination of cached information and subregion queries for assembling its response to the user agent. The number of required subregion queries depends on how the Region Manager is configured for caching. Note, however, that if the Region Manager relies entirely on cached information, no subregion queries may be necessary.

3.2.5 Caching and Updates

The caching of service and structure information is crucial for maintaining scalability and adequate performance in the region architecture. Caching is necessitated by the sheer volume of services that may be present in a region. It allows Region Managers to answer user agent queries in a reasonable amount of time without resorting to extra subregion queries. These have the negative effects of slowing down response time and generating excess network traffic.

Caching and update strategy also deserve special attention because they are heavily dependent on characteristics which are unique to each region. An example of one such characteristic is the rate at which services join, leave, and change their attributes. Regions in a networking research facility are subject to frequent changes as services go on- and off-line and change their attributes. In contrast, services in a home network region may be very static. Clearly, a caching strategy that works in one region may not work in another. Performance in a particular region depends on how well its caching and update policy is tuned to its specific characteristics.

Another region "characteristic" that affects the caching strategy is the region policy. The region administrator must make certain policy decisions about the amount of cache inconsistency permissible in exchange for better performance. Policy decisions, as well as region characteristics, are reflected in the caching strategy.

For the reasons given above, we do not attempt to prescribe a caching and update strategy for use with all regions. Rather, the region architecture supports a wide range of caching and update policies; the only requirement is that updates propagate from subregions to their parents. This gives administrators the freedom to make policy decisions on which subregions to cache; whether to use full updates, incremental updates, or both; and whether to use batch or individual updates. Administrators are free to implement higher-level policies, such as limiting the rate at which a parent region receives updates from its children, in terms of bandwidth or number of updates received per unit time. Administrators can also choose values for parameters such as time between updates. In this way, region administrators can pick the caching and update strategy and parameter values best suited for the conditions in their region.

Since caching and updates have such an impact on performance, they are also discussed in Section 6.1.3.

3.2.6 Typing

Region types are necessary to provide user agents with reasonable expectations about what services they will find in a region. Our ideal region type model would express a region type in terms of the service types and subregion types contained by a region of that type, as well as the attributes supported by the region type. Service types would in turn be expressed in terms of the attributes and functional interfaces supported by services of that type.

However, we have not yet formulated such a type model because it is unclear what role attributes should play within it. This confusion stems from two reasons. The first is that our service architecture allows a service's attributes and their values to be specified independently of its functional interface. This gives services a greater degree of flexibility, but is in opposition to a type model that associates certain attributes with certain functional interfaces under the umbrella of a single type. The second reason is that attributes can be imposed onto a region or service from an outside source. The most common occurrence of this is when a child region or service joins a parent region and inherits some of the parent region's attributes, as described later in this section. In this case, the parent region is imposing attributes on the child which may or may not be relevant to the child's type. It is not yet understood what effect these imposed attributes should have within the type model.

For now, the region infrastructure uses the following type model. Region types exist as constraints on the types of a region's subregions and on the attributes and functional interfaces of its services. A formalized notion of "service type" does not really play a role in this model, due to the reasons given above. (However, Section 4.2.2

contains a discussion of services and the possible relationships between attributes, functional interfaces, and service type.) The constraints for each type are expressed as a set of rules, and type-checking is the process of checking that a region adheres to the rules for its type. Some examples of these rules are:

- A region of type T_1 must contain at least one service with the functional interface I_1.
- A region of type T_2 can only contain services with functional interfaces I_1, I_2, or I_3.
- Every service in a region of type T_3 must have attributes A_1 and A_2, with attribute A_1 having a value of x.
- A region of type T_4 can only have regions of type T_1 and T_2 as its subregions.

A region type may also specify attributes and attribute values that are associated with the region itself, much like non-static and static field variables in a class. These attributes and attribute values are inherited by subtypes of the region type. Optionally, they can be inherited by services that join regions of that type. For example, a region type may have an attribute/value pair `Owner=MIT` which may be inherited by objects that join regions of that type.

Each Region Manager stores its own type information. This information includes the region type and supertypes and the type inheritance relationships. When a service joins a region, it consults the region for its type information, and then the service exposes the appropriate attributes and interface for that region type and instance (see Section 4.2.2). Of course, this has security ramifications—it assumes that Region Managers are honest about their type information. It does not prevent Region Managers from claiming to have a "privileged" type which is allowed access to all of a service's attributes and interfaces. Future versions of the region architecture will have to deal with this issue, possibly by using a trusted third party for storing type information, or by applying digital signatures to type information.

Type-checking for each region is performed by the Region Manager, both when the region is created and on a continuing basis as services and subregions join, leave, and change. The region architecture provides a basic Region Manager, which developers can extend with typing rules to create typed regions.

The region infrastructure typing model must eventually include types for services, in addition to types for regions. This is so that user agents can have reasonable expectations about what a service does. User agents themselves may have types, although that is not as integral to the region infrastructure as service and region typing are.

3.2.7 URN Resolution Service

One more seemingly-minor piece is necessary to complete the region infrastructure. Given the name of a region, how does one contact its Region Manager? In addition to a region's name, one must also know where to look for its Region Manager. In our architecture, this location is phrased as a Jini URL with the format `jini://`*hostname*`[:`*port*`]`. But how does one discover the mapping from a region name to its Jini URL?

Likewise, suppose one knows the identifier of a service but cannot query the service directly. Then how does one find out the regions to which the services belongs, so that the regions can be queried for service information instead?

The answer to problems like these is to incorporate URN resolution services into the architecture [14, 13]. These services map resource names to their locations. Within the region infrastructure, URN resolution services may map region names to Jini URLs, or they may map service names to their parent region names and/or URLs. By doing this, the URN resolution services in the region infrastructure enable communication with regions while keeping region and service names separate from and independent of their locations.

3.3 Summary

The main goal of the region infrastructure is to share information about services within a scope. At the same time, it must allow these scopes to be administered independently. Also, these scopes must be flexible enough in structure to handle large numbers of services which may be widely dispersed. In addition, regions must provide a typing model so that user agents can exploit their services effectively. Finally, regions must be sufficiently robust and secure for use in a distributed setting.

The region architecture is a framework for meeting these needs. Its basic building block is the three-part model of user agents, services, and Region Managers which govern the interaction between the services and user agents. Hierarchy is used to extend this model to more flexible structures which incorporate nesting and arbitrary overlap. Hierarchy also contributes to scalability and allows regions to be partitioned according to administrative policy and/or functionality. Within the extended model, the Region Manager keeps track of information about the region's services and internal structure. It provides user agents with service information in response to their queries. To ensure good performance, the Region Manager relies on caching and update strategies that are geared to its region's particular characteristics and the administrator's policy needs. The Region Manager also manages region typing information and performs the necessary type-checking. Finally, URN resolution services help user agents obtain the location of a particular Region Manager or the parent regions of a service.

Chapter 4

Services

Services are the basic units of functionality that are grouped together into regions. In this chapter, we explore the functional requirements for a service, in the context of the special functions it provides and also in the context of its interactions with regions. We then describe an architecture for services that handles these interactions and can be extended to provide any desired service functions. We end with examples that demonstrate how regions, services, and user agents work together.

4.1 Functional Requirements

Services have several functional requirements. Some stem from their role of providing functionality to user agents. Others, such as security and region-dependent functionality, arise from their interaction with regions. Still other requirements, such as typing and ease of configuration, belong to both categories. The various functional requirements of services are described below.

4.1.1 Service Function

A service must provide some kind of functionality of which user agents can make use. This functionality should be expressed through service attributes and functional interfaces. A user agent can obtain this information in order to use the service or decide whether or not the service is suitable for its needs. Also, the service type (see Section 4.1.5) provides hints about a service's functionality.

4.1.2 Integration with Regions

Services must integrate readily with the region infrastructure described in the previous chapter. They should capable of joining and leaving regions, and should also report any changes in their attributes, attribute values, or functional interfaces to their parent regions. To facilitate the sharing of service information across regions and user agents, services should have interfaces and mechanisms for providing and communicating their information.

4.1.3 Region-Dependent Functionality

A service should be able to expose different subsets of its functionality to different region types or instances. This means that two regions may directly contain the same service, but each region "sees" a different attribute set and functional interface for that service.

This serves two purposes. The first, obviously, is access control. Trusted regions may be able to access a larger set of functional interfaces and attributes than untrusted regions. The second reason for region-dependent functionality is that a service or device may have multiple features and functions, but only some of them are relevant in a particular region setting. For example, in a home network, a universal remote control might expose different features in a kitchen region than in an entertainment region.

4.1.4 Ease of Configuration

Services should also be easy to configure, especially with respect to regions. Services may require some configuration in regard to the functionality they provide to user agents; this should be kept to a minimum. However, a service should require little, if any, configuration in order to take part in the region infrastructure, join and leave regions, and perform other region-related tasks. The goal is to eliminate barriers to the development of region-capable services. The simpler a service is to configure, the more services will participate in the region infrastructure. This greatly enhances the region infrastructure, whose power lies in the ability to make large collections of services available to any user agent.

4.1.5 Typing

Services should have types. As is the case with region types, service types provide valuable information about what user agents and regions may expect of a service, in terms of its interface and attributes. This is helpful for user agents, especially ones like catalysts, which compose larger services from smaller ones and stand to benefit greatly from an object-oriented type model. Also, the service type model is part of the region type model; certain region types may put restrictions on the services they contain based on the service type.

4.1.6 Security

As with regions, service security will not be given a comprehensive treatment here or in the following architecture description. However, we will touch on a few of the security issues for services. Several of these were already mentioned in Section 3.1.7: access control and authentication for user agents who wish to use the service, and authentication of services when they join a region or send information update messages. Services also have type information trust issues similar to those of regions; namely, they can falsely claim to have a privileged type or supertype which allows them to join regions they could not join otherwise. The measures suggested in Section 3.1.7 for making region type information trustworthy can also be applied here.

4.2 Architecture

An architecture for services must enable a service to provide a specific kind of functionality while maintaining a common interface to the rest of the region infrastructure. It must also meet the requirements above. We describe a framework for services that maintains special information about regions and region-dependent features, and participates in a registration and update protocol with regions.

4.2.1 Service

Every service must manage its interaction with regions. It must also store information about regions along with information about its own functionality. The service architecture provides a generic service which handles this information and manages the interaction with regions. This generic service can be extended with additional functionality to create a specific service type, such as a printing service or a voice mail service.

Each service has a unique identifier. This serves to differentiate between service instances. Even if a service exposes different attributes and interfaces in different regions, the identifier can be used to determine whether the apparently-different services are actually one and the same. This information is important for various purposes, such as load-balancing between services. Also, it is important because a user agent may get unexpected results from using two services if it assumes that they are distinct, but in fact they are implemented by the same underlying object (e. g., a database being modified by the user agent).

In addition to its identifier, each service stores its attributes, attribute values, and information about the functional interfaces it provides. Another vital piece of information that services store is which attributes and interfaces to expose in which region types and instances. A mapping from a region name or type to an attribute set/interface pair is called a *view*, and these mappings are collected into a table of views, which is consulted whenever a service joins a region. A default view is provided for regions and region types not specifically listed in the view table. Views are discussed further in the next section.

Services also maintain a list of the parent regions to which they belong. This list can be modified by the administrator of the service. Services use this list to join and leave regions and send information update messages to regions as appropriate.

Services interact with regions via the registration and update protocol. Services use this protocol for joining, leaving, and updating their parent regions. This protocol is the means by which regions obtain information from services they contain directly; services send their information when they register to join a region and in subsequent updates. This protocol is discussed in Section 4.2.3.

Services provide a handle that allows user agents to interact with them. Currently, this handle is an object stub which performs remote procedure calls to the service. Presumably other forms of service handles will be available in the region infrastructure in later versions. User agents can obtain the service handle from the Region Manager of a region when they query it for services.

4.2.2 Views

The service architecture uses views to satisfy the requirement that a service should be able to expose different interfaces and features to different regions or region types. A view is a mapping from a region name or type to an interface and attribute set. The service keeps its set of views in a table.

Before a service joins a region, it must determine the appropriate interface and attribute set to use for its registration. This is dependent on the name and type of the region and is determined using the following algorithm. First, the service looks up the region name in the view table. If no matching view is found, it looks up the region's type and supertypes in the view table, starting with the region's immediate type and proceeding up the type hierarchy. If a matching view is not found this way, the service falls back on a default view. The service then registers with the region using the interface and attributes in the selected view. If a service has already joined a region, the same algorithm is used to send updates to the region when changes are made to the service's attributes or interfaces.

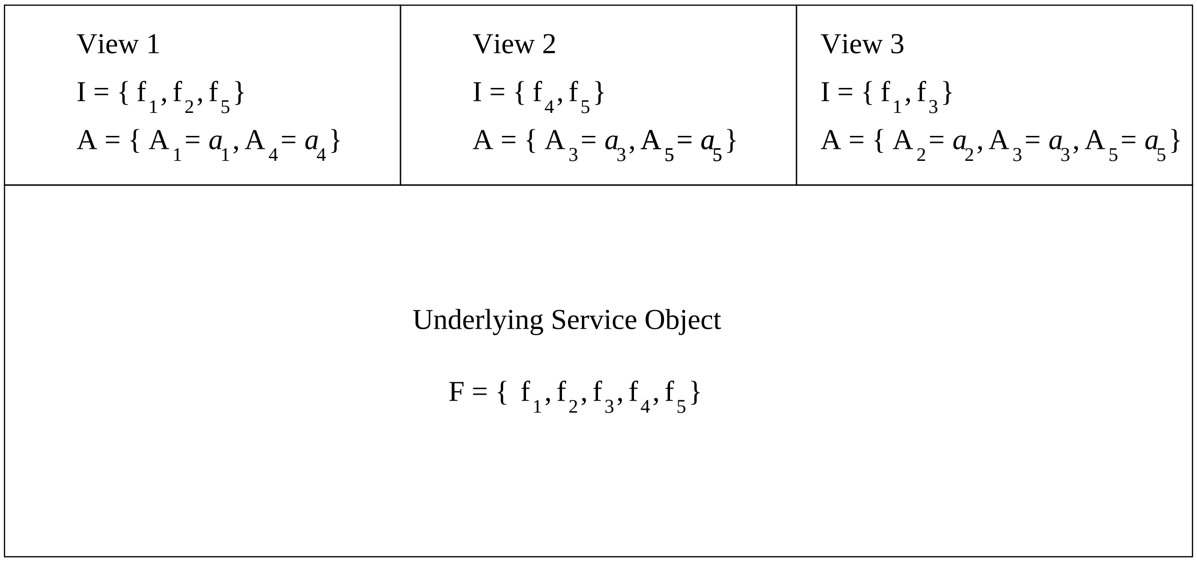


Figure 4-1: A service and its views. F is the total set of functions supported by the service. For each view, A is the set of attributes, and interface I is the subset of F that the view makes available.

It is worthwhile to note that the views described here bear a resemblance to the service types desired in the ideal type model of Section 3.2.6. They combine a functional interface with a set of attributes to be exposed in a region. However, for the reasons stated in Section 3.2.6, views are unsuitable for use as types at this time. Views operate according to the model in Figure 4-1. Each view shows a subset of the total set of functions supported by the service. Each view also shows its own set of attributes. However, views allow their functional interface and their attribute set to be specified independently, while types imply more of a constraint on which attributes are associated with which interfaces. Ultimately, service types will probably incorporate both functional interfaces and attribute sets in their definition, but at this point it is not clear how.

4.2.3 Registration and Update Protocol

The registration and update protocol serves as the interface between services and the rest of the region infrastructure. Services use it to indicate their desire to join or leave a region and to communicate the necessary attributes and information to the region. The protocol is modeled on the Jini Discovery and Join specification [16] both conceptually and in its implementation; in fact, our implementation of the region infrastructure makes substantial direct use of the Discovery and Join protocols and implementation.

The registration and update protocol consists of several phases. First, a service must locate and establish contact with the Region Manager of the region it wants to join. Then the service joins the region, sending its attribute and interface information to the Region Manager. Subsequently, the service may change its service information; it sends the appropriate updates to the Region Manager. Finally, a service can notify the Region Manager that it wishes to leave the region.

The first step, locating and establishing contact with the Region Manager, can happen in several ways. The most general way is for the service to contact a URN resolution service to obtain a URL that can be used to contact the Region Manager for a given region. This method is appropriate for regions whose members are widely-dispersed as well as regions with local members, e. g., regions situated across WANs as well as in LANs. Another method is to use a protocol akin to the Jini multicast request and multicast announcement protocols, which are part of the Discovery and Join specification. In the multicast request protocol, services multicast a message indicating their desire to join one or more particular regions, and the relevant Region Managers respond. Regions use the multicast announcement protocol to periodically advertise their existence and their corresponding URL. In this way, services can circumvent the URN resolution service when contacting a Region Manager. This method is only appropriate when the entities in the protocol are in the same local network or are reasonably close by. However, it does allow services to learn which regions are available in the local network.

Once the service has contacted the appropriate Region Manager, it registers to join the region. It sends a message which includes the attributes and interface it is using to join this particular region (determined using the algorithm in the previous section). When the service changes, it sends update messages to the Region Manager with the new information. Finally, when a service wants to leave a region, it notifies the Region Manager, which deregisters the service and performs various cleanup tasks.

It should be noted that the messages sent and received in the registration and update protocol can trigger events in the caching scheme, if one is in use. Most notably, the Region Manager cache information may be modified, and join, leave, and update messages from a service may cause updates to propagate up to a region's parents and ancestors, informing them of a change in the region's state.

It should also be noted that the process of joining and leaving a region is meant to occur as a part of service startup and shutdown. When the protocol is used this way, services automatically integrate themselves into the region infrastructure without user intervention. The only information that must be configured beforehand is the list of regions for the service to join initially and the table of views. This contributes to ease of configuration.

The protocol outlined here is virtually identical to those described in the Discovery and Join specification. The important differences are the use of a URN resolution service, the existence of an analogous protocol for regions to join and leave other regions (see Section 3.2.2), the propagation of update information in a region hierarchy, and the role of the Jini lookup service being played by the Region Manager instead. These changes augment the Jini protocols by adapting them for use in large-scale regions; Jini already presents a reasonable solution for the small-scale problem.

Details on the protocols in the Discovery and Join specification can be found in [16]. Notably, these details include mechanisms for addressing issues of distributed operation, such as using randomized timers to prevent response floods and using leases as an expiration mechanism for service registrations.

4.3 Examples

Now that we have laid out the functional requirements and architecture for both regions and services, we present some examples that illustrate the workings of the region infrastructure at a high level.

The following examples assume a caching and update model in which Region Managers cache information about all of the services they contain, and update messages are sent immediately and individually (as opposed to in a batch).

4.3.1 Region Join

Figure 4-2 shows region R4 joining region R2 and becoming one of its subregions (and hence, one of R1's subregions). For simplicity, services are not shown in the diagram.

1. R4 must find out the location of R2's Region Manager, so it queries a URN resolution service.

2. The URN resolution service returns the URL `jini://host.foo.org:33832` for R2.

3. R4 sends R2 a message saying that it wants to join. The message includes information about R4's structure and services. R2 updates its own internal information to reflect the new subregion.

4. R2 sends an update message to its parent, R1, to notify it about R4 joining. The message contains all of the information in the original join message, along with the name of the region being joined (R2). R1 also updates its internal information.

4.3.2 Region Leave

Figure 4-3 shows region R6 leaving its parent region, R4.

1. R6 sends R4 a message saying that it wants to leave. R4 updates its internal information to reflect the change.

2. R4 sends an update message to its parent, R2, to notify it that R6 has left R4. R4 updates its internal information to reflect the change.

3. R2 propagates the update up the hierarchy to its parent.

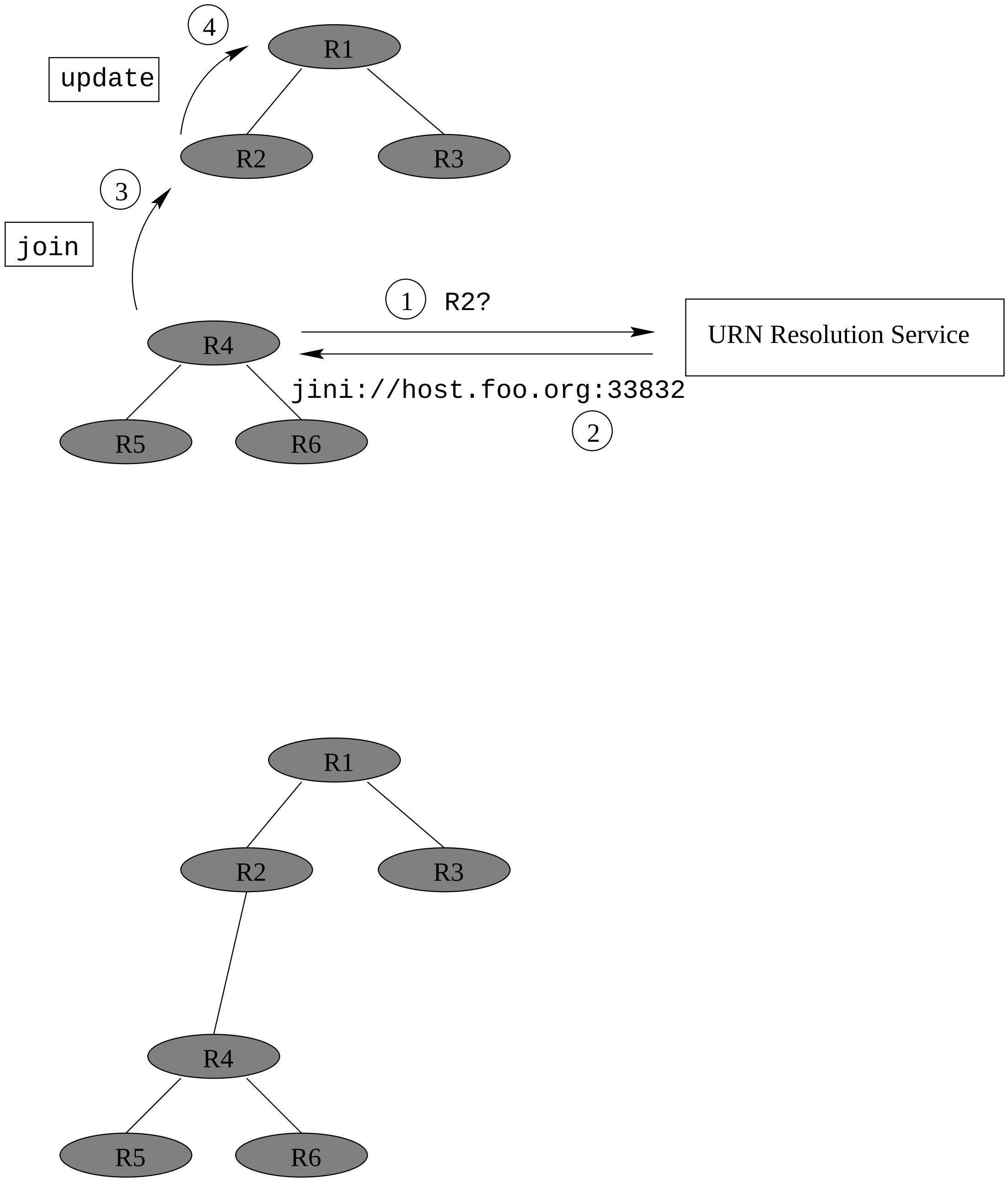


Figure 4-2: Top: Region R4 joining region R2. Bottom: Region graph after R4 joins.

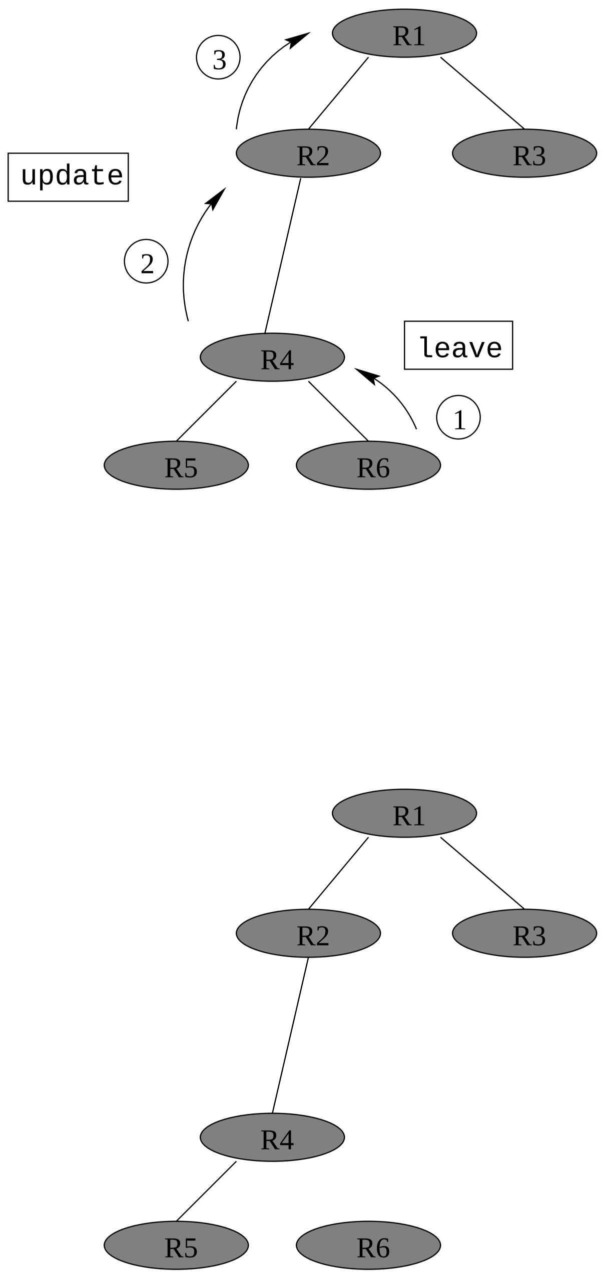


Figure 4-3: Top: Region R6 leaving region R4. Bottom: Region graph after R6 leaves.

Note that R6 has not been deleted or destroyed; it has merely asked R4 to remove it from R4's list of child regions. In fact, R6 may still be a subregion of other regions not shown here.

4.3.3 Service Join

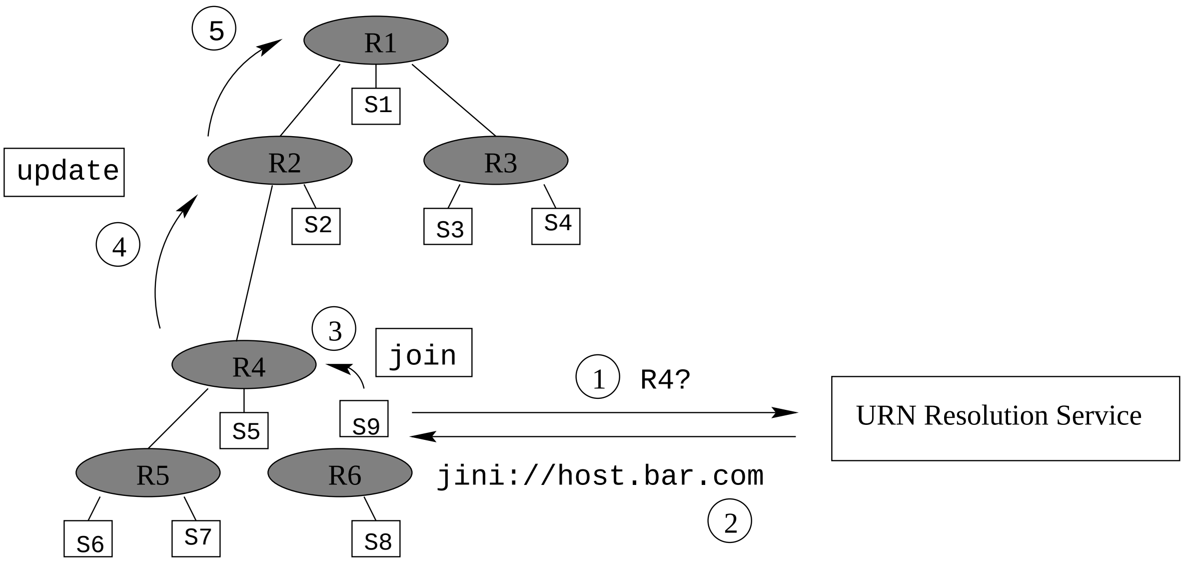
Figure 4-4 shows service S9 joining region R4. In this figure and those that follow, the services are shown.

1. S9 queries a URN resolution service to find out the location of R4's Region Manager.
2. The URN resolution service returns `jini://host.bar.com` for R4.
3. S9 sends R4 a message saying that it wants to join. The message includes S9's interfaces, attributes, and their values as obtained from the table of views. R4 adds S9 to its list of local services.
4. R4 sends an update message to its parent, R2, to notify it that a new service has joined. The message contains all of the information in the original join message, along with the name of the service's parent region (R4). R2 also updates its internal information.
5. R2 propagates the update up the hierarchy to its parent.

4.3.4 Service Change

Figure 4-5 shows service S9 changing its attributes and sending an update message about the changes to its parent, R4.

1. S9 sends an update message to its parent, R4, containing the updated set of attributes and values. R4 updates its service information accordingly.
2. R4 sends the update message to its parent, R2.
3. R2 continues to propagate the update upward.



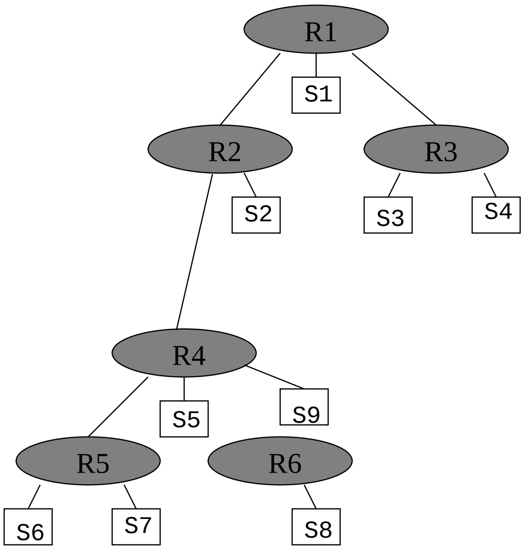


Figure 4-4: Top: Service S9 joining region R4. Bottom: Region graph after S9 joins.

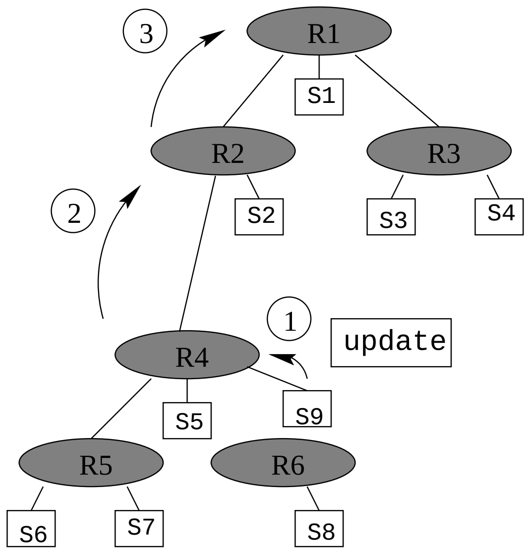


Figure 4-5: Service S9 changing its attributes and sending an update to region R4.

4.3.5 Service Leave

Figure 4-6 shows service S4 leaving its parent region, R3.

1. S4 sends R3 a message saying that it wants to leave. R3 removes S4 from its list of local services.

2. R3 sends an update message to its parent, R1, to notify it that S4 has left. R1 also updates its internal information.

4.3.6 User Agent Query

Figure 4-7 shows a user agent querying region R2 for services with a given set of characteristics.

1. The user agent queries a URN resolution service to find out the location of R2's Region Manager.

2. The URN resolution service returns `jini://host.foo.org:33832` for R2.

3. The user agent sends a query to R2's Region Manager. It requests all services contained by R2 that have interface types t_1 and t_2 and that have attributes A_1

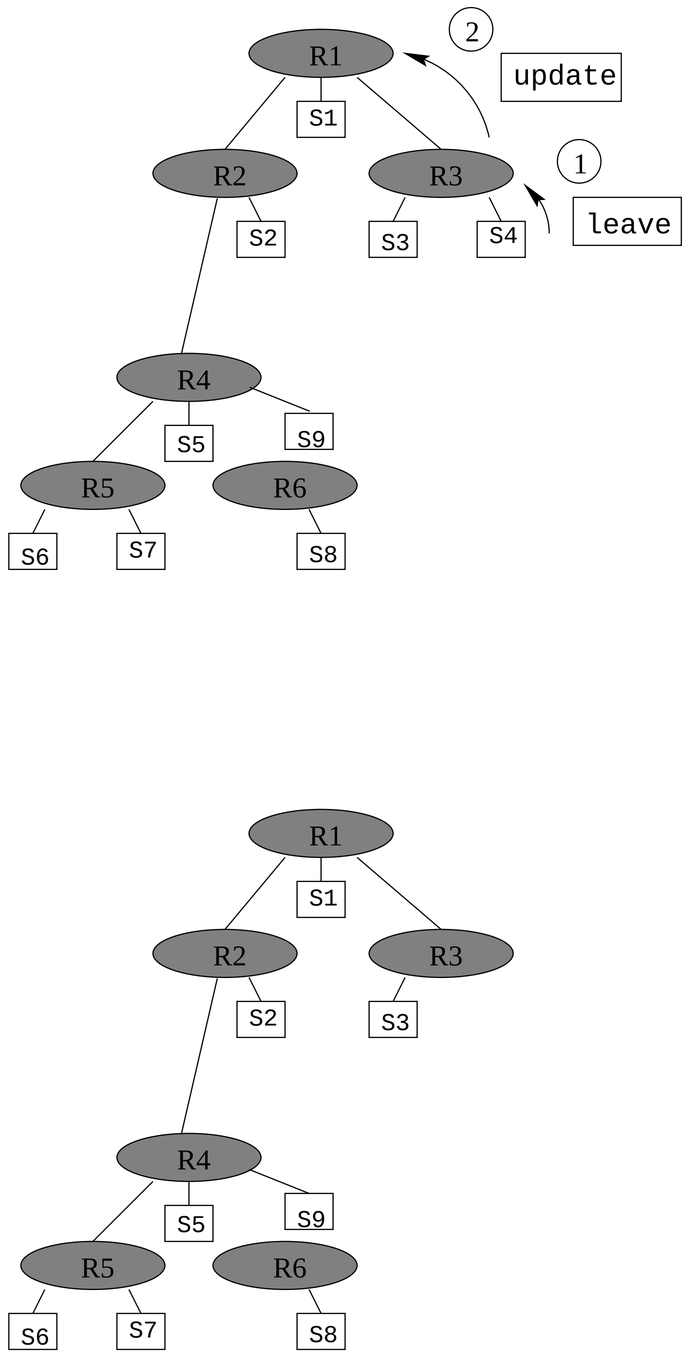


Figure 4-6: Top: Service S4 leaving region R3. Bottom: Region graph after S4 leaves.

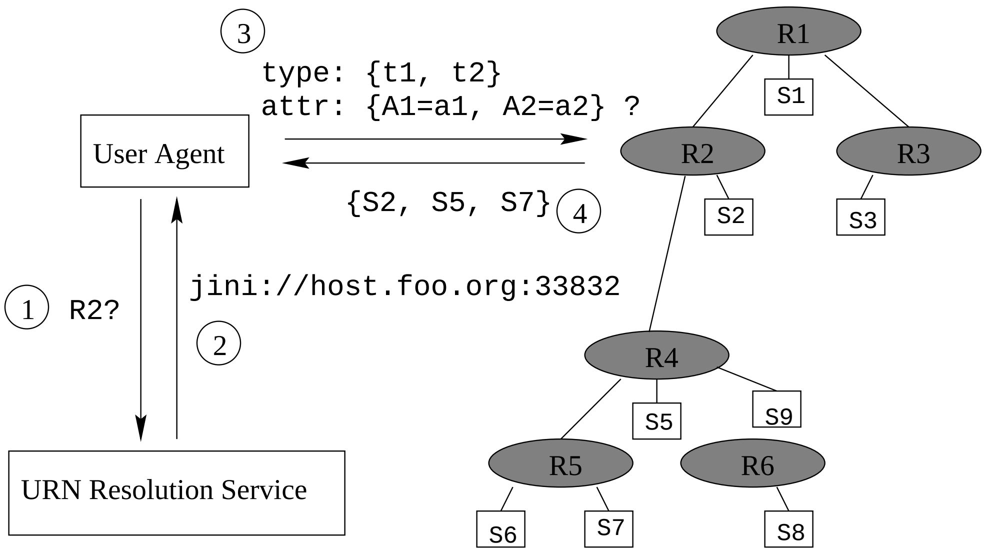


Figure 4-7: User agent querying region R2.

and A_2 with values a_1 and a_2 respectively.

4. R2's Region Manager checks its information on the services it contains—S2, S5, S6, S7, and S9. It finds that S2, S5, and S7 meet the above conditions, and includes them in the list of service entries it sends back to the user agent.

Note that R2's Region Manager examined the local services of its subregions, R4 and R5, as well as its own local services. Also note that since R2's Region Manager caches information on all of the services it contains, it did not need to query R4 or R5 in order to answer the user agent. A different caching scheme might have required R2 to query one or both of its subregions.

4.4 Summary

Services provide valuable functionality to user agents, and their usefulness can be increased substantially by including them in the region infrastructure. In order to work effectively as part of the region infrastructure, services must meet a number of requirements. One requirement is that not only must they provide functionality, but this functionality should be dependent on the region and/or region type, allowing a service to tailor itself to different regions. Services also need a way to share information and join regions easily. They need to be easy to configure so as not to hinder development and use. Finally, services must have a type model to assist user agents and to fit in with the region type model, and they must address security needs such as authentication, authorization, and type information trust issues.

The proposed architecture for meeting these requirements is a service augmented with a protocol and with additional information required by the region infrastructure. This additional information consists of an identifier for distinguishing between service instances, a table of views which capture the notion of region-dependent attributes and interfaces, and a list of the service's parent regions. The protocol is used for communicating with Region Managers in order to join and leave regions and update the service's information. Both this protocol and the additional information comprise a generic service which can be extended to create a specific service type. Lastly, some examples are given to provide a high-level view of services and regions in action.

Chapter 5

Implementation

We have built a prototype of the region infrastructure. This prototype implementation consists of four major parts: a Region Manager, a generic service framework, a URN resolution service, and a region browser for viewing and manipulating regions and their services. The prototype was developed with the Solaris Reference Implementation of Java 2, and it makes substantial use of the Jini 1.0 development kit. In the following sections, we describe the four parts of the prototype and give an example to demonstrate their use. We then describe some of the issues that arose during the implementation process, and conclude with a discussion of Jini's impact on the prototype implementation.

5.1 Region Manager

The Region Manager implementation divides its tasks into two parts. One is the management of local services. The other is the management of the region hierarchy. Management of the hierarchy is performed by the Region Manager, while management of local services is delegated to Jini.

The Region Manager uses a Jini lookup and a Jini construct called a *group* to manage a region's local services. A group is a flat collection of services, much like the "basic region" from Section 3.2.1. A Jini lookup does for a group what a Region Manager does for a region—that is, it coordinates the joining and leaving of services and also allows users to query for services based on attributes and type. Thus, groups and Jini lookups are well-suited for managing a region's local services. In the prototype system, there is a one-to-one mapping between regions and groups of the same name.

Since Jini does not have the kind of support for hierarchy that we desire for regions, the Region Manager has its own functions and information pertaining to region hierarchy. This includes knowledge about the region's parents, knowledge about its subregions and their services, functions for adding and removing subregions, functions for propagating updated information throughout the hierarchy, and a lookup function that spans across a region and its subregions.

The Region Manager and the Jini lookup interact to provide a complete picture of a region. The Jini lookup speaks the protocol and does the bookkeeping for local services that join, leave, or change. The Jini lookup notifies the Region Manager when changes occur in its group of services. The Region Manager takes care of propagating the changes. Meanwhile, the Region Manager handles other tasks, like service queries over all subregions and the addition and removal of subregions, of which the Jini lookup is unaware.

It is important to note that, despite their specialized functions, the Region Manager and the Jini lookup are considered to be services that exist as part of the region. They too have attributes, identifiers, and the other features that characterize a service. Of course, they are subject to the restriction that they must not leave the region, since they are responsible for managing it. Also, only one local Region Manager and Jini lookup are allowed in each region (not counting the ones for its subregions).

5.1.1 Interface

The Region Manager implementation provides an interface with the following functions:

- **Add/remove subregions** Add or remove a child subregion of a parent region.

- **Add/remove/modify local services** Called when the Jini lookup notifies the Region Manager about changes in the service group.

- **Update** Called by a region to inform parent regions of a change; used to propagate information up the hierarchy.

- **Get types** In this implementation, regions store their own type information and provide it upon request.

- **Get tree** Get the subtree consisting of the region and all regions which are its descendents.

- **Get local services** Get the list of services directly contained by this region (i. e., its child services).

- **Get all services** Get the list of all services contained by this region (including those in subregions).

- **Lookup** Find services with the specified ID, types, attributes, and/or attribute values.

5.1.2 Data Structures

The Region Manager implementation contains the following data structures for region management:

- **Name** A human-readable, globally-unique identifier for the region (not the same as its service ID).

- **Type information** Its interfaces, types, and supertypes, and their hierarchy.

- **Parent regions** The parents of the region.

- **Child regions** The region's child subregions.

- **Tree of subregions** The subtree consisting of the region and all regions which are its descendents. (The regions are represented as node objects; the tree represents the region structure.)

- **Jini lookup** A pointer to the Jini lookup that handles its group of local services.

- **URN resolver service** A pointer to a URN resolution service for performing any necessary URN lookups.

These data structures are examined and manipulated by the functions in the previous section.

5.1.3 Caching and Update Policy

Various choices can be made in regard to a caching and update policy. One choice is whether to use incremental updates or specify the full state in updates. Incremental updates are smaller, but full updates are idempotent and greatly reduce the complexity of error recovery. Another choice is whether to use individual or batch updates, and how often updates should occur. Individual updates require more overhead, but allow updates to be sent immediately. Frequent updates maintain a high level of cache consistency, but generate more traffic. Still another choice is the decision of which subregions to include in the cache. Caching many subregions reduces lookup time, but requires more storage space at the Region Manager. Also, the Region Manager may need to deal with missing, out-of-order, or duplicate updates, which can add to the complexity introduced by the caching and update policy. In general, the factors involved in caching and update policy tradeoffs include cache size, update message size, number of messages, and complexity.

The caching policy used in the prototype implementation is to cache all subregion information; use individual, incremental updates; send updates as soon as a change is detected; and ignore out-of-order and duplicate updates. The advantages of this scheme are that it is simple to implement, it keeps the region caches fairly consistent, and it does not require additional subregion queries when searching for services, since all service information is in the cache. Its disadvantages are that it is not very robust in case of distributed failure, and it incurs a significant communication cost by generating a large number of messages, which may additionally have overhead.

5.2 Generic Service Framework

The generic service framework in the prototype supports tasks common to all services, such as managing attributes and communicating via the registration and update protocol. It provides classes that implement these tasks and that can be extended to create a specific type of service. In this version, user agents access services using Java's Remote Method Invocation (RMI) mechanism. Future versions may have additional access methods.

Like the Region Manager, the service framework divides its functionality into several parts, and some of these parts are delegated to Jini. The Service Backend implements the service's specialized functionality. The Service Manager and Jini services are used to coordinate the joining and leaving of regions, to handle the associated protocols, and to control the exposure of interfaces and attributes.

5.2.1 Jini Services

Jini has a model for services. Jini services have types, attributes, and identifiers. They know the protocol for interacting with Jini lookup services. Jini services can even register with multiple groups. A problem, though, is that a Jini service cannot expose different interfaces and attribute sets to different groups, which is one of the requirements for services in regions. As a result, the Jini service model cannot be used directly for implementing the services in the region infrastructure.

The prototype implementation works around this by using Jini services as an interface between a region and a service in the region infrastructure. A service is represented by a different Jini service in each of the regions it joins, and each of the Jini services is configured with its own interface and attribute set. This architecture is shown in Figure 5-1. Although this arrangement may not be ideal from an efficiency

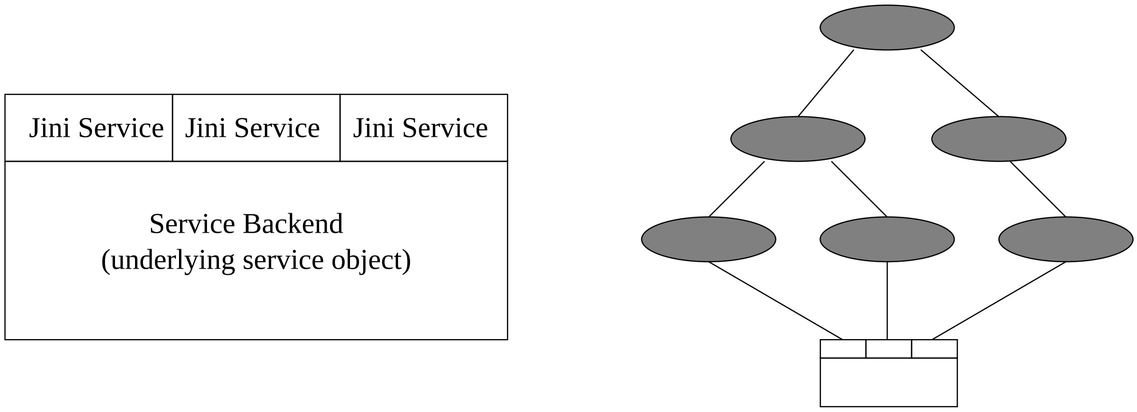


Figure 5-1: Left: The service architecture. Multiple Jini services are layered on top of an object that implements the service. Right: The Jini services can be used to expose different attributes and interfaces to different regions.

or design perspective, it enables rapid development by taking full advantage of Jini's capabilities.

5.2.2 Service Backend

The Service Backend is the component of the service that actually implements the service's specialized functions, which can be accessed remotely via RMI. The Service Backend uses the Java interface mechanism together with Jini services to expose different sets of these functions in different regions. Each Service Backend is associated with several Jini service classes. Each of these classes has an interface which provides access to a subset of the Service Backend's functions. Whenever the service joins a region, a Jini service class with the appropriate interface is instantiated, and it serves as the Service Backend's interface to the region. In this implementation, one Jini service is instantiated per joined region, rather than sharing Jini services between regions. This allows the service administrator to change the Jini service interface to one region without affecting other regions. Presumably, alternative implementations could use shared Jini services for efficiency, but they must have a way to address the situation in which regions that had a common Jini service interface subsequently require different ones.

Developers create services by extending the Service Backend class and adding their specialized functions. They must also write the Java interface files containing the various service interfaces they wish to make available. The prototype includes a compiler that will generate the appropriate Jini service classes for those interfaces.

One difficulty with this approach is that the selection of service interfaces must be specified at compile-time. It also does not allow the service provider to add or remove individual functions from an interface at runtime. These are constraints imposed by Java and its interface mechanism. However, developing a new mechanism for functional interfaces for services would require substantially more effort than using Java's interface mechanism and accepting its limitations, so we have chosen this approach for now.

5.2.3 Service Manager

The Service Manager handles the protocol for joining and leaving a region's Jini group. It maintains the list of regions to which a service belongs. It is responsible for joining and leaving all of those regions at startup and shutdown. When the service administrator makes changes to the region list, the Service Manager makes the necessary registrations and deregistrations. Likewise, when the service administrator changes the service attributes or interface, the Service Manager responds by updating the registrations in the affected regions.

The Service Manager's most important job is its coordination of Jini services as interfaces which provide region-dependent attributes. When a service joins a region, its Service Manager first looks up the appropriate view for the region, in the manner described by Section 4.2.2. The view contains an interface and an attribute set. The Service Manager spawns a new Jini service which implements the interface in the view. Then the Service Manager configures the Jini service with the attributes in the view. Finally, the Service Manager has the Jini service join the region's Jini group, and the Jini service subsequently serves as the interface between the region and the Service Backend. Thus, the Service Manager manages multiple Jini services in different regions for its single underlying service object.

Interface

The Service Manager implementation provides an interface with the following functions:

- **Join/leave one or more regions** Join or leave a set of one or more parent regions.
- **Add/remove/modify attributes** Add, remove, or modify the set of attributes seen by a particular parent region.
- **Re-register** Remove an old Jini service from a region and replace it with a new service. Necessary if the service changes the interface to expose in that region.
- **Get service ID** Get the identifier of the underlying service object (not the Jini service ID).
- **Get table of views** Get the table of views used by the service.
- **Get region list** Get the list of parent regions which the service has joined.
- **Save configuration** Save the service configuration information, which includes the view table, region list, and ID.

Data Structures

The Service Manager implementation contains the following data structures for service management:

- **Service Backend** A pointer to the Service Backend that implements all of the service methods.
- **List of Jini service interfaces** The Jini services that are acting as interfaces between the Service Backend and the various regions the service has joined.
- **Service ID** Identifier for the Service Backend.
- **List of regions to join** Regions to join upon service startup.

Figure 5-2: Region menu.

- **Table of views** Table that maps regions and region types to the view (attribute set and interfaces) exposed by the service in each region or region type.

5.2.4 GUI Menu

The generic service framework does not provide a full GUI because each service has different user interface requirements. However, it does provide a menu GUI component which can be incorporated into a service's menu bar (Figure 5-2). This menu contains options for viewing and editing configuration information. These options are useful for any service that is part of the region infrastructure.

5.3 URN Resolution Service

The URN Resolution Service is a simple database which maps region names such as `MyKitchen` to Jini URLs such as `jini://kitchen.home.net` that can be used to find the Region Manager (Figure 5-3). The database is populated by manual addition of entries and also by listening to multicast announcements made by a region's Jini lookup. The URN Resolution Service is an example of how the generic service framework can be extended to create a specialized service.

It should be noted that the URN Resolution Service provided with this implementation is in reality a placeholder for a more complicated URN resolution service. URNs and URN resolution pose a number of complex and interesting issues, but they are not the main thrust of this work. For a further discussion of URNs and URN

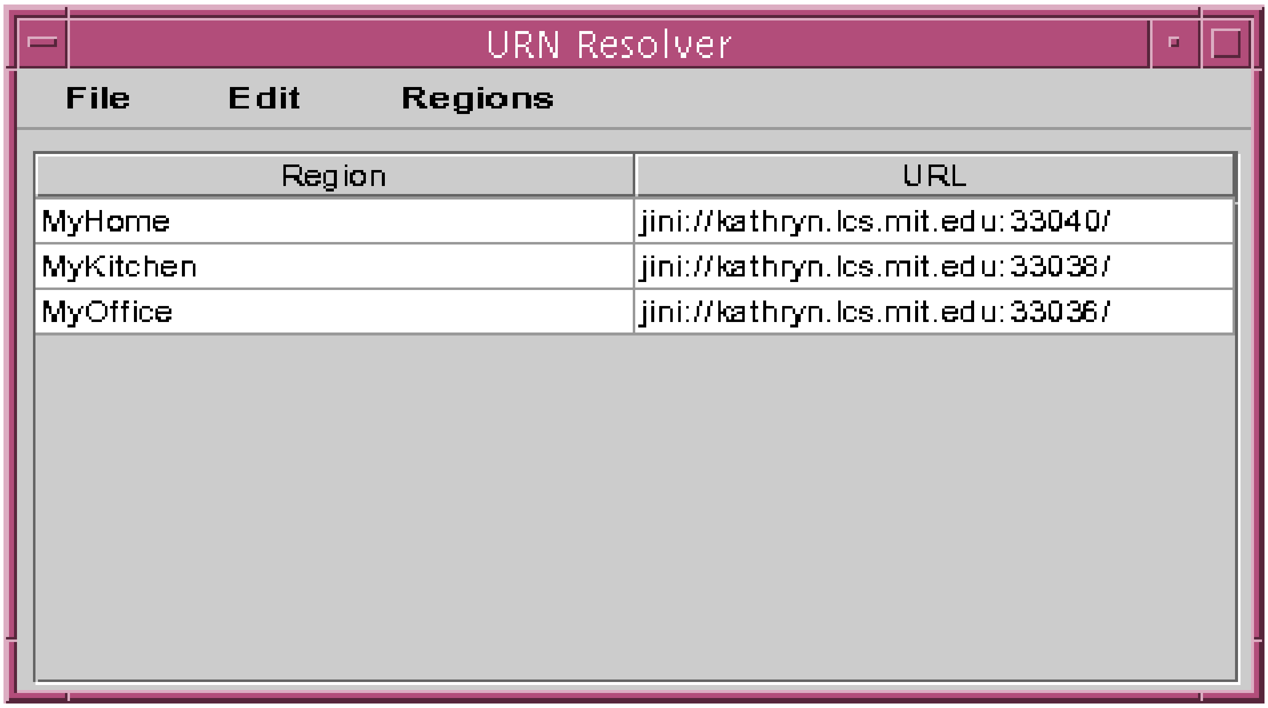


Figure 5-3: URN Resolution Service.

resolution, see [14, 13].

5.4 Region Browser

The Region Browser is a graphical tool for viewing and constructing regions and browsing their services (Figure 5-4). The browser is set to point at a top-level region, whose name is displayed at the top. The upper panel of the browser shows the hierarchy of subregions. The lower panel shows the region services. The user can elect to view either the local services only, or all services contained in the region. The `File` menu contains options for setting the top-level region and for viewing the names of the region's parents, the region's types and supertypes, its service ID, and its Jini URL. The `Edit` menu contains options to add and remove subregions and to find services using attribute/type lookup.

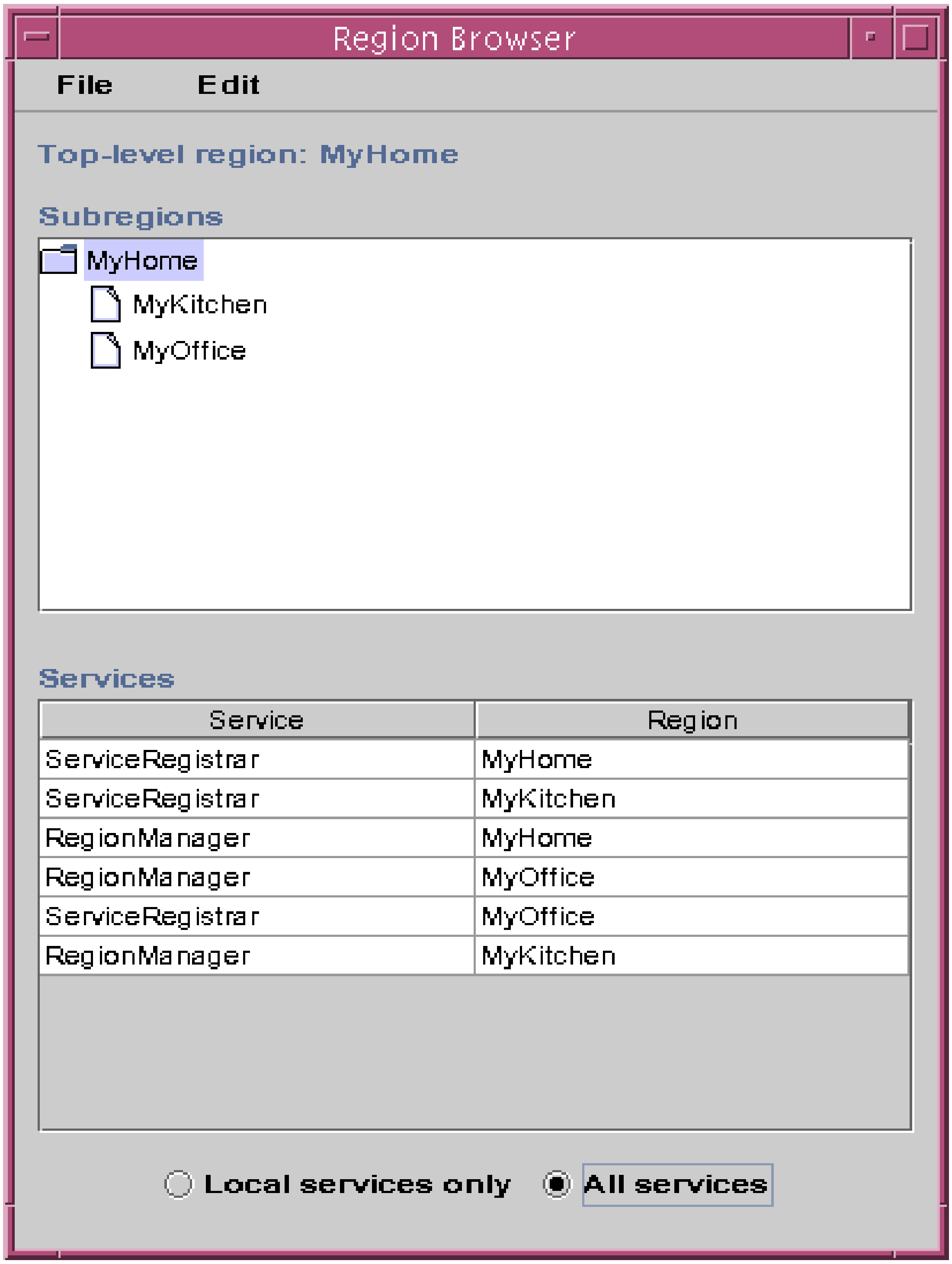


Figure 5-4: Region Browser.

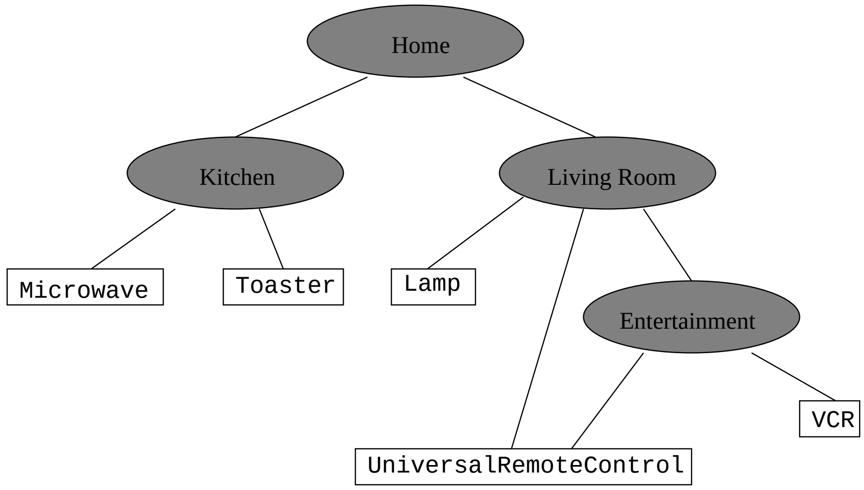


Figure 5-5: `Home` region.

5.5 Example

The following example illustrates how the concepts outlined in Chapters 3 and 4 are embodied in the prototype implementation.

In Figure 5-6, the Region Browser displays the `Home` region shown in Figure 5-5. The **Subregions** panel of the browser shows the region hierarchy. The hierarchy can be constructed by using the Region Browser's **Edit** menu or by directly calling the appropriate Region Manager methods. This causes joining and/or leaving messages to be sent to the regions involved, and it also triggers updates that propagate up the hierarchy.

The **Services** panel of Figure 5-6 shows all of the services contained by the `Home` region, along with each service's parent region. Note that each region contains the local services `ServiceRegistrar` and `RegionManager`. These correspond to the Jini lookup and the Region Manager, which are responsible for managing the region.

Figure 5-7 is the service information for the `Toaster` service. It shows the functional interface (Java interface `net.regions.example.home.toaster.Toaster`) and the attribute set and values that the `Toaster` service exposes in its parent region, `Kitchen`. The underlying service object is of Java type `net.regions.example.home.`

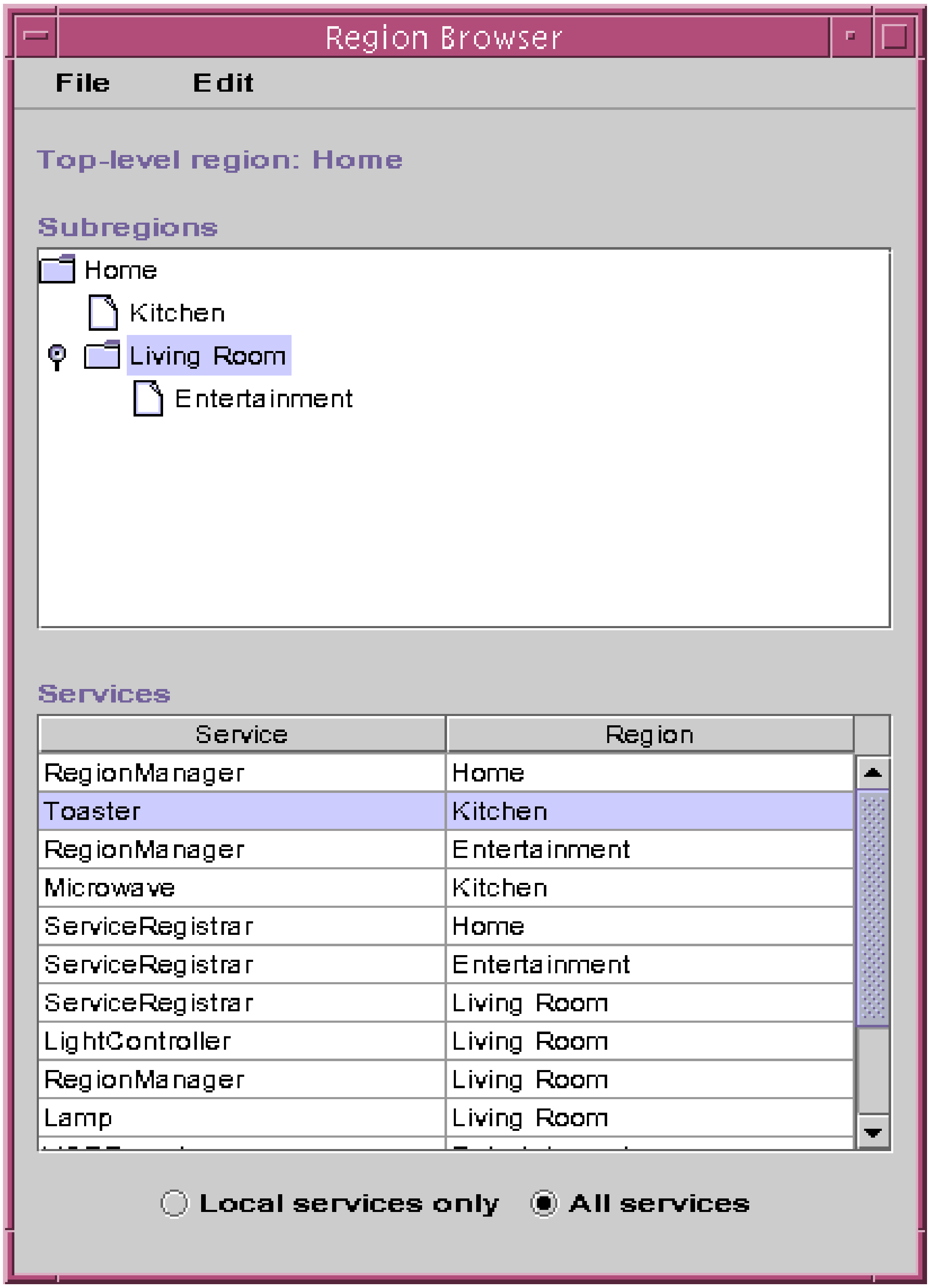


Figure 5-6: Region Browser displaying `Home` region.

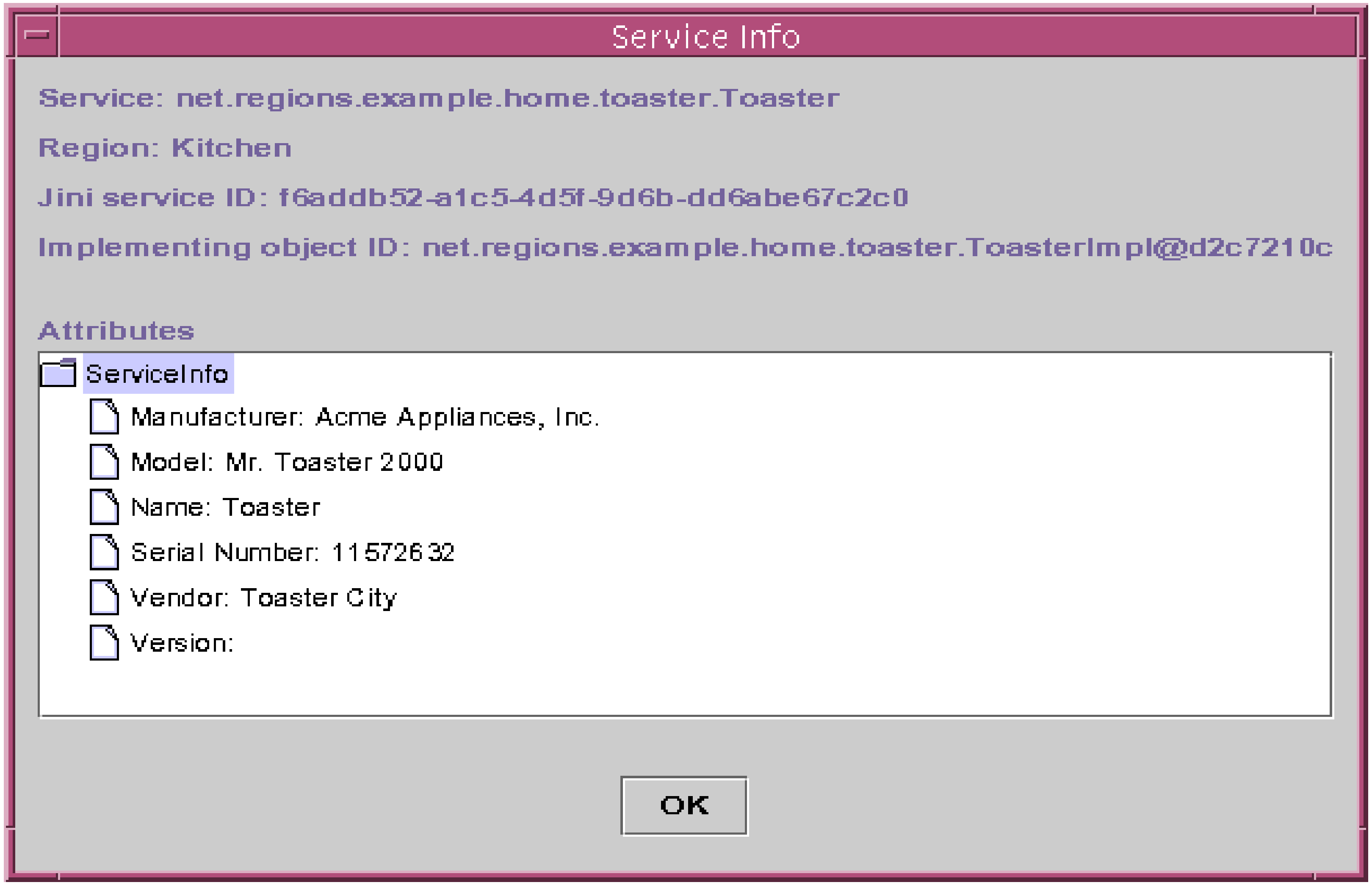


Figure 5-7: Service information for `Toaster`.

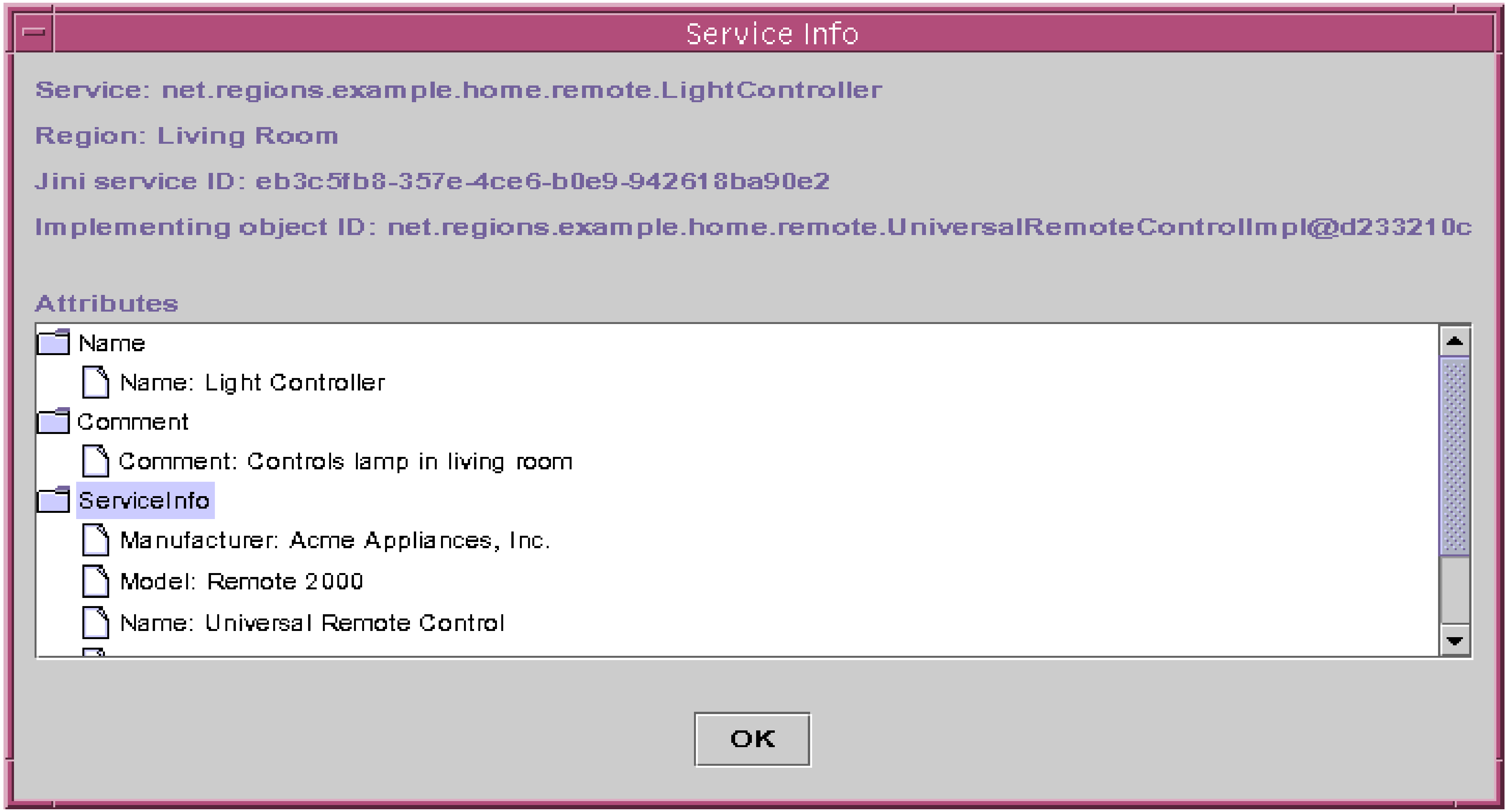


Figure 5-8: Service information for `LightController`.

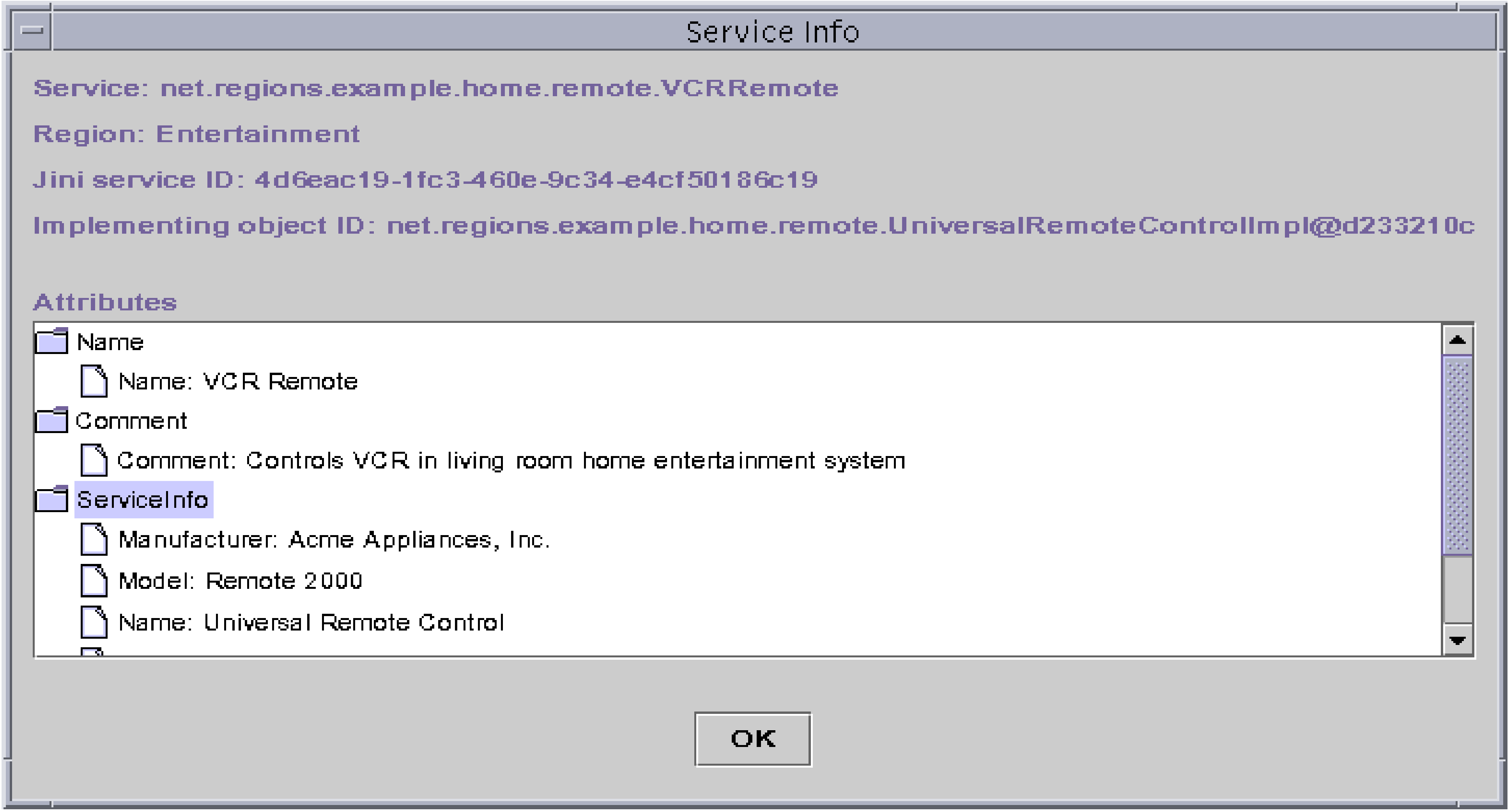


Figure 5-9: Service information for `VCRRemote`.

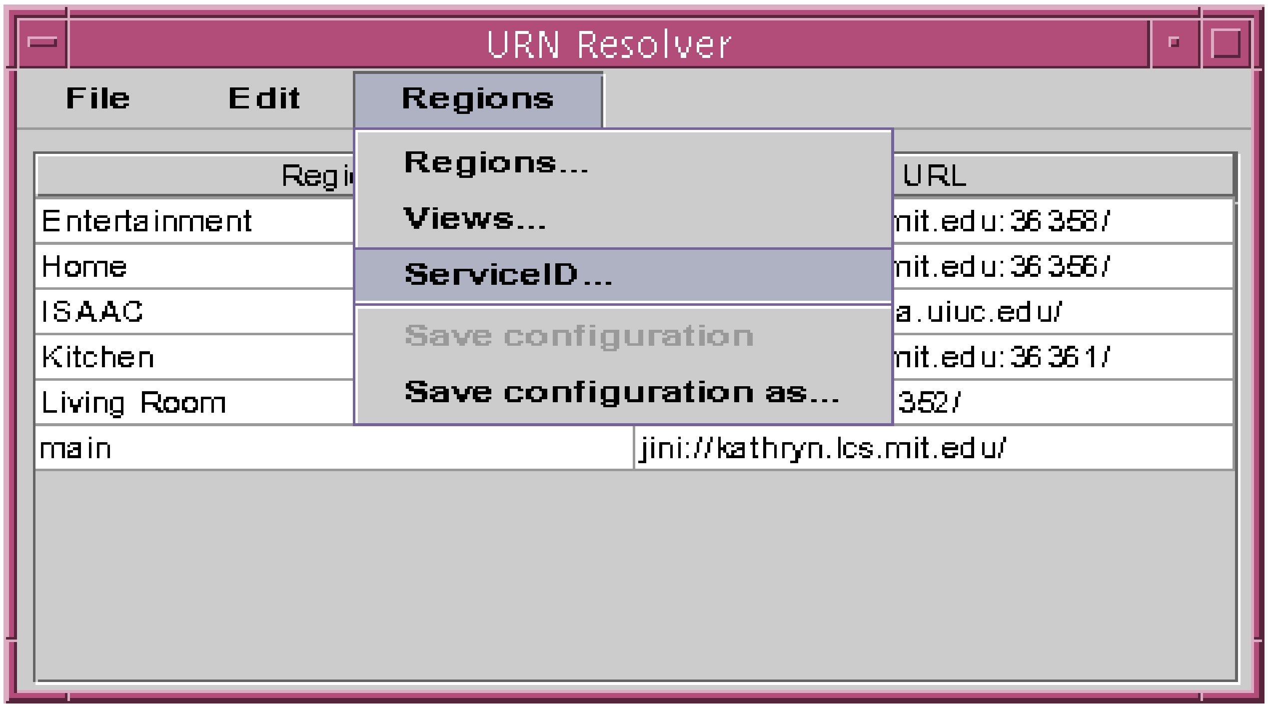


Figure 5-10: URN Resolution Service.

`toaster.ToasterImpl`, as indicated by the **Implementing object ID** field.

Figures 5-8 and 5-9 show an underlying service object which is exposing different views in different regions. In the `Living Room` region, it acts as a remote control for the `Lamp`. In the `Entertainment` region, it controls the `VCR`. Also, some of the attributes differ between the two regions. However, both `LightController` and `VCRRemote` are implemented by the same underlying service object of Java type `net.regions.example.home.remote.UniversalRemoteControlImpl`, as evidenced by the matching implementing object IDs.

The URN Resolution Service is shown in Figure 5-10. It contains the mappings from region names to Jini URLs. It also highlights some of the features of the generic service framework, which are made available through the **Regions** menu. This menu contains options for displaying the ID of the underlying service object, as well as options for editing the view table and the list of parent regions. When the view table and parent region settings are changed, the updated information is sent to the affected Region Managers, which propagate the information as necessary. In addition,

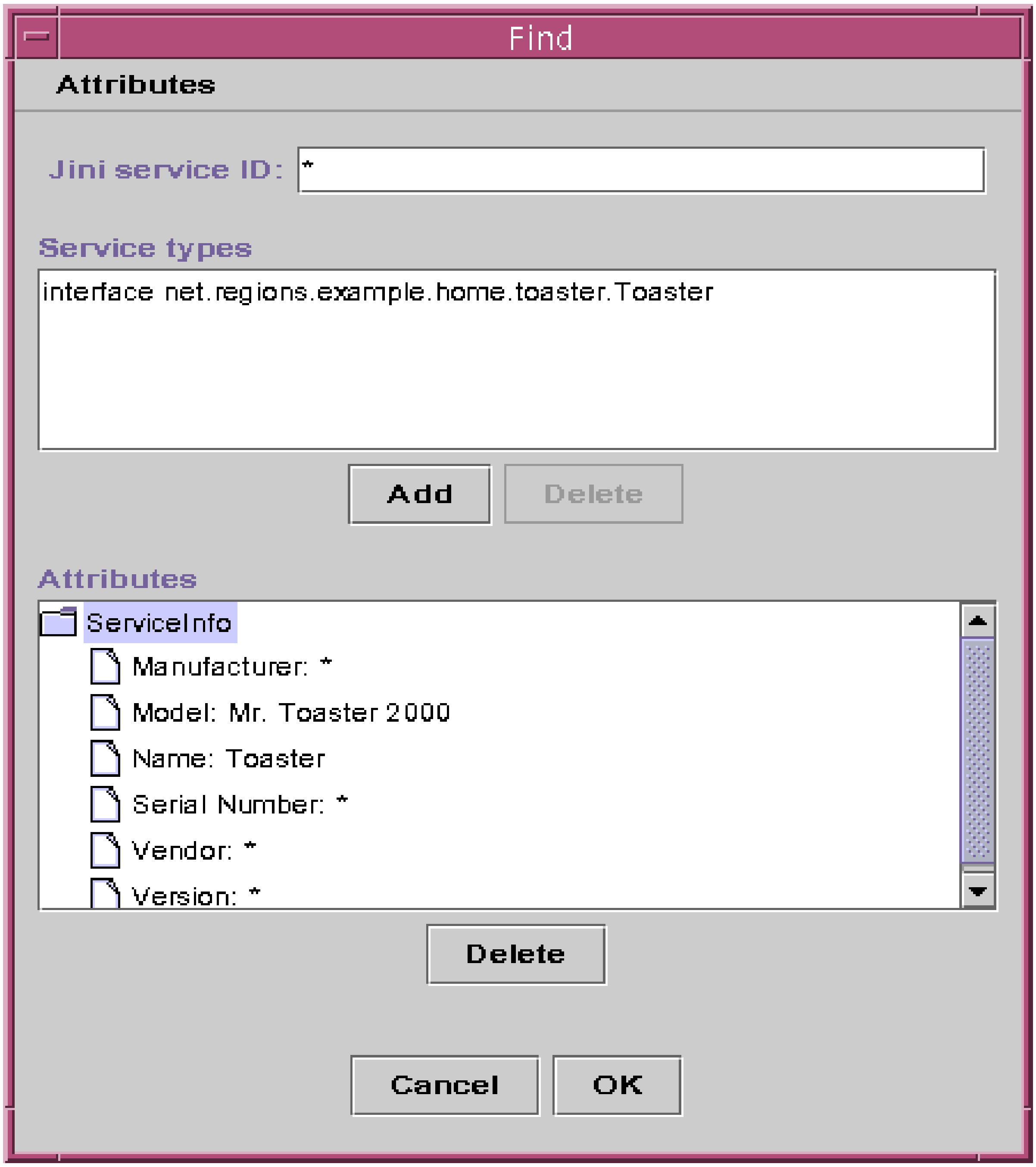


Figure 5-11: Search template for service lookup.

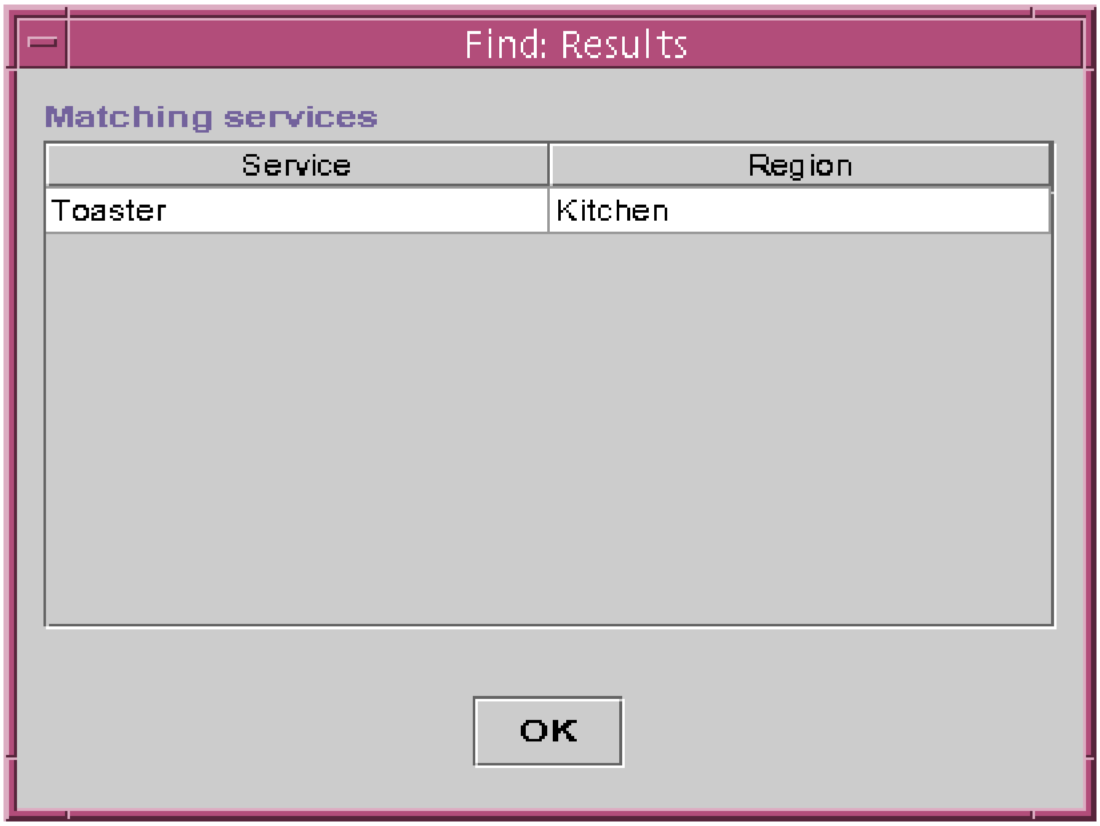


Figure 5-12: Service lookup result.

the settings can be saved so that the configuration automatically takes effect upon subsequent service startups.

Figures 5-11 and 5-12 demonstrate the Region Browser's service lookup capability. In the search template window, the user specifies the service ID, functional interface, attributes, and values desired in the service. The user can also modify the scope of the search by changing the Region Browser top-level region from `Home` to `Kitchen` or `Entertainment`, for example. In this case, the service and several of the attribute values are wildcards, and the top-level region is `Home`. Only one service matches the template—the `Toaster` service in the `Kitchen` subregion.

5.6 Some Comments on Implementation

One caveat to offer about the region infrastructure implementation described here is that it is a prototype, not a complete implementation. It contains the basic functionality for the region infrastructure. However, it is not as comprehensive as the system described in Chapter 3. It does not provide several caching and update policies; currently it only supports the policy in Section 5.1.3. It does not have multiple query formats, such as value-range or predicate-based queries. It does not contain examples of regions that hide information from their parents, or services that inherit attributes from their parent regions. It also contains no examples of typed regions that perform the kind of type-checking described in Section 3.2.6, although developers can create such regions by extending the Region Manager classes and adding any necessary type-checking restrictions and code.

Another comment to add is that this prototype is a demonstration and proof-of-concept. It is *not* meant to be production-quality software. Numerous improvements would be required to make it production-quality. Efficiency would have to be scrutinized, and some of the algorithms would have to be replaced with faster ones. The implementation would have to be more graceful and robust in its handling of errors, both distributed and local. The GUI would benefit from increased usability and perhaps a redesign. Finally, this system cannot be deployed in a real-life setting without more work on its security requirements and the mechanisms for meeting them.

5.7 The Role of Jini

From our description, it is clear that Jini was an intrinsic part of our prototype region infrastructure. Given its role, it is important for us to examine Jini's impact on our implementation.

What did we gain by using Jini? One huge benefit was that Jini already adheres to the three-part model we wished to adopt. It has a model for services, complete with types and attributes. The Jini lookups already keep track of services joining and leaving Jini groups, and Jini provides the Discovery and Join specification for communication between services and lookups. The Jini technology extends to tools for distributed computing, such as leases and distributed events and transactions, which are very convenient for developing the region infrastructure. Also, since Jini uses Java, our prototype can take advantage of Java's type model, interface mechanism, and ease of use. Basically, Jini provides the tools and infrastructure for a local solution to the scoped service location problem, and these can be used and adapted to our needs, resulting in a much faster implementation time than if we had started from scratch.

Of course, Jini is not without its drawbacks. The very fact that it is a local solution becomes a limitation in the domain of regions, which are meant to be large and scalable. This limitation is apparent in Jini's poor support for hierarchies of groups. It also shows up in the design of some of the protocols, which use multicast in a way that may not scale up to regions. We have had to introduce various workarounds in our design to address these difficulties. Another problematic aspect of Jini is its performance. Much of the development and testing occurred on one machine, which became unacceptably slow when multiple services and Region Managers were run on it. Performance was restored to a tolerable level by running the Region Managers and services as multiple threads under one Java Virtual Machine, rather than as separate processes, each with their own virtual machine. Presumably, performance problems would also be ameliorated by running the services and Region Managers on several different machines, which is a more realistic scenario. Another mitigating factor is that the region infrastructure's role of service discovery is only one part of the interaction between user agents and services. Depending on usage patterns of regions and services, these performance problems may not be as significant in the larger context of user agent–service interaction. However, Java is notorious for its performance issues. These may have to be addressed, especially in view of the major role played by lightweight, computationally-impoverished devices in the region infrastructure. The ultimate answer may involve developing part or all of the region infrastructure in a different language.

Our basic approach to integrating Jini was to leverage its strengths by utilizing it heavily for administration of local services, and augment it by adding hierarchy support and related functions like caching. We used workarounds when the models provided by Jini did not entirely suit our needs, as in the case of Jini services, which could not be used directly. We also treated Jini as a black box to be used through its functional interface, rather than by modifying its source code; this helped speed development.

Our strategy for using Jini proved highly effective for rapid prototyping of the region infrastructure. However, in retrospect the design and implementation of the region infrastructure might be cleaner and more efficient if Jini were directly modified to adapt it to regions. This would require considerably more development effort, and the end product would no longer conform to the Jini specification. In the long term, perhaps the best direction for the region effort is to persuade the Jini design community to go beyond local service location and adopt the more general goals of regions in their specification of Jini.

5.8 Summary

Our prototype implementation of the region infrastructure includes a Region Manager, an extensible generic service framework, a URN resolution service, and the

Region Browser. The Region Manager is organized into two parts: a Jini lookup, which manages local services, and a Region Manager service, which manages region structure. The generic service framework uses Jini services layered over an underlying Service Backend that implements the specialized functionality. A Service Manager coordinates the joining and leaving of services.

It is important to note that our prototype is not as expansive in its features as the region architecture described earlier, nor is it meant as production-quality software. It was intended, and has served, as an exploration of feasibility, an opportunity to thresh out problems, and a proof-of-concept.

It is also important to examine the role of Jini in our implementation. Jini is extremely effective for managing local services and service location, but falls short in the areas of scalability, hierarchy support, and performance. We exploit Jini's benefits and make up for its shortcomings by relying on Jini for local service administration, and augmenting it with additional components and workarounds as necessary. We also treat the Jini technology as a black box. This led to faster implementation, but the long-term solution may involve changing Jini's design goals and specification directly.

As of this writing, the source code for version 1.0 of the region infrastructure is available from [1]. The Jini development kit and related information may be found at [15].

Chapter 6

Performance Issues

Predicting the performance of the region infrastructure can be problematic. Its performance is dependent on a host of factors. Some of these factors differ from region to region, meaning that the same region infrastructure may be applied to different regions and yield different performance for those regions. Other variables that affect performance are set by the administrator of a region, who can select values to optimize performance for that particular region. Performance is affected by some of the implementation and representation choices made within the region infrastructure. The performance of a region is also affected by the other regions with which it interoperates; these other regions may use different implementation choices and system parameter values.

Because of the difficulties in predicting, analyzing, and measuring region infrastructure performance, we will limit ourselves to the following. First, we will discuss the factors that affect performance. We also list various metrics that can be used for performance quantification and comparison. Finally, we analyze the costs incurred by our own implementation of the region infrastructure, probe the factors that contribute to these costs, and discuss ways to improve the performance of this implementation. This discussion is intended to shed some light on how various factors affect region performance, and to serve as a general guide for performance evaluation.

6.1 Factors Affecting Performance

Most of the factors affecting region performance fall into four broad categories: region characteristics, network layout of the region, caching and update policy, and other policy goals.

6.1.1 Region Characteristics

The performance of the region infrastructure can be heavily influenced by the characteristics of a particular region. The topology of the region is a case in point. It determines the amount of traffic generated for updates, the time for information to pass from one region to another, and the load on various subregions. It may be difficult to characterize region topology, since regions were designed to allow arbitrary topologies. However, some topologies can be characterized in a way that may facilitate analysis of the relationship between topology and performance. For example, hierarchical regions can be characterized in terms of branching factor, height, and size.

Another characteristic which differs from region to region and which also affects performance is the frequency of changes in a region. This corresponds to how often subregions and services are added, removed, or changed. In a home network region, this will happen rarely, as the set of devices and attributes in such a region is fairly static. At a trade conference, however, changes could be very frequent, with numerous services joining and leaving regions continually. Of course, the frequency of changes will have a bearing on the amount of update traffic in a region, which in turn affects performance.

6.1.2 Network Layout

On a practical level, network layout of the various entities in a region can have an impact on performance. Region topology may not correspond exactly to network topology, and this may have an impact on how region load and traffic affect the network and the computers connected to it. Although we will not discuss this in further detail, it is prudent to keep in mind the influence that network layout can have on region performance.

6.1.3 Caching and Update Policy

The caching and update policy has a crucial impact on region infrastructure performance for several reasons. One reason is that it interacts with the region characteristics mentioned previously. Another reason is that updates account for a significant portion of communication costs in a region, and caching can help reduce these costs. Finally, the caching and update policy is one of the factors affecting performance over which the region administrator has complete control; the administrator's choices play an important role in performance tuning.

Section 3.2.5 listed the various choices that can be made when specifying a caching and update policy: the subregions to cache, the type and frequency of updates to use, the timing parameters for updates, and the high-level policies to implement. Intuitively, different choices seem better for different regions. For example, individual updates work well in a fairly static region, and they are easy to manage. However, the overhead might prove prohibitive for a very dynamic region, and batch updates might be a better solution in that case. Although the relative merits of specific caching and update policies are outside the scope of this discussion, the policy does have an impact on performance, and it should be chosen to optimize performance in the region in which it will be used.

6.1.4 Other Policy Goals

Another factor affecting the outlook on performance is the policy goals that are set by administrators. Region administrators want their regions to "perform well," but how is good performance defined? One possible performance goal is to limit the amount of bandwidth used by the region. Another is to limit the rate at which updates are received by any region in the system. Other possibilities are to have an upper bound on the time for updates to propagate, or on the time for looking up services. There are still more possibilities. Region administrators may have targets and tolerances for each of these quantities. When evaluating performance, one must also take into account how well the region infrastructure meets these goals, which are chosen by the administrator and may be different for different regions.

6.2 Metrics

What are the various quantities that can be measured and calculated when evaluating the performance of the region infrastructure? Some were mentioned in the previous section: the amount of bandwidth used, the rate at which updates are received by regions, the time for updates to propagate, and the time for looking up services in a region. Other quantities for measuring performance are the storage space required for each region; the running time for region operations such as adding, removing, and modifying subregions and services; and the number and size of messages sent during these operations. We will examine some of these quantities in the analysis to follow. It is important to remember that although these metrics can be calculated or measured quantitatively, performance comparisons are ultimately subjective because of the different performance goals one may have, as discussed in the previous section.

6.3 Analyzing the Prototype Implementation

We have analyzed some of the costs in our prototype implementation of the region infrastructure. Our approach was to determine the costs of each basic region operation (adding/removing child regions and adding/removing/modifying/looking up services) by examination of the algorithms and source code. These costs are expressed in terms of running time, space required, messages sent, and message size. The details of the analysis are given in Appendix A. The results are summarized in Table A.18, which is reproduced here as Table 6.1.

Table 6.1: Summary of costs for basic region operations.

Costs	Time	Space	# of Msgs	Message Size
Adding a region				
Parent region	$\Theta(S + N)$	$O(S + A + N)$	6	O(1)
Child region	O(1)	O(1)	2	$\Theta(S + A)$, Θ(N)
Update cost per ancestor	$\Theta(S + N)$	$O(S + A + N)$	1	$\Theta(S + A)$, Θ(N)
Removing a region				
Parent region	$\Theta(S + N)$	-	1	O(1)
Child region	O(1)	-	1	$\Theta(S + A)$
Update cost per ancestor	$\Theta(S + N)$	-	1	$\Theta(S + A)$
Adding a service				
Region	O(1)	$O(A)$	-	-
Service	-	-	1	$\Theta(A)$
Jini lookup	Θ(E + C + A)	$\Theta(A)$	1	$\Theta(A)$
Update cost per ancestor	O(1)	$O(A)$	1	$\Theta(A)$
Removing a service				
Region	O(1)	-	-	-
Service	-	-	1	O(1)
Jini lookup	Θ(C + A)	-	1	O(1)
Update cost per ancestor	O(1)	-	1	O(1)
Modifying a service				
Region	O(1)	$O(A)$	-	-
Service	-	-	1	$\Theta(A)$
Jini lookup	Θ(E + C + A)	$O(A)$	1	$\Theta(A)$
Update cost per ancestor	O(1)	$O(A)$	1	$\Theta(A)$
Looking up services				
User agent	-	-	1	$\Theta(A_Q + C_Q)$
Region	$O(A_Q A + C_Q C)$	-	1	$\Theta(S_M + A_M)$

As an example of how the costs of operations were analyzed, Tables A.1 and A.2 from Appendix A are reproduced here as Tables 6.2 and 6.3, respectively. They show the costs for the operation of adding a region. (R_1 is the parent region and R_2 is the child region.) Table 6.2 outlines the steps that take place during the operation. Table 6.3 shows the cost per region ancestor for propagating update messages up the region hierarchy. In both of these tables, the steps shown in boldface contribute to the cost of the operation, and the cost is shown to the right. Appendix A contains similar analyses for the remainder of the region operations.

Table 6.2: Operations and costs for R_1 to add R_2 as a child region.

Operations	Time	Space	Message Size
R_1 finds R_2's Region Manager			
R_1 resolves R_2's name			
R_1 sends message to URN resolution service	-	-	O(1)
URN resolution service looks up URL for R_2	O(1)	-	-
URN resolution service sends message to R_1	-	-	O(1)
R_1 queries R_2's Jini lookup for its Region Manager service			
R_1 sends message to Jini lookup	-	-	O(1)
Jini lookup finds Region Manager service	O(1)	-	-
Jini lookup sends message to R_1	-	-	O(1)
R_1 adds R_2 to its set of child regions	O(1)	O(1)	-
R_1 queries R_2 for all of its services			
R_1 sends a message to R_2	-	-	O(1)
R_2 sends a message to R_1	-	-	$\Theta(S + A)$
R_1 adds R_2's services to its set of services	$\Theta(S)$	$O(S + A)$	-
R_1 queries R_2 for its subtree			
R_1 sends a message to R_2	-	-	O(1)
R_2 sends a message to R_1	-	-	Θ(N)
R_1 inserts an edge in its graph between itself and R_2	O(1)	-	-
R_1 adds R_2's subtree nodes to its set of nodes	Θ(N)	O(N)	-
R_1 sends update messages to its parents			
R_2 adds R_1 to its set of parent regions	O(1)	O(1)	-

Table 6.3: Update cost per ancestor of R_1.

Operations	Time	Space	Message Size
Receive update message	-	-	$\Theta(S + A)$
Add child region's services to set of services	$\Theta(S)$	$O(S + A)$	-
Look up nodes for parent and child regions	O(1)	-	-
Add child region's subtree nodes to set of nodes	Θ(N)	O(N)	-

The variables in the first two rows of Table 6.1 are as follows: S is the number of services contained by the child region (including non-local services), A is the total number of attributes for all S services, and N is the number of nodes in the subtree consisting of the child region and its descendents. The variables in the next three rows are as follows: A is the number of attributes that the service has, C is the number of types that the service instantiates (i. e., how many classes and interfaces it has, including all supertypes), L is the parent region's Jini lookup, and E is the number of services already registered in L. The variables in the last row are as follows: A is the total number of attributes in all of the region's services; C is the total number of types instantiated by all of the region's services; A_Q and C_Q are the number of attributes and types, respectively, in the user agent's query; S_M is the number of matching services in the region; and A_M is the total number of attributes in all S_M matching services.

We can make several observations about this implementation. First of all, the most costly operation is adding a region. It is significantly more expensive than the other operations in most measures of cost shown, particularly in communication costs. This is partly due to the Region Manager's policy of caching the entire state of a region and its subregions. It is also due to the decision to use a Region Manager service in conjunction with a Jini lookup to implement the Region Manager functionality. As a consequence of this decision, the two services must communicate with each other frequently. Another observation is that updates are potentially expensive because they require one message per region ancestor, which could add up to a significant number of messages. This is a result of the policy of using individual updates which are sent immediately upon detection of a change. Lastly, the lookup operation incurs no communication costs beyond the query and reply messages. In particular, it incurs no costs from messages to subregions. This is also because of the caching and update policy.

This implementation might be best suited for regions with a low to moderate frequency of change, especially those that do not add regions frequently. The reason for this is that it generates considerable traffic for each change. This implementation may also work well for regions in which it is important for information changes to propagate quickly, as long as one is willing to bear the communication costs. The implementation also provides fast lookups, but again the caveat about update communication costs applies. However, the implementation should probably not be used for regions in which services or subregions frequently have multiple parents because the high "inverse branching factor" could drive up the number of ancestors and hence the number of update messages generated.

In the following sections, we use the cost analysis of the prototype implementation as a jumping-off point for making observations about improving efficiency, design tradeoffs, and hidden costs. Many of the issues raised here are applicable when evaluating performance for any region or implementation.

6.3.1 Improving Efficiency

We now turn to the question of how to improve the efficiency of this implementation. The first and most important observation to make is that communication costs are most likely the bottleneck. Commercial databases are available that can handle the thousands or even millions of entries necessary for a large number of subregions and services. As a result, running time and storage space are probably less significant than communication costs, unless the Region Manager is running on a device with limited computational ability or storage. A second observation is that special attention should be given to improving the efficiency of the operation for adding regions, since it is the most expensive.

In light of these two points, several changes can be made to improve performance. First of all, responses to URN resolution service queries should be cached. This will reduce the number of messages sent during the region-adding operation by two. Consolidating the Region Manager and the Jini lookup into one service would also help, since messages sent between the two services account for a significant part of the traffic when adding a region. Furthermore, integrating the Jini lookup instead of treating it as a black box may reduce the degree to which performance depends on the Jini implementation. The reason for this is that the Jini internals could then be modified. Lastly, the use of batch updates as opposed to individual updates would reduce the communication cost per ancestor for all of the operations, although it might affect the rate at which updated information propagates through a region.

This implementation already has a few features which bode well for efficiency. In most cases, incremental updates are used, instead of full updates which contain the complete state. This reduces the size of the update messages, although it makes updates non-idempotent, which introduces other issues. Also, the policy of caching complete state and issuing updates immediately hides the latency of subregion queries from user agents that are looking up services. If a region must query one or more of its subregions while processing a user agent's query, the additional communication costs are added to the latency perceived by the user agent. For this reason, cache misses are relatively expensive to the user agent. But since the full state is cached and updates happen "offline" (i. e., not during the user agent's query), the user agent does not see any additional communication costs other than the messages it sends to and receives from the Region Manager.

Lastly, to improve performance one must consider the characteristics of the region in which this implementation will be used. The operation costs are functions of variables which are dependent on region topology and other region characteristics (such as S and N). A knowledge of which of these variables dominate is invaluable to analyzing and improving efficiency. Also, the bounds given for running time or storage space sometimes depend on region topology. For example, the $O(S + A + N)$ upper bound on the space requirement for adding a region is only reached when no previous path exists from the parent region to the services in the child region to be added. In addition, the implementation should be tested extensively in the region, if possible. This provides a more accurate picture of where the bottlenecks are, and how to tune the system parameters for performance, especially those for caching and updating.

6.3.2 Tradeoffs

The cost analysis of this prototype implementation also sheds some light on the various tradeoffs that can be made within the region infrastructure, and how they affect various aspects of performance. Some of these tradeoffs are made by the implementor of the region infrastructure, and others are made by the administrator of a region.

One of the choices made by the implementor is how to represent the region as one or more data structures. The tradeoff here is between storage and computation costs; some representations are more compact, but they may require more computation to derive the desired information. In our own implementation, the basic data structures are a tree that represents region structure and a separate table of service entries containing the attributes, ID, and parent region. Auxiliary tables store tree nodes and services keyed in different ways. Only one copy of a tree node or service entry is stored. This makes for a compact representation. However, it requires additional management such as reference counting. It also renders certain operations slightly more expensive, such as finding all services with a given parent region. This requires a pass through the table of service entries, while a different representation could accomplish it in constant time.

Another decision borne partly by the implementor and partly by the administrator is how to distribute costs between the various operations. Some of the costs are "hard," but there is sometimes flexibility as to which operations should bear these hard costs. For example, if a region needs to respond to a user agent query, then at some point it must receive a message of size $\Theta(S+A)$ to get the necessary information from its subregion, making the message size a hard cost. In our implementation, the operation for adding a region bears this cost. However, this cost could be deferred to the lookup operation, which would result in a less-expensive region-adding operation, but would add to the latency perceived by the user agent. For another example, suppose we dispensed with caching in our implementation. The space requirement for adding a region would be reduced to almost nil, but the communication cost for lookups would become very high. Needless to say, the choice of caching and update strategy influences the distribution of costs between operations, which is why the region administrator is involved in this tradeoff.

6.3.3 Hidden Costs

One last contributor to the overall view of performance is the hidden costs of the underlying mechanisms and infrastructure used to implement regions. In our implementation, these stem from URN resolver traffic, Java's Remote Method Invocation (RMI) mechanism, and Jini.

Although URN resolution is only listed once in the cost analysis of the region-adding operation, it also occurs during the service-adding operation, although it does not play as integral a role for that operation. There are also redundant URN lookups within the region-adding operation itself. Many of these repeated queries to the URN resolution service are probably unnecessary, and they serve as another argument for caching these queries. Nevertheless, they do occur in this implementation, and can add to the cost of operations.

The use of RMI may also incur a significant cost in our implementation. Typically, the marshalling and demarshalling of arguments and return values for remote method calls is an expensive operation. During development, we treated RMI as a black box and did not attempt to assess its impact on performance. However, it would be foolish to assume that RMI's contribution to the total cost is negligible.

Lastly, Jini itself has some bearing on the overall performance of the system. Tables A.7, A.10, and A.13 in Appendix A list parts of basic region operations whose costs depend on the implementation of Jini. Again, we did not attempt to assess the performance of Jini beyond a brief examination of the source code, but these costs must be taken into account as well.

6.4 Summary

Predicting and analyzing the performance of the region infrastructure is difficult because it is a complex system with many variables and many arbitrary choices that can be made. Among the many factors that influence performance are the characteristics of the region, the network layout of the infrastructure, the caching and update policy adopted, and any other policy goals set by the administrator. Performance can be assessed in several ways, including the amount of bandwidth used, the rate of update propagation, the load on Region Managers, and the time required for lookups. However, the final measure of performance is how well the region infrastructure meets its goals, which are defined by the region administrator and expressed in terms of these quantities.

We have conducted an analysis of the costs of basic region operations in our own region infrastructure implementation. By doing so, we have uncovered several general issues that may provide guidance in assessing the performance of any region system. Some of these involve efficiency. We have identified communication costs and the region-add operation as bottlenecks. We suggest URN query caching, consolidation of the Region Manager and Jini lookup, batch updates, and region testing in order to increase efficiency. We also examined tradeoffs in data structure representations and in cost distribution. Finally, we took a look at some of the hidden costs of the implementation, which included costs from URN query traffic, RMI, and Jini.

Much remains to be done in the area of region infrastructure performance. An important question which remains open is the question of how regions can be categorized according to their characteristics. If related regions can be classified by topology, frequency of change, and other attributes, this would greatly assist research on what parameter choices work best for a particular family of regions. And of course, these efforts need to be validated by extensive testing of different types of regions with different configurations of the region infrastructure. Together, these research efforts will provide a more comprehensive picture of region infrastructure performance.

Chapter 7

Conclusion

7.1 Future Work

Many avenues for exploration remain in the region infrastructure project, and there are also many areas for improvement. Future efforts will be aimed at specification languages for attributes, security mechanisms for regions, distributed operation and failure, improvements in implementation, and characterization of regions for performance analysis.

Services within regions might benefit from a common scheme for specifying their attributes and values. Such a scheme might specify the set of attributes that a particular service (like a printer service) might have, as well as the format used to express the attribute value. An attribute specification language needs to support rich, specialized feature sets for services, yet be applicable across all services. It must also be extensible, especially since new types of services must be supportable. Right now, attribute schemes tend toward the ad-hoc, which is sufficient for locally-used services whose attributes have human-readable names and values. However, a broader solution is required for regions, which may cover large areas and large numbers of services, and which requiring sharing of information across different administrative domains.

Security is another area that requires further study as it applies to regions. Security efforts should focus on identifying the principals in the region system and developing authentication and access control mechanisms to ensure that only authorized user agents, regions, or services can join, leave, or otherwise obtain information from a region. Regions also have special trust issues relating to service and region types, authorization based on type, and maintaining the trustworthiness of type information. These issues must addressed, possibly with a digital signature scheme as mentioned earlier.

The caching and update strategies used by the region infrastructure give rise to issues with distributed operation and failure that have not been fully addressed. Methods must be developed for maintaining consistency across Region Manager caches, and further investigation is required on how to define an acceptable level of consistency. In addition, the Region Manager needs to be able to detect and recover from distributed failure; it should have a way to repair the cache and restore a consistent world view.

Our implementation of the region infrastructure stands to benefit from a few changes. The measures for improving efficiency discussed in Section 6.3.1 should be applied. The lookup and matching algorithms can also be replaced with faster versions. Issues with the GUI and thread safety should be resolved, and error-checking and robustness should be strengthened.

Lastly, additional work must be done on characterizing region based on their features. The questions of what features are relevant, and how they can be quantified need to be addressed. Also, it would be valuable to determine if there are any "common case" region configurations that would be particularly beneficial to analyze. Once regions can be characterized, research should be done on which region system parameter values are good and bad for the various region categories, especially in the context of caching and update policy choices. Finally, this research should be backed up by extensive simulation and testing.

7.2 Final Thoughts

The future of computation that we envision is no longer bound by notions of centralized computing by immobile PCs. Having moved beyond the age of the massive mainframe, we are now moving beyond the age of the desktop PC as the center of computation. With the advent of laptops, cellular phones, pagers, PDAs, and computer chips in just about every appliance and device imaginable, the power of computation and communication is cropping up everywhere.

Just as the networking of PCs ushered in a new era in computing, we believe that networking these devices and appliances has the potential to unleash a new wave of distributed computing with mobile, lightweight devices. In particular, we believe that networking these devices would open the floodgates for powerful and innovative applications. These applications would be created by intelligent agents called catalysts. The catalysts combine the various lower-level services provided by the devices into a new application that accomplishes a higher-level goal.

But what is the best way to organize services into a network? Until now, most approaches have been ad-hoc networks that tie together a small number of devices or services. A more systematic approach is necessary in order to enable catalyst application-building on a large scale and tap the true potential of this networking concept. Our approach must have a means for sharing information about a service and its capabilities. It must provide independent administration of groups of services, yet permit sharing of service information across administrative boundaries. It must scale to large numbers of services, which may be widely dispersed geographically or in the network. It must provide scoping, which allows catalysts to work with a manageable and meaningful set of services. And it must be robust and secure.

Our answer to this problem is regions. Regions are basically groups of services. Regions can be nested, overlapping, or combined in arbitrary ways. Most importantly, they can be queried for information about the services they contain, and they can be searched for services that meet a certain set of criteria.

The region infrastructure provides the supporting framework for regions. It consists of the region architecture and the service architecture, which have their roots in Jini and SLP. The region architecture is composed of a three-part model of user agents, services, and Region Managers, which is extended using hierarchy to give it better scaling and organizational properties. The Region Manager manages subregion and service information, responds to user agent queries about services, and handles caching for performance. The region architecture also provides a type model and a URN resolution service. The service architecture provides interfaces that allow services to expose different attributes, values, and functions depending on the region or region type. It also has a registration and update protocol which allows services to join and leave regions and send updates to regions when the attributes of the service change, all with a minimum of configuration.

Our prototype implementation demonstrates the capabilities of the region infrastructure. The implementation provides a Region Manager with full caching and immediate individual updates. The implementation also includes a generic service framework that can be extended to create specialized services that will work with regions. A URN resolution service and a Region Browser for building and browsing regions are also included. All of these come with convenient GUI tools and interfaces. The implementation also makes extensive use of Jini, which introduces some issues when applied to regions, but provides a wealth of features that enable rapid development.

Our prototype also provides some valuable lessons about the performance of the region architecture and some of the factors that influence performance, such as region characteristics, physical layout, caching and update strategy, and policy goals. Analyzing the costs of region operations in our implementation led to insights about efficiency, tradeoffs, and hidden costs that can be applied to any region infrastructure implementation or configuration.

How will regions be used in the future? One can envision them in a variety of settings. Universities could have their own regions, with subregions to manage the resources of various departments and labs. A global corporation with offices worldwide could use regions to group together the resources in each office, and to connect offices or divisions that are geographically distributed but share resources. Regions could tie together participants in a company teleconference. A company employee in the Sydney office could access services in the Hong Kong office. These services would not only include PCs, files, and databases, but also laptops, printers, audio/visual services, voice mail, fax, and PDAs. Regions also have a wealth of potential in the home. Imagine a "chef" catalyst using services in the kitchen region to brew coffee and make breakfast every morning. Another catalyst could integrate the PC, VCR, TV, video game system, and stereo system in the living room region into an impressive, Internet-enabled home entertainment system.

Regions group services together in flexible and powerful ways. They make services widely available and easy to use, on a large scale. They make it possible for users and agents to find exactly the kind of services they need, within the boundaries they specify. By accomplishing all of these goals, regions help break down the barriers that separate our futuristic vision from reality.

Appendix A

Cost Analysis of Basic Region Operations

The following tables present a cost analysis of the basic region operations, based on our implementation of the region infrastructure. Three tables are shown for the first five operations. The first table outlines the steps that take place during the operation. The second table shows the cost per region ancestor for propagating update messages up the region hierarchy. In both of these tables, the steps shown in boldface contribute to the cost of the operation, and the cost is shown to the right. The third table summarizes the information in the previous two tables. Only the first and third type of table are shown for the last operation, since it does not involve propagating updates.

Note that our implementation uses hash tables, so most insertion/deletion/lookup operations are O(1). Note also that the costs of some operations depend on Sun Microsystems' Jini implementation, over which we had no control.

A.1 Adding a Region

Tables A.1–A.3 show the costs for region R_1 to add region R_2 as a child region. The variables in this and the next set of tables are as follows: S is the number of services contained by R_2 (including non-local services), A is the total number of attributes for all S services, and N is the number of nodes in the subtree consisting of R_2 and its descendents.

Table A.1: Operations and costs for R_1 to add R_2 as a child region.

Operations	Time	Space	Message Size
R_1 finds R_2's Region Manager			
R_1 resolves R_2's name			
R_1 sends message to URN resolution service	-	-	O(1)
URN resolution service looks up URL for R_2	O(1)	-	-
URN resolution service sends message to R_1	-	-	O(1)
R_1 queries R_2's Jini lookup for its Region Manager service			
R_1 sends message to Jini lookup	-	-	O(1)
Jini lookup finds Region Manager service	O(1)	-	-
Jini lookup sends message to R_1	-	-	O(1)
R_1 adds R_2 to its set of child regions	O(1)	O(1)	-
R_1 queries R_2 for all of its services			
R_1 sends a message to R_2	-	-	O(1)
R_2 sends a message to R_1	-	-	$\Theta(S + A)$
R_1 adds R_2's services to its set of services	$\Theta(S)$	$O(S + A)$	-
R_1 queries R_2 for its subtree			
R_1 sends a message to R_2	-	-	O(1)
R_2 sends a message to R_1	-	-	Θ(N)
R_1 inserts an edge in its graph between itself and R_2	O(1)	-	-
R_1 adds R_2's subtree nodes to its set of nodes	Θ(N)	O(N)	-
R_1 sends update messages to its parents			
R_2 adds R_1 to its set of parent regions	O(1)	O(1)	-

Table A.2: Update cost per ancestor of R_1.

Operations	Time	Space	Message Size
Receive update message	-	-	$\Theta(S + A)$
Add child region's services to set of services	$\Theta(S)$	$O(S + A)$	-
Look up nodes for parent and child regions	O(1)	-	-
Add child region's subtree nodes to set of nodes	Θ(N)	O(N)	-

A.2 Removing a Region

Tables A.4–A.6 show the costs for region R_1 to remove its child region R_2.

A.3 Adding a Service

Tables A.7–A.9 show the costs for region R to add service S. The variables in this and the next two sets of tables are as follows: A is the number of attributes that S has, C is the number of types S instantiates (i. e., how many classes and interfaces it has, including all supertypes), L is R's Jini lookup, and E is the number of services already registered in L.

A.4 Removing a Service

Tables A.10–A.12 show the costs for region R to remove service S.

A.5 Modifying a Service

Tables A.13–A.15 show the costs for service S to modify its attributes in region R.

A.6 Looking Up Services

Tables A.16–A.17 show the costs for user agent U to look up services in region R. A is the total number of attributes in all of R's services. C is the total number of types instantiated by all services in R. A_Q and C_Q are the number of attributes and the number of types, respectively, in the user agent's query. S_M is the number of matching services in R, and A_M is the total number of attributes in all S_M matching services.

A.7 Summary

Table A.18 summarizes the costs for the various basic region operations.

Table A.3: Summary of costs for R_1 to add R_2 as a child region.

Costs	Time	Space	# of Msgs	Message Size
Parent region	$\Theta(S + N)$	$O(S + A + N)$	6	O(1)
Child region	O(1)	O(1)	2	$\Theta(S + A)$, Θ(N)
Update cost per ancestor	$\Theta(S + N)$	$O(S + A + N)$	1	$\Theta(S + A)$, Θ(N)

Table A.4: Operations and costs for R_1 to remove child region R_2.

Operations	Time	Space	Message Size
R_1 removes R_2 from its set of child regions	O(1)	-	-
R_1 queries R_2 for all of its services			
R_1 sends message to R_2	-	-	O(1)
R_2 sends message to R_1	-	-	$\Theta(S + A)$
R_1 removes R_2's services from its set of services	$\Theta(S)$	-	-
R_1 removes the edge in its graph between itself and R_2	O(1)	-	-
R_1 removes R_2's subtree nodes from its set of nodes	Θ(N)	-	-
R_1 sends update messages to its parents			
R_2 removes R_1 from its set of parent regions	O(1)	-	-

Table A.5: Update cost per ancestor of R_1.

Operations	Time	Space	Message Size
Receive update message	-	-	$\Theta(S + A)$
Remove child region's services from set of services	$\Theta(S)$	-	-
Look up nodes for parent and child regions	O(1)	-	-
Remove child region's subtree nodes from set of nodes	Θ(N)	-	-

Table A.6: Summary of costs for R_1 to remove child region R_2.

Costs	Time	Space	# of Msgs	Message Size
Parent region	$\Theta(S + N)$	-	1	O(1)
Child region	O(1)	-	1	$\Theta(S + A)$
Update cost per ancestor	$\Theta(S + N)$	-	1	$\Theta(S + A)$

Table A.7: Operations and costs for R to add S.

Operations	Time	Space	Message Size
S joins R's Jini group			
S **sends message to** L	-	-	$\Theta(A)$
L **adds** S **to its set of services**[1]	Θ(E + C + A)	$\Theta(A)$	
L notifies R's Region Manager about S			
L **sends message to** R	-	-	$\Theta(A)$
R **adds** S **to its set of services**	O(1)	O(A)	-
R sends update messages to its parents			

[1]Dependent on Jini implementation.

Table A.8: Update cost per ancestor of R.

Operations	Time	Space	Message Size
Receive update message	-	-	$\Theta(A)$
Add service to set of services	O(1)	O(A)	-

Table A.9: Summary of costs for R to add S.

Costs	Time	Space	# of Msgs	Message Size
Region	O(1)	O(A)	-	-
Service	-	-	1	$\Theta(A)$
Jini lookup	Θ(E + C + A)	$\Theta(A)$	1	$\Theta(A)$
Update cost per ancestor	O(1)	O(A)	1	$\Theta(A)$

Table A.10: Operations and costs for R to remove S.

Operations	Time	Space	Message Size
S leaves R's Jini group			
S **sends message to** L	-	-	O(1)
L **removes** S **from its set of services**[1]	$\Theta(C + A)$	-	-
L notifies R's Region Manager about S leaving			
L **sends message to** R	-	-	O(1)
R **removes** S **from its set of services**	O(1)	-	-
R sends update messages to its parents			

[1]Dependent on Jini implementation.

Table A.11: Update cost per ancestor of R.

Operations	Time	Space	Message Size
Receive update message	-	-	O(1)
Remove service from set of services	O(1)	-	-

Table A.12: Summary of costs for R to remove S.

Costs	Time	Space	# of Msgs	Message Size
Region	O(1)	-	-	-
Service	-	-	1	O(1)
Jini lookup	Θ(C + A)	-	1	O(1)
Update cost per ancestor	O(1)	-	1	O(1)

Table A.13: Operations and costs for S to modify its attributes in R.

Operations	Time	Space	Message Size
S notifies L of changes			
S sends message to L	-	-	$\Theta(A)$
L modifies S's entry in its set of services[1]	Θ(E + C + A)	O(A)	-
L notifies R's Region Manager about S changing			
L sends message to R	-	-	$\Theta(A)$
R replaces S in its set of services	O(1)	O(A)	-
R sends update messages to its parents			

[1]Dependent on Jini implementation.

Table A.14: Update cost per ancestor of R.

Operations	Time	Space	Message Size
Receive update message	-	-	$\Theta(A)$
Replace service in set of services	O(1)	O(A)	-

Table A.15: Summary of costs for S to modify its attributes in R.

Costs	Time	Space	# of Msgs	Message Size
Region	O(1)	O(A)	-	-
Service	-	-	1	$\Theta(A)$
Jini lookup	Θ(E + C + A)	O(A)	1	$\Theta(A)$
Update cost per ancestor	O(1)	O(A)	1	$\Theta(A)$

Table A.16: Operations and costs to look up services in R.

Operations	Time	Space	Message Size
U queries R for services matching a given set of attributes and interfaces			
U sends a message to R	-	-	$\Theta(A_Q + C_Q)$
R looks up matching services in cache	$O(A_Q A + C_Q C)$	-	-
R sends a message to U	-	-	$\Theta(S_M + A_M)$

Table A.17: Summary of costs to look up services in R.

Costs	Time	Space	# of Msgs	Message Size
User agent	-	-	1	$\Theta(A_Q + C_Q)$
Region	$O(A_Q A + C_Q C)$	-	1	$\Theta(S_M + A_M)$

Table A.18: Summary of costs for basic region operations.

Costs	Time	Space	# of Msgs	Message Size
Adding a region				
Parent region	$\Theta(S + N)$	$O(S + A + N)$	6	O(1)
Child region	O(1)	O(1)	2	$\Theta(S + A)$, Θ(N)
Update cost per ancestor	$\Theta(S + N)$	$O(S + A + N)$	1	$\Theta(S + A)$, Θ(N)
Removing a region				
Parent region	$\Theta(S + N)$	-	1	O(1)
Child region	O(1)	-	1	$\Theta(S + A)$
Update cost per ancestor	$\Theta(S + N)$	-	1	$\Theta(S + A)$
Adding a service				
Region	O(1)	$O(A)$	-	-
Service	-	-	1	$\Theta(A)$
Jini lookup	Θ(E + C + A)	$\Theta(A)$	1	$\Theta(A)$
Update cost per ancestor	O(1)	$O(A)$	1	$\Theta(A)$
Removing a service				
Region	O(1)	-	-	-
Service	-	-	1	O(1)
Jini lookup	Θ(C + A)	-	1	O(1)
Update cost per ancestor	O(1)	-	1	O(1)
Modifying a service				
Region	O(1)	$O(A)$	-	-
Service	-	-	1	$\Theta(A)$
Jini lookup	Θ(E + C + A)	$O(A)$	1	$\Theta(A)$
Update cost per ancestor	O(1)	$O(A)$	1	$\Theta(A)$
Looking up services				
User agent	-	-	1	$\Theta(A_Q + C_Q)$
Region	$O(A_Q A + C_Q C)$	-	1	$\Theta(S_M + A_M)$

Bibliography

[1] Kathryn F. Benedicto. Regions: A scalable infrastructure for scoped service location in ubiquitous computing (web site with source code). http://www.ana.lcs.mit.edu/puc/java-regions/, May 1999.

[2] William J. Bolosky, Richard P. Draves, Robert P. Fitzgerald, Christopher W. Fraser, Michael B. Jones, Todd B. Knoblock, and Rick Rashid. Operating system directions for the next millennium. Technical report, Microsoft Research, 1999.

[3] Microsoft Corp. The Millenium research project. http://www.research.microsoft.com/research/os/millennium/, March 1999.

[4] Bluetooth Special Interest Group. Bluetooth. http://www.bluetooth.com/, March 1999.

[5] Caltech Infospheres Group. Caltech Infospheres Project description. http://www.infospheres.caltech.edu/infospheres.html, March 1999.

[6] Eric Guttman, Charles Perkins, John Veizades, and Michael Day. Service Location Protocol. RFC 2165, Internet Engineering Task Force, June 1997.

[7] Eric Guttman, Charles Perkins, John Veizades, and Michael Day. Service Location Protocol, Version 2. Internet Draft, Internet Engineering Task Force, April 1999. Work in progress.

[8] HAVi website. http://www.havi.org/, March 1999.

[9] G. Klyne. A syntax for describing media feature sets. RFC 2533, Internet Engineering Task Force, March 1999.

[10] L. Masinter, D. Wing, A. Mutz, and K. Holtman. Media features for display, print, and fax. RFC 2534, Internet Engineering Task Force, March 1999.

[11] Paul V. Mockapetris and Kevin J. Dunlap. Development of the Domain Name System. In *Proc. of SIGCOMM Symposium*, pages 123–133, 1988. Also published as ACM Computer Communications Review 18, 4 (August, 1988).

[12] California Institute of Technology. Caltech Infospheres Project. http://www.infospheres.caltech.edu/, March 1999.

[13] K. Sollins. Architectural principles of uniform resource name resolution. RFC 2276, Internet Engineering Task Force, January 1998.

[14] K. Sollins and L. Masinter. Functional requirements for uniform resource names. RFC 1737, Internet Engineering Task Force, December 1994.

[15] Sun Microsystems, Inc. Jini. http://www.javasoft.com/products/jini/, December 1998.

[16] Sun Microsystems, Inc. Jini Discovery and Join Specification. http://www.sun.com/jini/specs/boot.ps, January 1999.

[17] Lucent Technologies. Inferno. http://www.lucent-inferno.com/, March 1999.

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