An Application Layer Framework for Location- based Service Discovery and Provisioning for Mobile Devices

by

Sunil Gopinath

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Dr. Scott F. Midkiff, Co-Chair

Dr. Srinidhi Varadharajan, Co-Chair

Dr. Sallie Henry

An Application Layer Framework for Location-based Service Discovery and Provisioning for Mobile Devices

Sunil Gopinath

(ABSTRACT)

There has been a tremendous rise in the use of Wireless Application Protocol (WAP) services for cellular telephones. Such services include electronic mail, printing, fax delivery, and weather reports. But, current services are limited both in type and nature. Today, mobile telephone users need access to more dynamic, location-based, distributed services that include both hardware resources, like printers and computers, and software services, like application software. Problems due to mobility include clients disconnecting from the network, services leaving the network, and communication problems.

This research proposes and demonstrates the feasibility of a framework for a system to meet such a need. More specifically, this work develops and demonstrates a distributed environment where mobile telephone users have access to services dynamically as they enter and leave different service areas. It also provides a framework to support mobility in the application layer context.

This work utilizes Sun Microsystem's JINI connection technology [4] to provide distributed services to mobile telephones over WAP. It provides a prototype system to provide Java based software services to mobile telephones. The work also provides several optimizations with respect to client

communication by harnessing key features of WAP. This provides a robust, dynamic environment for service provisioning.

Acknowledgements

I would like to thank my advisor, Dr. Scott Midkiff, for his excellent guidance and support during the progress of my thesis. I would also like to thank my committee members, Dr. Srinidhi Varadharajan and Dr. Sallie Henry, for providing useful comments on my thesis.

I thank Mr. Scott Farmer for supporting me through a USDA-funded project throughout my stay at Virginia Tech and for being a great source of support. I also thank all my colleagues at the Public Service programs and all my friends for their support.

Last, but not the least, I would like to thank my parents and my brother for their love and encouragement.

Table of Contents

List of Figures

Chapter 1. Introduction

1.1. Problem

The Wireless Application Protocol (WAP) is proposed as the gateway to a new world of mobile data services [1]. It provides a universal standard that enables users to access web-based interactive information services and applications from their mobile telephones. Common services provided by most WAP service providers include electronic mail, personal information managers (PIM) that provide subscribers with integrated calendar, address book, to-do list tools, and other functions. Despite the current excitement, "killer" WAP applications have yet to present themselves. Being able to receive up-to-date news and weather information on their telephone is not an exciting prospect for potential buyers of WAP phones and services. The problem with WAP is that slowly, but surely, cellular telephone companies have begun to admit that WAP is really just a transition technology that paves the way for something faster and better like I-mode technology [22]. Improved services will be provided by the next big step of wireless infrastructure, the Universal Mobile Telecommunications System (UMTS) the official specification of which will be released in 2001 [11].

An alternative technology on the rise is Sun Microsystem's Java 2 Micro Edition (J2ME) [13] and Connected Limited Device Configuration (CLDC) [12]. These are designed to enhance and ease application development for resource-constrained, connected devices, such as mobile telephones, pagers, personal digital assistants (PDAs), home appliances, and point-of-sale terminals.

J2ME and CLDC provide a minimal layer of Java 2's core application programming interfaces (APIs) targeted at providing enough functionality to securely and safely download Java classes to a device and configure the Java environment. This allows devices to be configured dynamically over a wireline or wireless network by downloading classes not included with the J2ME library and, thus, attain the functionality and provide the same services of current, more powerful computers. J2ME has been designed to run on devices with a minimum of 128 kilobytes (KB) of available memory for the Java environment. Applications developed with Java 2 Micro Edition run on the K Virtual Machine (KVM) [12]. The KVM is currently 40 KB of object code, requires 128 KB of available memory, and runs on 16- or 32-bit processors with a minimum speed of 25 megahertz (MHz).

Hence, for WAP to retain its current level of interest and usefulness to the wireless community, service providers need a framework that offers widespread location-based services to WAP-enabled devices. Such a framework might sustain the current interest in WAP until newer technologies, like UMTS and J2ME, are widely available. Implementing this framework could provide a tremendous boost in dynamic service and resource provisioning for current users of WAP-enabled devices.

1.2. Approach

In this research, a framework for providing dynamic, location-based service discovery for mobile telephones that is based on the Wireless Markup Language (WML) was developed and demonstrated. The framework consists of

a set of Java servlets that provide an interface and execution context to mobile telephones to dynamically avail themselves of services and resources in the current service area. The framework makes use of Jini connection technology [4] for service discovery, lookup, and event management. It also provides a set of applications relevant to mobile telephones to demonstrate the versatility offered by dynamic service discovery, including providing addresses of the nearest domain name server (DNS) and two-way paging via WAP Push technology to reduce the cost of wireless telephony. Framework components interact with the mobile telephone using WML over WAP and with the Jini Lookup Server and services though Java remote method invocation (RMI) [14]. The framework also provides caching of frequently used services at the web-server in order to improve efficiency and the speed of transactions. It also provides cache validation to the mobile telephone to provide up-to-date notification of services currently available in the network as services dynamically enter and leave a network.

1.3. Related Work

Many related technologies for service discovery exist. Some of the most important ones are Jini [4], UPnP [16], and Salutation [17]. This work evaluated the existing technologies and chose to use Jini for service discovery and lookup since the focus of this work was primarily Java-based software services. Also, Jini is protocol independent since it deals mainly with the application layer. In addition, Jini provides code downloading which is essential for querying interfaces and this is one of the most important aspects of this work. UPnP and

Salutation are discussed in this section and Jini is discussed in Section 3.1, as it is an essential part of this research.

1.3.1. Universal Plug and Play (UPnP)

UPnP [16] is a framework defined at a much lower level than Jini. UPnP works primarily with the Transmission Control Protocol/Internet Protocol (TCP/IP) network protocol suite, implementing standards at the network level instead of at the application level. This primarily involves adding certain optional protocols to the TCP/IP suite that can be implemented natively by devices. By providing a set of defined network protocols, UPnP allows devices to build their own APIs that implement these protocols in whatever language or platform they choose.

UPnP uses a special protocol known as the Simple Service Discovery Protocol (SSDP) [23] that enables devices to announce their presence to the network as well as discover available devices. SSDP uses the HyperText Transfer Protocol (HTTP) over both unicast and multicast instances of the User Datagram Protocol (UDP). The registration and query process will send and receive data in HTTP format, but the data will have special semantics. An announcement message, called “ANNOUNCE,” and a query message, called “OPTIONS,” are embedded in HTML to facilitate registration and query. A device joining the network can send out a multicast ANNOUNCE telling the network about itself using a reserved multicast channel to which all devices must listen. The ANNOUNCE message must also contain a Universal Resource Identifier (URI) that identifies the resource (e.g., "dmtf:printer") and a Universal Resource Locator (URL) for an Extensible Markup Language (XML) file that provides a

description of the announcing device. A query for device discovery, i.e., an OPTIONS message, can also be multicast to which devices may respond.

UPnP also addresses the problem of automatic assignment of IP addresses and DNS names to a device being plugged in to the system. If a Dynamic Host Control Protocol (DHCP) server is not available, then the device can choose an IP address from a reserved IP address range known as the LINKLOCAL network with addresses in the range 169.254.x.x. It must use the Address Resolution Protocol (ARP) to find an unused address in this range locally and, then, assign the address to itself. This is known as AutoIP. Since this is not a routable IP address, packets will not cross gateways. If a DHCP server becomes available at any time, the devices will attempt to switch their IP address to an address provided by the DHCP server.

To address issues in naming, a multicast DNS proposal has been drafted [16]. This would allow connected devices to discover DNS servers via multicast and also resolve DNS queries about itself via multicast. When plugged into a network having no DNS servers locally, a device can send out a multicast packet and discover a remote DNS server and then use the server for Internet name resolution. The device itself can optionally listen on the multicast channel and respond to queries for its own name. However, security issues have yet to be addressed by UPnP. Once the discovery process is completed and the XML description of a device is received, proprietary protocols can take over in communicating with the devices.

UPnP does not address the problem of invoking services. Even though style sheets can describe device capabilities, it would still be difficult to talk to any device of the same class. For example, sending a page to a Hewlett-Packard printer or a Fujitsu printer would still need drivers even though both would be members of the same class, unless both support a protocol like Hewlett-Packard's JetSend protocol or some other specification. By concentrating only on base-level discovery and device capability querying, UPnP leaves a void in this area, presumably to be filled by future additions.

1.3.2. Salutation

In Salutation, a device talks directly to a Salutation Manager, which may be in the same device or located remotely. A system includes an ensemble of Salutation Managers (SLMs) that coordinate with one another. They act like agents that do everything on behalf of their clients. Even data transfer between devices, including those across different media and transport layers, is mediated through SLMs. The framework provides callbacks into the devices to notify the devices of events like data arriving or devices becoming unavailable. All registration is done with the local or nearest available SLM. SLMs discover other nearby SLMs and exchange registration information, even with those on different transports or media. The latter is done using appropriate transport-dependent modules called Transport Managers that may use broadcast internally. Broadcast RPC is optionally used for this. The concept of a "service" is broken down into a collection of Functional Units, with each unit representing some essential feature, such as fax, print, scan, or sub-features like rasterize. A service

description is then a collection of functional unit descriptions, each having a collection of attribute records of the form (name, value). These records can be queried and matched against during the service discovery process. Certain well-defined comparison functions can be associated with a query that searches for a service. The discovery request is sent to the local SLM that, in turn, will be directed to other SLMs. SLMs talk to one another using Sun's Open Network Computer (ONC) Remote Procedure Call (RPC) [21]. Salutation defines APIs for clients to invoke these operations and gather the results.

One of the key features that Salutation tries to provide is media and transport protocol independence. The communication between clients and functional units (services) can occur in a number of ways. In the native mode, devices may use native protocols and talk to one another directly without the SLMs being involved in data transfer. In the emulated mode, the SLMs manage the session and act as a conduit for the data, delivering them as messages to either party. This capability provides transport protocol independence. In the salutation mode, SLMs not only carry the data, but also define the data formats to be used in the transmission. In this mode, the design must adhere to well-defined standards of interoperation with functional units to allow generic inter-operability. All communication in the latter two modes involve APIs and events that are well defined. Data descriptions use Abstract Syntax Notation One (ASN.1) [20] from the International Organization for Standards (ISO) for representation and encoding.

1.4. Contributions of this Work

This research provides a Java-based framework for providing distributed services to mobile devices. The mobile devices for this prototype are WAP-enabled mobile telephones. The work makes use of Sun Microsystem's Jini for providing a lookup server for service discovery. The work also makes use of WAP APIs provided by Phone.com (now part of Openwave Systems) for communication with the mobile telephone. The research also proposes a solution for mobility at the application level and provides optimizations for communication at the application level for mobile telephones.

More specifically, the following components were developed in this research.

1. The communication component, for communication between the mobile telephone and MobiServ. This component provides conversion from Java to WML and message digest optimizations. This component uses WAP APIs provided by Phone.com.
2. The lookup discovery and registration components, which discover Jini lookup servers in the vicinity and register Java software services with the lookup servers.
3. The leasing component, which is a background process that renews leases on behalf of services and purges discarded service entries in the MobiServ leasing registry.
4. The service components, which provide sample services for testing the prototype system. These components are Java applications.

1.5. Thesis Organization

Chapter 2 of this thesis describes the design of a solution and the operation of the system. Chapter 3 discusses the technologies used in the work. Chapter 4 describes the functionality and architecture of the service discovery component. Chapter 5 describes the interaction with the client over WAP. Chapter 6 describes the testing procedure and results. Chapter 7 summarizes the research and includes ideas for further work.

Chapter 2. System Design and Operation

This chapter describes the high-level design of the proposed system and its operation. Figure 2.1 shows the high-level design of the proposed system. When the mobile telephone user enters a network, the user contacts the MobiServ servlet running on the web server via a WAP gateway. The MobiServ servlet extracts user information, such as subscriber identifier (ID) and preferences, and stores them in a database. The mobile telephone user can make a request for a specific service, such as a print service, nearest DNS service, or any other software service such as weather information, in the current location. The MobiServ application provides a WML-based user interface to the mobile telephone. The MobiServ servlet in turn uses JINI lookup discovery to search for a lookup service in the local network. Once a lookup service is found, the MobiServ application communicates with the lookup service via Java RMI to query for the service requested by the telephone user.

If the service is registered with the lookup server, MobiServ downloads the service proxy for that service via HTTP and communicates with the service and invokes appropriate methods on the service. If the service is not available or not registered with the lookup server, MobiServ registers a remote event listener with the lookup service that will be notified by the lookup service when the requested service becomes available.

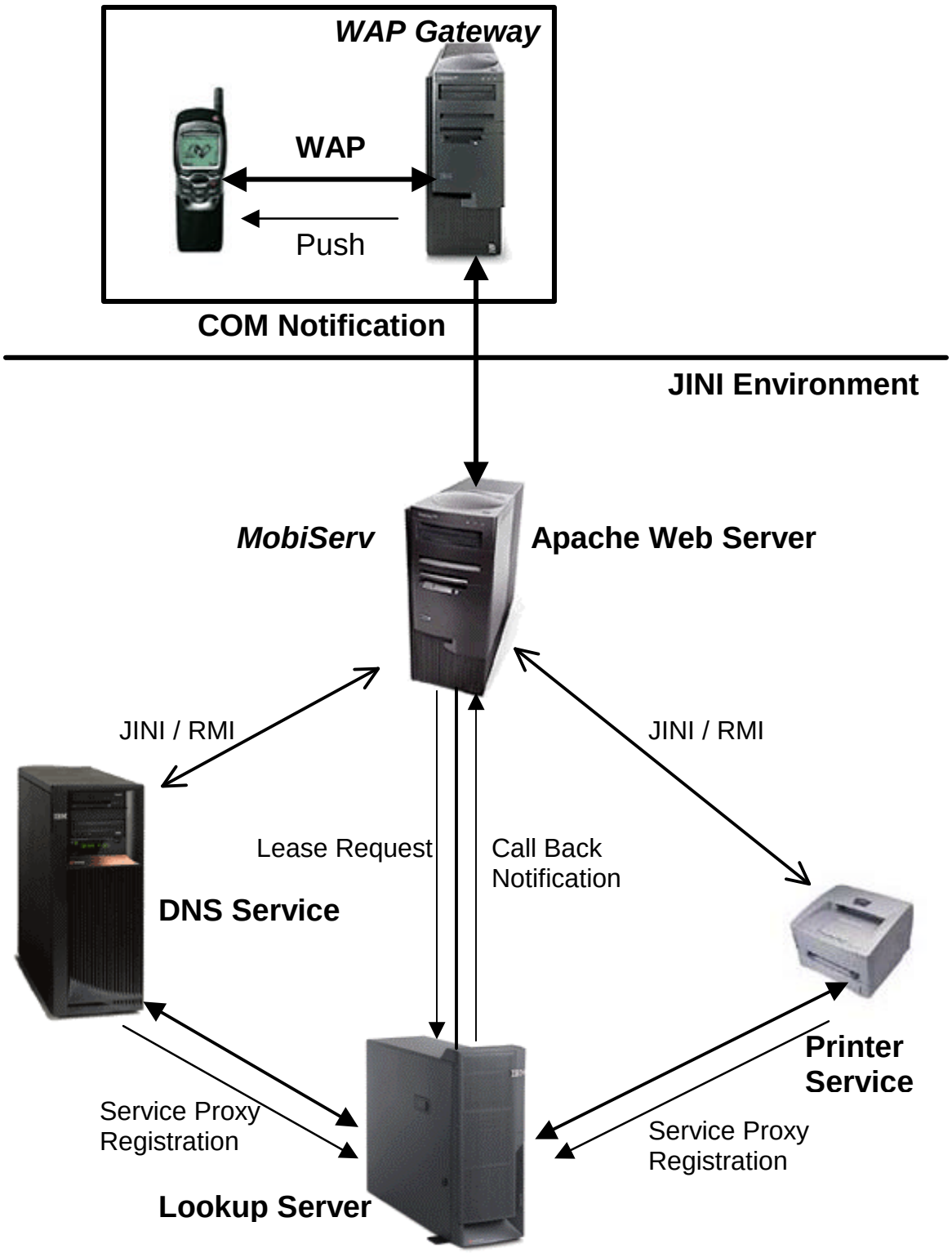


Figure 2.1. High-level design of the system.

When the service becomes available, the *MobiServ* servlet notifies the mobile telephone user by sending a notification via the WAP Push Access Protocol (PAP) [18]. This is available to the mobile telephone user in the form of a WAP Alert. The telephone user can then communicate with the service by communicating via the MobiServ servlet. In effect, MobiServ provides a service context for mobile telephone users to avail themselves of software services in their current location.

The following chapters describe the design in greater detail.

Chapter 3. Background

This chapter provides enough information about Jini and WAP to understand how they are used in this work. The Jini Discovery and Lookup protocols are described. The WAP architecture and features utilized for this work are also described.

3.1. JINI

The heart of the Jini system is a trio of protocols called discovery, join, and lookup [4]. A pair of these protocols, discovery and join, occur when a service becomes available. Discovery occurs when a service is looking for a lookup service with which to register. Join occurs when a service has located a lookup service and wishes to join that service. Lookup occurs when a client or user needs to locate and invoke a service described by its interface type (as specified in Java) and possibly other attributes. Figure 3.1, which shows a service provider seeking a lookup service, illustrates the discovery process.

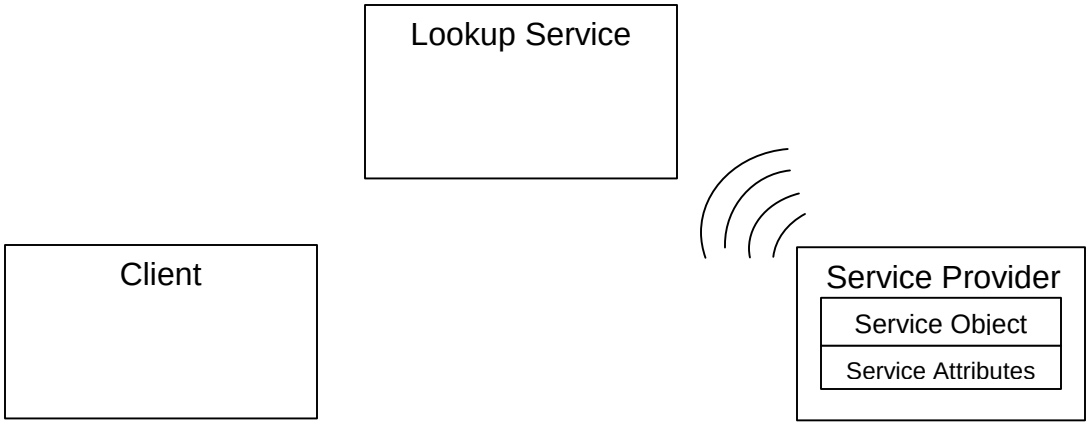


Figure 3.1. The discovery process.

The service provider searches for a lookup service in the current network. Jini discovery/join is the process of adding a service to a Jini system. A service provider is the originator of the service which can be, for example, a device or software. First, the service provider locates a lookup service by multicasting a request on the local network for any lookup services to identify themselves. This is a join action, as shown in Figure 3.2. In Figure 3.2, a service provider registers a service object (proxy) with the lookup service.

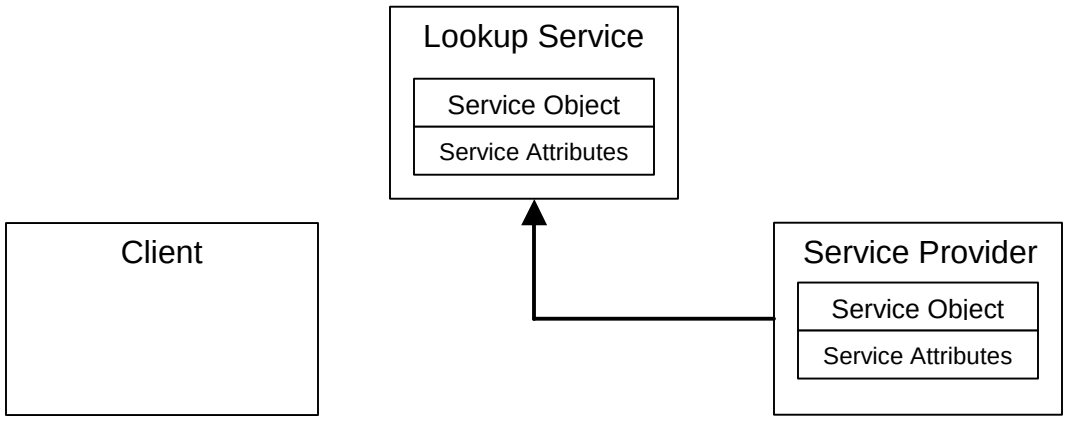


Figure 3.2. The join process.

This service object contains the Java interface for the service, including the methods that users and applications can invoke to execute the service, along with any other descriptive attributes. Services must be able to find a lookup service. However, a service may delegate the task of finding a lookup service to a third party. The service is now ready to be found via lookup and used, as shown in Figure 3.3. In the figure, a client requests a service by Java programming language type and, perhaps, other service attributes. A copy of the

service object is moved to the client and used by the client to communicate with the service.

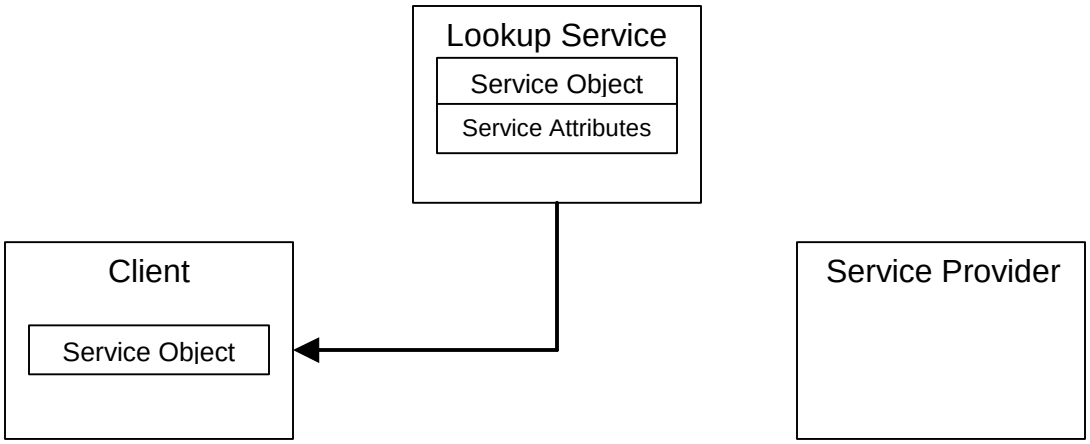


Figure 3.3. The lookup process.

A client locates an appropriate service by its type that is, by its interface as written using the Java programming language, along with descriptive attributes that are used in a user interface for the lookup service. The service object is loaded into the client. The final stage is to invoke the service, as shown in Figure 3.4. The client interacts directly with the service provider via the service object or proxy. The service object's methods may implement a private protocol between itself and the original service provider.

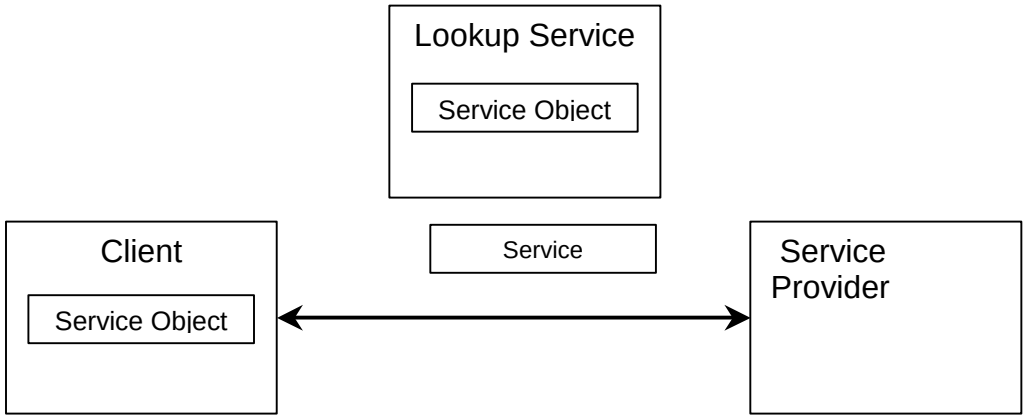


Figure 3.4. Client uses the service.

Different implementations of the same service interface can use completely different interaction protocols. The ability to move objects and code from the service provider to the lookup service and from there to the client of the service gives the service provider great freedom in realizing the communication patterns between the service and its clients. This code movement also ensures that the service object held by the client and the service for which it is a proxy are always synchronized because the service object is supplied by the service itself. The client knows only that it is dealing with an implementation of an interface written in the Java programming language, so the code that implements the interface can do whatever is needed to provide the service. Because this code came originally from the service itself, the code can take advantage of implementation details of the service that are known only to the code. The client interacts with a service via a set of interfaces written in the Java programming language. These interfaces define the set of methods that can be used to interact with the service. Interfaces are identified by the type system of the Java programming language, and services can be found in a lookup service by asking for those that support a particular interface. Finding a service in this manner ensures that the program looking for the service will know how to use that service because that use is defined by the set of methods that are defined by the type.

3.2. WAP

3.2.1. WAP Architecture

WAP content and applications are specified in a set of well-known content formats based on the familiar WWW content formats. Content is transported

using a set of standard communication protocols based on the WWW communication protocols. A microbrowser in the wireless terminal coordinates the user interface and is analogous to a standard web browser.

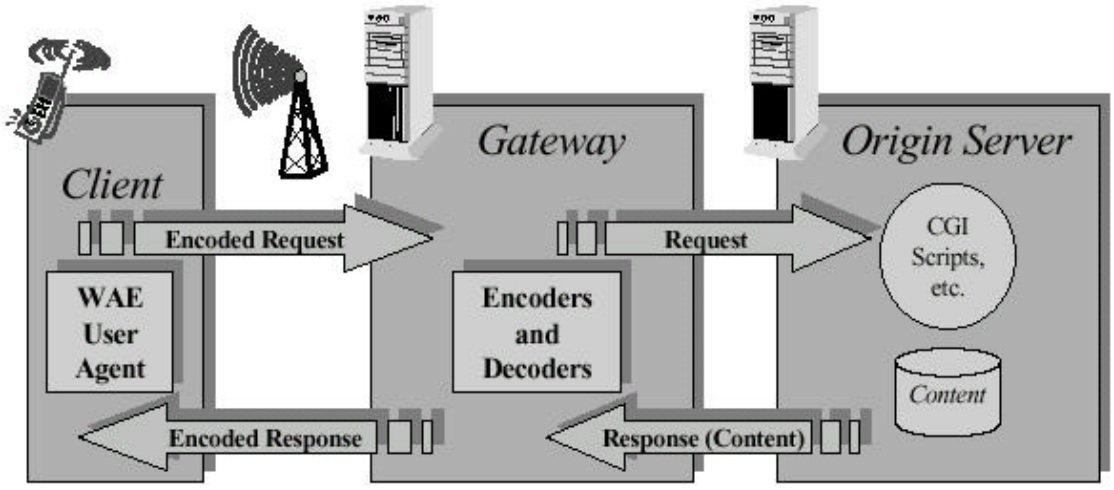


Figure 3.5. WAP model (taken from [1]).

WAP defines a set of standard components, illustrated in Figure 3.5, that enable communication between mobile terminals and network servers, including the following [1].

- Standard naming model – Standard URLs, as used in the World Wide Web (WWW) are used to identify WAP content on origin servers. Standard URIs are used to identify local resources in a device, e.g., call control functions.
- Content typing – All WAP content is given a specific type consistent with WWW typing. This allows WAP user agents to correctly process the content based on its type.

- Standard content formats – WAP content formats are based on WWW formats and include display markup, calendar information, electronic business card objects, images, and a scripting language.
- Standard communication protocols – WAP communication protocols enable the communication of browser requests from the mobile terminal to the network web server.

The WAP content types and protocols have been optimized for mass market, hand-held wireless devices. WAP utilizes proxies to connect between the wireless domain and the WWW. A WAP proxy typically includes the following functionality [1].

- Protocol gateway – The protocol gateway translates requests from the WAP protocol stack to the WWW protocol stack. The WAP protocol stack includes the Wireless Session Protocol (WSP), Wireless Transaction Protocol (WTP), Wireless Transport Layer Security (WTLS), and Wireless Datagram Protocol (WDP). The WWW protocol stack includes HTTP and TCP/IP.
- Content encoders and decoders – The content encoders translate WAP content into compact binary-encoded formats to reduce the size of data transferred over the wireless network, while decoders translate contact back to standard form.

This infrastructure ensures that mobile terminal users can browse a variety of WAP content and applications and that the application author is able to build content services and applications that run on a large base of mobile terminals. The WAP proxy allows content and applications to be hosted on standard web

servers and to be developed using WWW techniques, such as writing Common Gateway Interface (CGI) scripts.

3.2.2. WAP Push

The Push Initiator contacts the push proxy gateway (PPG) from the Internet side, delivering content for the destination client using Internet protocols as shown in Figure 3.6. The PPG does what is necessary to forward the pushed content to the WAP domain and the content is then transmitted over the air in the mobile network to the destination client.

Figure 3.6. WAP push example (taken from [1]).

In addition to providing simple proxy gateway services, the PPG notifies the Push Initiator about the final outcome of the push operation or waits for the client to accept or reject the content in two-way mobile networks. The PPG also provides the Push Initiator with client capability lookup services, letting a Push Initiator select the optimal “flavor” of its particular content for this particular client. The Internet-side PPG access protocol is called the push access protocol (PAP). The WAP-side protocol is called the push over-the-air (OTA) protocol. PAP uses

XML messages that are tunneled through HTTP. The OTA protocol is based on Wireless Session Protocol [1] services.

3.2.3 Service Indication

The Service Indication (SI) content type provides the ability to send notifications to end-users in an asynchronous manner. Such notifications may, for example, indicate new electronic mail messages, changes in a stock price, news headlines, advertising, or a low prepaid balance. In its most basic form, an SI contains a short message and a URI indicating a service. The message is presented to the end-user upon reception and the user is given the choice to either start the service indicated by the URI immediately or postpone the SI for later handling. If the SI is postponed, the client stores it and the end-user is given the option to act upon it at a later time.

3.3. UP.SDK Tools

The following tools [6] were used in designing and testing the client part of the system.

- The UP.SIMULATOR was used to simulate the mobile telephone on a standard personal computer.
- The UP.SDK was used for developing the application for the WAP communication part of the project.
- The UP.LINK WAP gateway was used for sending WAP PUSH messages and communication with the UP.SIMULATOR.

The UP.SDK supports the Microsoft Common Object Model (specifically COM+) notification scheme for sending notifications to the mobile telephone via the WAP gateway [6]. This project makes use of the COM APIs for this purpose.

3.4. Summary

Jini provides a simple Java-based mechanism for service discovery and lookup. To provide an architecture- and operating system-independent interface, this research utilized Jini for service discovery and lookup as it provides a Java-based interface for communication.

Most of the mobile telephone use WAP for accessing Internet-based services. The UP.SDK provides the required APIs for WAP-based communication. This work also utilizes the WAP APIs for client communication.

The next chapter describes the issues involved in service discovery and leasing, features implemented, including distributed events and notifications, application level support for disconnected operations, load balancing and providing wireless markup language (WML)–based services.

Chapter 4. Service Discovery

This chapter describes the service discovery part of the MobiServ framework. The chapter describes in detail the implementation of service discovery and lookup, the interface used for specifying service attributes, leasing, and distributed events. It also describes how disconnected operation is handled using the event mail box service and how load balancing is implemented.

4.1. Discovery and Lookup

The MobiServ framework includes a set of Java servlets that perform the actual service discovery and lookup. When the mobile telephone enters the current service area, it initiates a connection to the MobiServ servlet. The servlet in turn initiates the service discovery by searching for a lookup server in the vicinity. This is done using the Jini Discovery and Join protocols as described in Chapter 3.

The Jini Lookup service is found dynamically so that the clients of the lookup service do not need to know the location of the lookup service. They automatically discover the location at runtime. The Jini lookup service can store any serializable object.

Any service that needs to find and register with a lookup server needs to implement the Discovery Listener interface. This class will be alerted when the service discovers a lookup service and when the service needs to discard a lookup service.

When a new lookup service is detected, the *discovered()* method is called. Once a new lookup service is found, the service registers its object with the

lookup service. The service has to register its service, a Java object, with the lookup server using a ServiceID which is unique among multiple lookup servers that might exist in the same network. When a lookup service is no longer available, the *discarded()* method is called. At this time the service can decide whether to register with a new lookup server or take further action.

4.2. Attributes and the Entry Interface

Attributes are used to specify the particulars of the service. For example, if the service is a printer service its attributes could be information such as the location of the printer, configuration information, such as whether the printer supports two-sided printing or color, or status information, such as whether the printer is out of toner or paper. Attributes are not simple name-value pairs. They implement the Jini Entry interface.

Entry objects are serializable objects that are treated in a special fashion when they are serialized or reconstituted. Matching an Entry object requires an additional Entry object to be used as a template, specifying the field values to search. A template Entry object matches an entry object if the type of the template object is the same as, or a superclass of, the type of the Entry object. The fields within the objects must also have the same values. A template with a null value for a field is treated as a wildcard and matches any value for that field.

4.3. Leasing

Leases are requested for a period of time. In distributed applications, there may be partial failures of the network or of components on this network. Leasing is a

way for components to register that they are alive, but for them to be “timed out” if they have failed, are unreachable, or are otherwise unavailable. In Jini, one use of leasing is for a service to request that a copy be kept on a lookup service for a certain length of time for delivery to clients upon request. The service requests a time in the ServiceRegistrar’s *register()* method. Two special values of the time value for a lease are as follows.

1. Lease.ANY - the service lets the lookup service decide the time of the lease.

2. Lease.FOREVER - the request is for a lease that never expires.

A lease is a time period during which the grantor of the lease ensures to the best of the grantor’s abilities that the holder of the lease will have access to some resource. The time period of the lease can be determined solely by the lease grantor, or can be a period of time that is negotiated between the holder of the lease and the grantor of the lease. Duration negotiation need not be multi-round; it often suffices for the requestor to indicate the time desired and the grantor to return the actual time of grant.

During the period of a lease, a lease can be cancelled by the entity holding the lease. Such a cancellation allows the grantor of the lease to clean up any resources associated with the lease and obliges the grantor of the lease to not take any action involving the lease holder that was part of the agreement that was the subject of the lease. A lease holder can request that a lease be renewed. The renewal period can be for a different time than the original lease and is also subject to negotiation with the grantor of the lease. The grantor may renew the lease for the requested period or a shorter period or may refuse to

renew the lease at all. A renewed lease is just like any other lease and is itself subject to renewal.

A lease can expire. If a lease period has elapsed with no renewals, the lease expires and any resources associated with the lease may be freed by the lease grantor. Both the grantor and the holder are obliged to act as though the lease agreement is no longer in force. The expiration of a lease is similar to the cancellation of a lease, except that no communication is necessary between the lease holder and the lease grantor.

4.4. Distributed Events and Notifications

The basic, concrete objects involved in a distributed event system are as follows.

- The object that registers interest in an event.
- The object in which an event occurs, referred to as the event generator.
- The recipient of event notifications, referred to as a remote event listener.

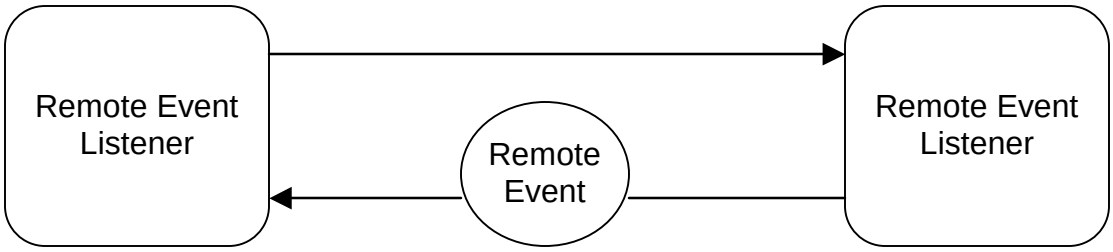
These objects are illustrated in Figure 4.1 and described below.

An event generator is an object that has abstract state changes that might be of interest to other objects and that also allows other objects to register interest in those events. This is the object that will generate notifications when events of this kind occur. The event generator will send these notifications to the event listeners that were indicated as targets in the calls that registered interest in that kind of event.

A remote event listener is an object that is interested in the occurrence of events at some other object. The major function of a remote event listener is to receive notifications of the occurrence of an event in some other object.

A remote event is an object that is passed from an event generator to a remote event listener to indicate that an event of a particular kind has occurred. At a minimum, a remote event contains information about the kind of event that occurred, a reference to the object in which the event occurred, and a sequence number allowing identification of the particular instance of the event. A notification also includes an object that was supplied by the object that registered interest in this kind of event as part of the registration call.

1. The remote event listener registers interest in a particular kind of event with the event generator



2. The event generator fires a remote event to indicate that an event of that kind has occurred

Figure 4.1. Operation of a remote event generator.

MobiServ registers a remote event listener with the Jini Lookup Service. The remote events that will be delivered to MobiServ relate to the changes in the registered services. Mobiserv will be informed if a new service has been added to the lookup service, if a service is being removed from the lookup service, or if the attributes of the service have changed. The Jini lookup service returns to the client a lease that the client must renew. If the lease is cancelled, the remote event listener will stop receiving events.

4.5. Disconnected Operation

An event mailbox service allows the delivery of events from a Jini service to be deferred. This service is useful for several reasons including the following.

- An application might want to disconnect temporarily from the network without losing any events in its absence.
- A service that is eligible for activation might not want to wake up just to receive an event.
- An application might want to batch events.

The Jini event mailbox service supports the above requirements by providing a remote event listener that receives events. The events are placed in a mailbox. The application can, at a later time, such as when the mobile device re-connects to the network, connect to the mailbox and events are delivered to the application.

4.6. Load Balancing

The MobiServ framework also provides load balancing for services. Many different mobile telephones might request a similar service. Depending on the current transaction load on each application or device, the MobiServ framework appropriately routes such a request to the most lightly loaded system or device that can provide the requested service.

An entry, mapping each registered service to the number of clients being serviced by the service is maintained. When a new client makes a request for a

service, the MobiServ framework provides a connection to the least heavily loaded service if more than one instance of that service is available.

A low priority thread is initiated that keeps track of the number of connections made to different applications, the current memory constraints of the system providing the service, and a policy file that includes, among other information, the maximum number of connections accepted by the service.

4.7. Caching

To speed up the response time, caching is implemented by MobiServ using hash tables. The hashing is implemented using the Java hash table APIs. A (name, value) pair is stored in the cache. If a previously retrieved value is available in the cache, it is returned to the client. Otherwise, the result is recalculated and cached. The use of a hash table provides response in constant time. Advanced caching mechanisms were not a priority for the current project. However, a minimal caching scheme was implemented to illustrate feasibility and to speed up the response from the service to the mobile telephone.

4.8. Providing Service using WML

The Wireless Markup Language is used to communicate with the WAP microbrowser in the mobile telephone. Once, the transaction with the service via Java RMI is initiated, the results are sent via WML forms to the client and inputs from the user are requested, if needed. The WML inputs are again stripped of WML tags and the data is sent via Java RMI to the service with which the

transaction is initiated. A screen shot of the UP.SDK simulator showing an alert delivered to the mobile phone is provided in Figure 4.2.

Figure 4.2. Snapshot of client when an alert is delivered.

4.9. Summary

This chapter described issues related to supporting mobility at the application level. Leasing a service is used as a mechanism to dynamically assign resources. Handling remote events using event mailboxes supports disconnected operation. Load balancing provides a way to utilize duplicate services and reduce the load on a single service provider. The next chapter describes in client communication via WAP and optimizations in the design and implementation.

Chapter 5. Client Interaction

This chapter describes the interaction between the client, i.e., the mobile telephone, and Mobiserv. It describes servlet interaction, the client notification process , paging, browser cache control, cache validation, and optimization using message digests.

5.1. Communication with Servlets

The WAP-enabled mobile telephone contacts MobiServ, which is a Java servlet running on an Apache Web Server powered by a servlet engine [10]. The HttpServlet class provided by the servlet API is the most important class that is subclassed by all of the classes implemented by MobiServ.

5.1.1. Requests and Responses

Methods in the HttpServlet class that handle client requests take two arguments.

1. An HttpServletRequest object that encapsulates the data from the client.
2. An HttpServletResponse object that encapsulates the response to the client.

5.1.2. HttpServletRequest Objects

An HttpServletRequest object provides access to HTTP header data, such as any cookies found in the request and the HTTP method used by the request. The HttpServletRequest object is also used to obtain the arguments that the client sent as part of the request.

5.1.3. HttpServletResponse Objects

An HttpServletResponse object provides two ways of returning data to the user.

- The *getWriter()* method returns a Writer.

- The *getOutputStream()* method returns a ServletOutputStream.

5.2. Subscriber information

When the UP.Link Server makes an HTTP request to a WML service, it adds headers that provide information about the subscriber, the UP.Phone, and the UP.Link Server. These headers are converted by the Web server to environment variables that are retrieved by the MobiServ framework using the *getHeaders()* method in the set of Java Servlet API. The subscriber information includes subscriber ID, which is used to identify a subscriber with an IP address and whether or not the phone is ready to cache information or not.

5.3. Notification

5.3.1. Process

The process for notifying service availability to the client is detailed below.

1. The WML service posts the notification to the UP.Link Server. The notification specifies the subscriber ID of the UP.Phone to which the notification is directed and a time to live (TTL) value, specifying how long the UP.Link Server should attempt to deliver the notification.
2. The UP.Link Server issues the notification to the UP.Phone. The UP.Phone signals the user that an alert has arrived and adds it to the Inbox card in the WML deck.
3. When the user chooses the alert, the UP.Phone requests the specified URL.
4. The UP.Link Server relays the request to the WML based service.
5. The service returns the content for the URL.

6. The UP.Link Server relays the content to the UP.Phone.

5.3.2. Using the COM Notification library

The UP.SDK includes a Component Object Model [19] notification library for Microsoft Windows. Since the MobiServ application is written in Java, a COM–to–Java bridge is used for calling a COM object from a Java application. The COM Notification library contains the following classes.

- Ntfn3Client — non-secure notification class.
- Ntfn3SClient —secure notification class.

The notification is sent using the following process.

1. To send non-secure notifications, a NtfnClient object is created. To send secure or secure-preferred notifications, a NtfnSClient object is created.
2. If notifications are being sent to a UP.Link Server that does not use the standard notification port numbers, the *NtfnSetNonSecurePort()* or *NtfnSetSecurePort()* function is called to set the port number that should be used.
3. The *NtfnSetHost()* method is used to specify the domain of the UP.Link Server to which MobiServ sends notifications.
4. The *NtfnPush()*, *NtfnPostAlert()*, *NtfnPostCacheOp()*, *NtfnPostPrefetch()*, or *NtfnPostAlertAndInvalURL()* method is used to send push or pull notifications. These methods allow an expiration time, or TTL, to be set for the notifications. A value of 0 is specified to set the TTL to the maximum timeout value allowed by the UP.Link Server.

5. The *NtfnGetLastResult()* method is used to check the status of notifications that have just been sent by *MobiServ*. The *NtfnGetLastResult()* method returns an HTTP status code indicating whether the UP.Link Server accepted the notification. If the UP.Link Server did not receive or accept the notification, the *NtfnGetErrorDetail()* method is called to retrieve additional information about why the notification failed.
6. The *NtfnGetStatus()* method is used to check the status of notifications that have been accepted by the UP.Link Server. The *NtfnGetStatus()* method indicates whether the notification has been successfully sent to the user or is still pending.
7. The *NtfnClearPending()*, *NtfnDeleteAlert()*, *NtfnDeletePrefetch()*, and *NtfnRemoveAlertFromInbox()* methods are used to delete or cancel pending notifications that no longer need to be sent.

5.4. Paging

Paging another mobile telephone user can be relatively inexpensive if sent over the Internet. The cost of airtime to send a short message can be reduced by contacting the MobiServ application and providing a short message and the subscriber ID for the destination of the paging message. The MobiServ application then performs a WAP Push and sends the message to the specified user. This greatly reduces the cost involved in making a separate call to the user to send a short message.

5.5. Browser Cache Control

Caching at the mobile telephone can potentially improve the performance perceived by the user. When a user visits a URL, the UP.Phone caches the content. The next time that the user requests the URL, the UP.Phone retrieves it directly from cache instead of re-requesting it from the UP.Link Server. This is particularly beneficial in a relatively low data rate environment.

MobiServ controls when a URL expires from the cache by including a <meta> element in the deck header that specifies the TTL value. In some cases, however, a URL might need to be explicitly removed from the cache. For example, if a URL becomes obsolete before its TTL expires because a service is disconnected from the network or is no longer available. To remove a URL from the cache, MobiServ sends a cache operation to the device. Cache operations can instruct the mobile telephone to remove individual URLs or all URLs for the service from the cache. A cache operation can also be included in a digest along with WML content. To remove a URL asynchronously “behind the scenes,” i.e., without waiting for the user to request the digest, MobiServ sends a notification that contains a cache operation.

5.6. Cache Validation

A service may become unavailable for a period of time. The system must ensure that obsolete information is removed from the cache. A PUSH notification containing a cache operation removes the specified URL from the cache. The next time the user navigates to the URL, the UP.Phone requests it from the Web

server, ensuring that the URL is up to date. All cache entries for the current service or cache entries for the specified URL can be invalidated.

5.7. Optimization using Message Digests

MobiServ uses digests or multipart MIME formats [6] by sending one or more entities, i.e., WML decks and other content types, in a single message to the UP.Link Server. This optimizes use of the wireless network and makes the service appear to be more responsive to the user. If it can be known beforehand that the user will request multiple entities, MobiServ can send them all in a single response, instead of sending them individually with each user request. Because each HTTP request-response cycle involves a minimum time overhead, regardless of the amount of data transmitted, sending a single response is normally much more efficient than sending multiple responses. The size limit for a digest after the UP.Link Server compiles it is 1,492 bytes.

5.8. Summary

The UP.SDK COM library is used for WAP-based communication with the mobile telephone. WAP alerts are used to notify clients of services available in the vicinity. Cache validation is used to keep the mobile phone updated of the status of existing services. Message digests improve performance by reducing network traffic. The next chapter discusses the results obtained from conducting tests on the system, including the overall performance of the system.

Chapter 6. Results

This section describes the results obtained as a result of the work and the degree of conformance of the system to the original intent and design. The testing setup to test the system is also described along with the results obtained for a simple experiment. Certain observations based on the results are made, along with suggestions for a few possible applications for the system in a commercial environment.

6.1. Successful Testing of a Prototype Implementation

A prototype system consisting of MobiServ, sample Java services, Jini lookup server and the UP.SIMULATOR was successfully implemented and deployed. The prototype indicates that implementing a large-scale, commercial system may be feasible. The system was tested by deploying 25 clients and increasingly replicating the number of software services. The clients were implemented as multiple threads running on a personal computer and concurrently accessing MobiServ. The multiple instances of the services were replicated on two different machines for testing the system performance with service replication. There has been some research on analyzing WAP traffic versus HTTP traffic [24]. According to the results reported in [24], 95 percent of all WAP packets are less than 200 bytes in length. Based on these results, the system can be fine tuned for better performance, such as caching policies.

6.2. Example Screen Shots

Screen shots of the lookup server and the service when a request is made by a mobile telephone to Mobiserv are shown in Figures 6.1 and 6.2. Figure 6.1 shows the Jini lookup server ("reggie"), the HTTP server, and the RMI daemon responding to requests from MobiServ for a particular service.

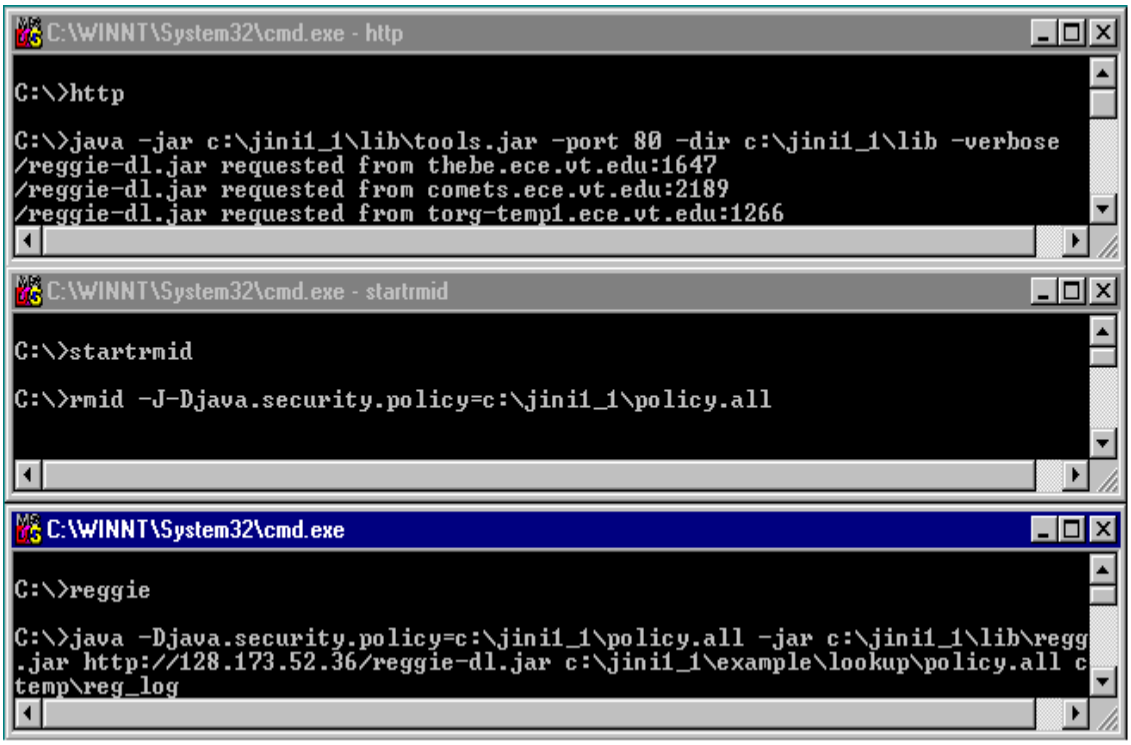


Figure 6.1. Snapshot of screens at the Jini lookup server.

The lookup server was setup on the machine thebe.ece.vt.edu. The service for PI computation was setup on comets.ece.vt.edu. Mobiserv was setup on torg-temp1.ece.vt.edu. The snapshot shows that the interface for the lookup server reggie-dl.jar is requested by the lookup server, the service, and by MobiServ, in order to communicate with the lookup server.

Figure 6.2 shows the service responding to a request from MobiServ. Mobiserv first requests the interface from the service. Once it gets the interface for the service and RMI stubs, the actual class is requested using a remote procedure call.

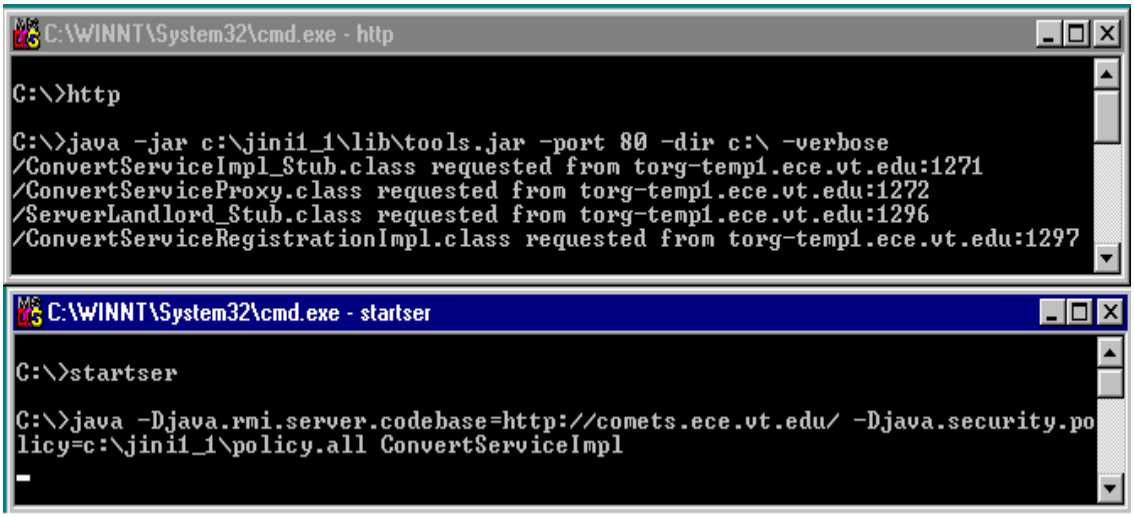


Figure 6.2. Snapshot of screens at the service.

6.3. Performance Improvement with Replication

A sample service to test the response time of the system was implemented and deployed. The service, which is not a practical service for a mobile environment but does require some processing time, computes the value of π. The response time of the system is the average waiting time for the mobile telephone user to get the result, i.e., the value of π, from the service. This result suggests the feasibility of the system on a larger scale, optimizations required, if any, and system bottlenecks. The service setup is as described in Section 6.2. The results were observed at the UP.SIMULATOR.

6.3.1. Improvement in Time

Figure 6.3 shows the effect of service replication on response time for the simple

service that computes the value of π. The graph indicates the response time in seconds (1 second = 1 unit) as a function of the number of instances of the service when the service is simultaneously accessed by 25 clients (threads).

We can see from the graph that initial improvement in response time due to service replication is drastic. After six replications the change in response time does not vary much with replication. This shows that the time taken by MobiServ to relay multiple requests to services based on their current loads increases as the number of requests increases. Hence, the performance improvement from replication is minimal after six replications. Also, the time taken to relay the response back to the UP.SIMULATOR increases with an increase in the number of requests.

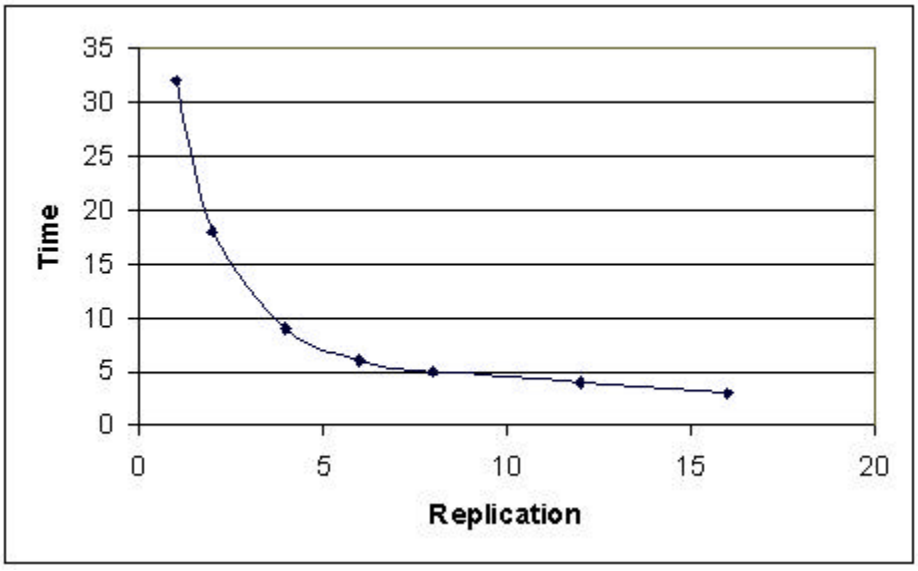


Figure 6.3. Improvement in time due to service replication.

6.3.2. Speedup

Speedup is defined as the ratio of response time using multiple service instances to the response time using a single service instance. The speedup observed for

response time data presented in Section 6.3.1 is plotted in Figure 6.4.

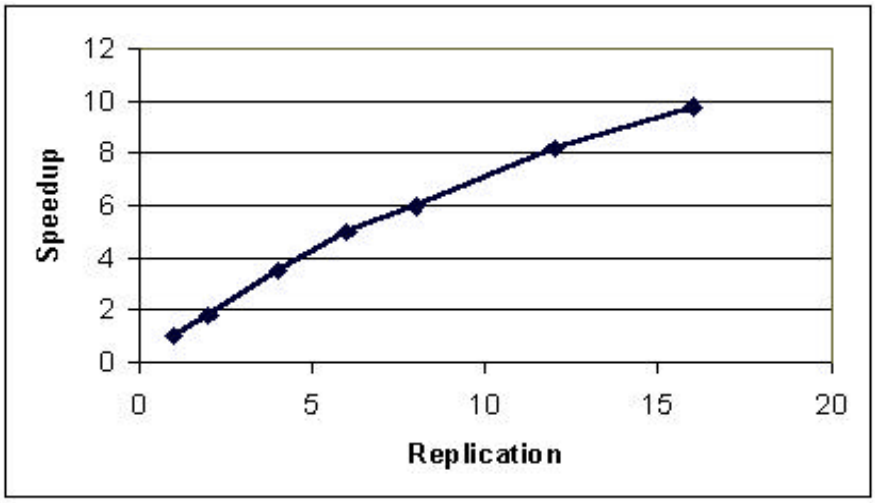


Figure 6.4. Speedup as a function of service instances.

The speedup shows that the improvement in time is significantly worse than linear for eight or more replications. This clearly indicates the increase in time taken by MobiServ in processing requests and responses after a certain number of requests. An improvement is seen with caching, but the particular set of data used in this particular test did not include repetitive requests. Hence, the effect of caching is not seen.

6.4. Summary

This chapter discussed the results obtained through limited testing of the system. The effect of service replication on the overall system response was observed. After a few replications, the system did not improve its response time drastically with replication due to the time taken in processing requests and responses at the MobiServ servlet.

Chapter 7. Conclusions

This chapter presents a summary of the work, conclusions drawn from the research, and suggestions for future work.

7.1. Summary of this Research

This research investigated the integration of distributed services with mobile environments. Current services being provided to mobile telephone users are relatively primitive. But, with wireless markets exploding, there is a need to provide high quality services and high functionality to mobile telephone users.

This work led to the development of an initial framework for providing distributed services to mobile devices at the application level. More generally, the framework is a step toward handling a broad range of mobility issues at the application level. The concept of code downloading provided by Jini was utilized for service discovery and communication with services. The initial framework mainly involves Java-based applications and can be extended to other services. The choice for communication with client devices was WAP. This introduced further issues related to mobile communication using WAP, including caching, performance, and communication protocols.

The framework was realized as a prototype system. The prototype implementation works well under reasonable loads. The system was not tested for commercial viability under excessive loads due to resource limitations. Overall the implementation conformed to the original design objectives of the

research and provides a prototype, which can be implemented for a larger, commercially viable system.

7.2. Conclusions

We can make several observations based on this work.

- Dynamic code downloading was found to be a promising solution for application-level distributed service provisioning. Movement of a proxy service, which holds only the interface for the service, rather than the state and code for the entire service saves a lot of network traffic and time.
- One of the concerns with this solution was that its interface is heavily based on the Java programming language. The acceptance of this solution on a large scale will depend on the acceptance of Java RMI as the standard for distributed applications.
- Mobility needs to be handled at the application level, as well as in service discovery and registration. Though the Jini environment provides a framework for service discovery and registration, it does not scale well with mobile environments.
- Deferred event handling, browser cache control, and paging are effective for handling mobility at the application level.
- Wireless Java, i.e., Java taken to the mobile telephone, can provide effective services to mobile telephones. MobiServ can reside on the mobile telephone, so the bottleneck involved in converting Java-based communication to WAP-based communication can be eliminated.

- The system developed can serve as a prototype for a commercial system to provide application layer service provisioning for mobile devices. The system would not necessarily need to be based on WAP.

7.3. Suggestions for Future Work

MobiServ can be integrated with newer technologies to make service provisioning easier and faster. For example, it can be integrated with Bluetooth [15] for providing service provisioning at the application level. It can also be integrated with J2ME [13] to speed up the process of service discovery and provisioning.

7.3.1. Integration with Bluetooth

Bluetooth is the latest wireless technology for personal area networks [15]. Any device equipped with a Bluetooth radio can become part of a Bluetooth “piconet” on the fly. If MobiServ is integrated with Bluetooth technology, it can potentially provide a solution for mobile and distributed resource management. As devices enter and leave a piconet, resources become available on the fly. The devices can use MobiServ for resource discovery and management and the system can provide a solution for mobility at the application layer. This would be an inexpensive operation and provides better service as it avoids the necessity for devices to be connected directly to the Internet and inexpensive Bluetooth radios can be used for communication.

7.3.2. Integration with J2ME and MIDP

Java 2 Micro Edition [13] is the latest entrant to wireless world. Combined with the Mobile Information Device Profile (MIDP) [12], it provides a limited Java

Virtual Machine (JVM) that can be installed on any wireless device. This provides a tremendous opportunity for integrating MobiServ directly with wireless Java. With the elimination of interaction with WAP, the service should become faster as MobiServ can directly interact with the JVM in the mobile telephone. As an alternative, MobiServ could exist as a low priority thread in the JVM in the mobile telephone itself and begin searching for a JINI lookup service as soon as the mobile telephone user enters the service area. This should further reduce the search and response time. An HTTP server Push is used for receiving notification of service availability. But, it would be an extremely costly application in terms of memory as the service proxy would be downloaded onto the mobile telephone which would take up a major part of the limited resources in the telephone.

References

[1] WAP Forum, “WAP Architecture Specifications,” April 1, 2001, available at http://www.wapforum.org/what/technical.htm.

[2] K. Arnold, “The Jini Architecture: Dynamic Services in a Flexible Network,” *Proc. of the 36th ACM/IEEE Design Automation Conf.*, 1999, pp. 157-162.

[3] J. Waldo, “The Jini Architecture for Network-Centric Computing,” *Communications of the ACM,* Vol. 42, No. 7, pp. 76-82, July 1999.

[4] Sun Microsystems, “JINI Specifications,” April 1, 2001, available at http://www.sun.com/jini.

[5] S. Oaks and H. Wong, *JINI in a Nutshell*, O’Reilly & Associates, Inc., 2000.

[6] Phone.com, “UP.SDK Technical Library,” April 1, 2001, available at http://developer.phone.com.

[7] Sun Microsystems, “Java Tutorial,” April 1, 2001, available at http://www.java.sun.com/docs/books/tutorial.

[8] JINI Developer Forum, http://developer.jini.org.

[9] Apache, “Apache Web Server Documentation,” April 1, 2001, available at http://httpd.apache.org.

[10] Apache, “Apache JServ Documentation,” April 1, 2001, available at http://java.apache.org/jserv/index.html.

[11] UMTS Forum, “UMTS Specifications,” April 1, 2001, available at http://www.umts-forum.org/.

[12] Sun Microsystems, “CLDC Specifications,” April 1, 2001, available at http://www.java.sun.com/products/cldc/.

[13] Sun Microsystems, “J2ME Specifications,” April 1, 2001, available at http://www.java.sun.com/products/cldc/.

[14] Sun Microsystems, “Java Remote Method Invocation (RMI),” April 1, 2001, available at http://java.sun.com/products/jdk/rmi/index.html.

[15] Bluetooth Special Interest Group, “Bluetooth Specifications,” April 1, 2001, available at http://www.bluetooth.com.

[16] Microsoft Corporation, “Universal Plug and Play Device Architecture,” April 1, 2001, available at http://www.upnp.org/resources.htm.

[17] Salutation Consortium, “Salutation Specifications,” April 1, 2001, available at http://www.salutation.org.

[18] WAP Forum, “WAP Push Architecture Specification,” April 1, 2001, available at http://www.wapforum.org/what/technical.htm.

[19] Microsoft Corporation, “COM specifications,” April 1, 2001, available at http://www.microsoft.com/com/resources/specs.asp.

[20] ASN.1 Information Site, “Introduction to ASN.1,” April 1, 2001, available at http://asn1.elibel.tm.fr/en/introduction/index.htm

[21] Internet Engineering Task Force, “ONC Remote Procedure Call,” April 1, 2001,available at http://www.ietf.org/html.charters/oncrpc-charter.html

[22] DoCoMo Inc., “I-mode characteristics,” April 1, 2001, available at http://www.nttdocomo.com/i/index.html

[23] IETF, “Simple Service Discovery Protocol,” April 1, 2001, available at http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt

[24] Thomas Kunz, “WAP Traffic: Description and comparison to WWW traffic,” April 1, 2001, available at http://kunz-pc.sce.carleton.ca/talks/MSWiM00.htm

Appendix A. Configuring Apache JServ on Win32 Systems

This appendix provides the necessary details required to configure the Jserv servlet engine on a Windows 95/98/2000 system. The purpose of the appendix is to provide those details that are required to configure MobiServ with Jserv.

The Apache JServ servlet engine is composed of two main parts, the servlet engine server application and the web server module (mod_jserv) [10]. This module is a communication layer that allows the web server to relay requests to the servlet engine.

A.1. Server Configuration

This section describes the procedure involved in installing the Jserv servlet engine, the procedure for installing the web server, and issues involved in troubleshooting.

A.1.1. Installing the Servlet Engine

The servlet engine is a 100 percent pure Java server application with its own configuration files. File jserv.properties is the main configuration file while zone.properties is an example servlet zone configuration file. A few directives in the jserv.properties file need to be edited to inform the JVM spawner, either the module or the wrapper, where to find things. The following properties should be edited.

wrapper.bin=@JAVA@ (the name of the JVM interpreter, absolute if not in PATH)

wrapper.classpath=@JSERV_CLASSES@ (the path to ApacheJServ.jar)

wrapper.classpath=@JSDK_CLASSES@ (the path to jsdk.jar)

root.properties=@ZONE_CONF@ (the path to ./conf/zone.properties)

log.file=@JSERV_LOG@ (the path to ./logs/jserv.log)

A.1.2. Installing the Web Server Module

The Apache web server communicates with the servlet engine using the Apache JServ module named ApacheModuleJServ.dll. This file must be copied under the \modules directory of the existing Apache installation. After the file has been copied to the modules directory, the configuration template file \conf\httpd.conf should be appended to the existing Apache configuration file (usually httpd.conf). This template helps to configure the module and the web server. The first thing to do is to uncomment the loadmodule directive to let Apache know about the new module, as shown below.

```
# Tell Apache on win32 to load the Apache JServ communication module
LoadModule jserv_module modules/ApacheModuleJServ.dll
```

After this, one must specify the servlet engine configuration file, usually jserv.properties, and the module log file with the following directives.

```
ApJServProperties <full path to ./conf/jserv.properties>
ApJServLogFile <full path to ./logs/jserv.module.log>
```

The log file will be created if it does not exist, or logs will be appended to an existing file.

A.1.3. Checking the Installation

To test the Apache JServ installation, we have to run the Apache Web Server and request the URL http://127.0.0.1/jserv/.

If the web server returns a “File not found” error code, the module is not properly installed or the jserv-status handler is disabled. Otherwise, we see the dynamic Apache JServ configuration pages that show the status of the servlet environment.

A.1.4. Troubleshooting

If we get an internal server error while connecting to the servlet engine, we should check the log files for problems.

If the *jserv.log* file was not created this means that Apache JServ did not start. Usually this is due to a “dirty” or broken *classpath* passed to the spawned virtual machine. One can check the *jserv.module.log* file and the Apache *error.log* file for clues to the problem. When the classpath is correct and Apache JServ starts, the *jserv.log* file is created.

A.2. Servlet Configuration

This section describes how to configure Java servlets.

A.2.1. Partitioning the Servlet Environment

Apache JServ has the ability to divide its execution environment into separate areas, called servlet zones, which act as independent virtual servlet engines. Apache JServ is designed in such a way that servlets are handled by the servlet zones, not by the servlet engine itself. For this reason, at least one servlet zone is needed. Let us suppose that we need two different servlet zones, one for production and one for development. Simply enough, we will call the first zone production and the second development. Let us also suppose that we already have the servlet engine up and running.

A.2.2. Creating the Servlet Zones

Apache JServ has one main configuration file, usually called jserv.properties, that is used for general information and one configuration file for each servlet zone that is managed by the servlet engine. For this reason, we create two copies of the sample zone configuration file, zone.properties, and, for clarity, we name them production.properties and development.properties. Then we edit the jserv.properties file and add these two simple directives:

```
# List of servlet zones Apache JServ manages
zones=production,development
# Configuration file for each servlet zone (one per servlet zone)
production.properties=/servlets/production/production.properties
development.properties=/servlets/development/development.properties
```

When Apache JServ starts up, it reads the list of servlet zones and looks for the specified zone configuration file. We should use absolute paths for these values since incorrect behavior with relative paths has been reported.

A.2.3. Configuring the Servlet Zone

Once the servlet engine knows about the servlet zones, we must tell the servlet zones where to look for its servlets. To do this, we simply add these directives to each of the zone configuration files, as illustrated below.

```
# The list of servlet repositories controlled by this servlet zone
repositories=/servlets/production/project1/
repositories=/servlets/production/project2.zip
repositories=/servlets/production/shared_servlets.jar
```

In this example, the production servlet zone has three servlet repositories: a directory (project1), a zip archive (project2.zip), and a Java archive

(shared_servlets.jar). These directives tell the servlet zone's classloader where to look for the requested servlet.

A.2.4. Mapping Servlets

Since servlets are Java classes and each servlet zone has its own classloader, servlets are accessed, mapped and named using their fully-qualified Java name, like in the classpath, using the package name followed by the class name. We must note that these mapping rules do not take into account the location on the file system as done by web servers. In fact, servlets located in different servlet repositories, but sharing the same package, are seen in the same directory by the classloader. For example, the servlet /servlets/production/project1/Hello.class belonging to the org.dummy.project1 Java package has a fully-qualified Java name of org.apache.project1.Hello, independently of its location. This absolute ordering based on package names alone is used to avoid class name collisions inside servlet zones. We also note that if two servlet zones share the same servlet repository, collisions are avoided by the use of different classloaders. These create different instances of the same servlets residing in their own address space and remove any collision problem. Also, the .class extension is always removed to form the fully qualified name to avoid conflicts with packages named class.

A.2.5. Servlet Aliases

Even if this class addressing is successful in ordering and avoiding collisions, it generates long and deep names for servlets. For this reason, alias facilities are provided by the servlet zones to simplify the naming process. To avoid long

names, aliased servlets may be used instead of the original fully-qualified ones. However, these two servlets are now seen as two separate entities and the two names generate two different instances of the same servlet. For example, to call the class org.dummy.project1.Hello simply by hello, we need to add this line to the proper zone configuration file.

```
servlet.hello.code=org.dummy.project1.Hello
```

A.2.6. Startup Servlets

During normal operation, a servlet is instantiated and initialized when the first request for that particular servlet is made. This guarantees lower memory consumption even if the first request is a little slower than successive requests. To avoid this overhead, or simply to have the servlet running as soon as the servlet engine is started, we should place our servlet in the startup servlets list for our servlet zone. To do this, we add the following line to our servlet zone configuration file.

```
servlets.startup=org.dummy.project1.Hello
```

Both fully-qualified servlet names (as in this case) or servlet aliases can be used.

A.2.7. Servlet Initialization Arguments (initArgs)

Like other forms of executable code, servlets need some initialization arguments. Every servlet may have an unlimited number of initArgs, placed in the servlet zone configuration file or in a separate file. For example, our servlet hello may need some initialization arguments such as message and color. This is done by adding the following lines to the servlet zone configuration file.

```
servlet.hello.initArgs=message="Hello world to everyone"
servlet.hello.initArgs=color=red
```

Appendix B. UP.SDK APIs

This appendix lists and briefly describes the UP.SDK APIs from Phone.com [6] that were used in the project. These APIs are based on Microsoft COM.

NtfnClearPending() deletes all pending notifications for a subscriber that match the prefix of a specified URL. NtfnClearPending() only deletes pending notifications for the WML service calling it. Notifications from other services are not affected.

NtfnDelete() deletes a pending alert or prefetch notification. When NtfnDelete() is called, the notification type must be specified.

NtfnGetStatus() gets the status for a specified alert or prefetch notification. When NtfnGetStatus() is called, the notification type must be specified.

NtfnPostAlert() posts a push notification containing an alert to the UP.Link Server. The UP.Link Server host must be set before calling NtfnPostAlert().

NtfnPostAlertAndInvalURL() is a convenience method that posts a push notification containing both an alert and a cache operation to the UP.Link Server. The cache operation removes the alert URL from the device's cache, ensuring that when the user requests the URL, the device will request it from the UP.Link Server instead of retrieving it from the cache.

NtfnPostCacheOp() uses the push channel to deliver a cache operation that removes a specified URL from the cache of a user's telephone.

NtfnPush() posts a notification to the UP.Link Server using the push channel. The only content types can be included in push notifications are:

- alerts,

- cache operations, and
- digests containing one or more alerts and/or cache operations.

The following parameters are passed to NtfnPush().

- **subs:** A null-terminated string containing the ID of the subscriber to whom the push is addressed. For example: 1234567-123_uplink.foo.com.
- **url:** A null-terminated string containing the notification URL. The URL must use the HTTP or HTTPS scheme.
- **TtlSeconds:** The notification's Time To Live (TTL) in seconds. If the UP.Link Server cannot deliver the notification within this period of time, it expires the notification. To specify the maximum TTL allowed by the UP.Link Server, we must specify 0. By default, the maximum TTL is 60 days. However, this period is configurable by the UP.Link Server operator.
- **ContentType:** A null-terminated string specifying the type of content to be sent in the notification. Three types of strings are currently supported:
  - application/x-up-alert, an alert,
  - application/x-up-cacheop, a cache operation, and
  - multipart/mixed, a digest containing alerts, cache operations, or both.
- **Entity:** The alert, cache operation, or digest to push to the subscriber.

NtfnSetHost() specifies the UP.Link Server host to which to send notifications. This method must be called before sending an alert or prefetch notification.

Appendix C. Starting the JINI Setup

This appendix gives the command line to start the Jini lookup server reggie.

1. Starting RMI Activation Daemon

rmid –J-Djava.security.policy=c:\policy.all

2. Starting ‘reggie’

java –Djava.security.policy=c:\policy.all –jar c:\jini1_1\lib\reggie.jar

http://128.173.52.36/reggie-dl.jar *c:\jini1_1\example\lookup\policy.all c:\temp\log*

3. Starting the HTTP Server

java –jar c:\jini1_1\lib\tools.jar –port 80 –dir c:\jini1_1\lib –verbose

Vita

Sunil Gopinath was born on March 15, 1977, in the city of Mysore, India. He received a Bachelor of Engineering degree in Computer Science from the University of Mysore in 1998. He worked with Philips for a year in their software development center in Bangalore, India. While completing his masters degree at Virginia Tech, he worked with the Public Service programs as a graduate assistant. He is interested in wireless networking, routing and other issues in networking.