Service Discovery: Access to Local Resources in a Nomadic Environment

Bruce Jacob
Advanced Computer Architecture Lab
EECS Department, University of Michigan
blj@eecs.umich.edu

ABSTRACT

We propose a standard level of support for nomadic computing in which a mobile client can move into unfamiliar environments and obtain services for which it has no corresponding client RPC stub. We call this activity service discovery and motivate its importance for mobile computing. This paper describes the general components of service discovery and points out where traditional paradigms of distributed computing fall short of its requirements. We suggest an architecture for an environment which uses a directory service based on the ODP trading function and a dynamic interface mechanism for ad hoc client-server interaction.

1 INTRODUCTION

The goal of nomadic computing transcends simply making one's environment portable; mobile users require the ability to communicate with local servers despite location and the ability to obtain local services despite statically defined service interfaces. To this end, we expect the portable computer or PDA to perform as a "universal interactor" [46]. There are three fundamental problems that need to be addressed to achieve a minimal level of interaction:

1. How does a mobile client locate an appropriate resource?
2. How does a mobile client interact with the (server providing access to the) resource?
3. What physical communication is used?

The first problem is largely solved. The second is not. The third has a simple short-term solution.

Location. A client can locate resources via a directory. There are several such directories, including the CCITT X.500 Directory [7], CORBA's Interface Repository [31], and the ODP trading function [22]. There are also several academic systems including Prospero [30] and Cygnus [8, 9, 39]. These systems provide methods that enable clients to locate services in a distributed environment. We have previously described the flexibility of the trading function in [23]. Whatever one's choice of directory, the point remains: the ability exists for a client to locate services based solely on descriptions of the service desired, rather than some specific piece of knowledge such as the name of the server or a unique ID referencing the service. One could even go as far as using the Web [3] to locate services, as many cataloging and webpage-directory services already exist.

Interaction. A client must be able to communicate intelligently with servers that it has never heard of before. This implies not only that the client does not *have* an RPC stub for the appropriate service interface, but that the client does not *need* an RPC stub. The client must determine how to interact with the server dynamically; the statically defined interfaces required in most RPC implementations are not adequate. The knowledge shared by the client and server must be reduced from a specific set of interfaces to a general interface description language in which higher-level data types are defined. For instance, the data type INTEGER is generally well-known. In a higher level language, the type DOCUMENT would also be well-known, with subtypes such as ASCII, POSTSCRIPT, ACROBAT, or HTML. Additionally, semantic knowledge will need to be attached to certain variable names or variable-name extensions. For instance, suppose the variable *Document* has a type of ASCII and the interface specifies another variable called *Document-Length*, of type INTEGER. The "-Length" keyword has semantic meaning attached to it, that the value of the variable *Document-Length* can be derived from the variable *Document*. An obvious corollary is that a variable called *Foobar-Length* without a corresponding *Foobar* is to be ignored.

Communication. The common denominator in virtually all systems world-wide is the Internet Protocol. Therefore, the protocol for communication should be IP. While we wait for a mobile IP standard to be adopted and supported, there is a short-term solution. Virtually every mobile client has a modem, and freeware or shareware versions of SLIP and PPP are widely available. We suggest that in the short term a mobile client can connect to local resources via traditional TCP/IP messaging over SLIP or PPP connections. The advantage is that SLIP and PPP are known interfaces with plenty of supported software, and the local environment can have userids with restricted capabilities prepared in advance for visitors. The burden of providing twisted pair connections is on the local environment.

For the purposes of this discussion, we assume the existence of solutions for the first and third problems. This paper describes a model for client-server interaction in a nomadic environment.

1.1 A Standard for Nomadic Computing

While mobile, we wish to be able to interact with the local environment in arbitrary ways. However, the current dominant paradigm for distributed computing, RPC, inhibits such free-form interaction. RPC requires static knowledge of a service and its interface. It is impossible, using present RPC implementations such as DCE or Sun RPC [32, 44], to discover and use new services unless they conform exactly to the interfaces expected by the client. A user desiring the use of a newly discovered service has no alternative but to roll up his or her shirtsleeves and start coding.

We propose a new standard to define a fundamental level of support for nomadism—that of *service discovery*. This is the ability for a client to discover services based on descriptive names, bind dynamically to the servers that offer them, and communicate intelligently with the servers to obtain the services offered. Any system which allows such ad hoc invocation of services supports service discovery.

Presented at the *OOPSLA'96 Workshop on Object Replication and Mobile Computing*. San Jose CA, October 1996.

The analogy is that of a telephone booth containing a copy of the local yellow pages, and marked with the well known international symbol for a payphone. A stranger to the area in search of donuts need not know the location of the nearest supplier of donuts; he must only be familiar with payphones, the payphone symbol, and how to use a yellow pages directory. From the directory he can look up the listings for all local suppliers of donuts and can call each one for donut prices, donut types, and directions.

The advantage of this scenario is that it reduces the amount of knowledge necessary for intelligent interactions in unfamiliar environments from *syntactic and semantic* to just *semantic*. The wandering donut consumer need not know any specifics about the location of donut stores, the prices of donuts, the manner of ordering donuts from the stores; he need only know the service that he wants (donuts), how to get to a directory (look for a payphone symbol), and how to ask a donut supplier for the specific information (look up 'donuts' in the directory and call the listed phone number). As long as he knows the semantics of the interaction (obtain donuts), he can learn the syntax (how to get there, what choices are available, how are they ordered, how much do they cost).

Similarly, a mobile client moving into an unfamiliar environment may wish to avail itself of local services, but should not need to know the interfaces beforehand. Knowing the semantics of the interaction should be sufficient; a mobile client should be able to learn the syntax of the interaction dynamically.

1.2 Service Discovery and RPC

The Remote Procedure Call paradigm (RPC) [5] has been largely responsible for the shape of distributed computing. Operating systems such as the Mach microkernel [15, 38], Amoeba [29, 45], Spring [17, 25], and V [10] range from being patterned after, to completely centered around RPC. Similarly, RPC is the central theme of most standards of distributed computing, including the Open Software Foundation's Distributed Computing Environment (OSF's DCE) and its Apollo predecessor [32], Sun's RPC [44], the Object Management Group's Common Object Request Broker Architecture (OMG's CORBA) [31], and the International Telecommunication Union's Open Distributed Processing recommendation (ITU's ODP) [21]. RPC provides a quick way to turn a monolithic application into a distributed application involving processes on several different machines. The syntax of a procedure call is expressed in a high-level language from which *stubs* (object code to be linked into client and server applications) are created for both the client side and the server side. The stubs handle the network connection as well as the marshalling and de-marshalling of the data packets. They make the network communication transparent to both client and server, so that the client and server applications can be written as if the procedure call is a typical stack-based function call. The server provides a function and the client merely calls it, unaware of the network communication occurring between the call and execution of the function. The stubs perform all communication functions once linked into the client and server.

Most RPC systems require an Interface Description Language (IDL) file to specify the service interface—including the parameters, their order, and their types. A special compiler called a *stub generator* uses this file to create the client and server stubs. Client applications must be written using the procedure interface defined in the IDL file. The stub is a prerequisite for communication between client and server, but in current RPC implementations the IDL file is a prerequisite for creating the stubs. Therefore the IDL file must be present before the remote procedure can be called. The human programmer must write the procedure call using the correct arguments in the correct order, must create and link the stubs—only then can communication happen. This is an unwieldy model; the problem is that too much of the knowledge is required statically; the syntactic information required by RPC at compile time could instead be determined at run time.

Ravishankar describes the problem in the preface to [9]. He argues that the benefits of an abstract model are undermined by binding too many concrete implementation details:

> The service notion has evolved into a dominant system structuring paradigm with the growth of distribution. This powerful abstraction serves both as an elegant alternative to the classical approaches to resource management as well as a practical means for ensuring scalability. The client/server model, which many current systems use, represents the prevalent implementation of this idea. Systems have largely tended to use the server interface that implements a service as a representation of the service itself.
>
> However, this approach promotes an artificial association between function and implementations. An interface is concrete. It already binds many specifics like syntax and may bind others, including protocols. It may even be specific to a server. A client interested in abstract functionality is therefore committed to detail ... an alternative and more abstract view is clearly required.

The problem with RPC, as well as virtually all distributed mechanisms in large-scale use, is that a client interested in abstract functionality is committed to knowing too much ahead of time. To use a service, the client program must have an explicit reference to a function call providing the desired service (including the correct parameters in the correct order), and often an explicit reference to a remote server. Systems like Prospero and Cygnus address the latter problem: they provide attribute-based lookup of services. They recognize that binding servers to services and their interfaces undercuts the flexibility of distributed systems, and instead allow clients to search for services using descriptions of the desired services.

However, half of the problem remains—that of implicitly binding clients to service interfaces. This is a real issue, recognized by the OMG and the ITU. Their CORBA and ODP standards address the issue, but since they are proposed standards they suggest functionality and say little about implementation. The CORBA standard advocates an intermediary between objects, called the Object Request Broker, that can translate between protocols. The responsibility of learning service interfaces thus shifts from the client object to the ORB. The ODP standard specifies a *trading function* [21, 22] in which server objects export service offers to traders and client object import offers. The recommendation however does not specify how clients and servers are to communicate.

An implementation similar to the ORB, called *RPC agents* [19] solves the interface problem by interposing itself between RPC clients and servers and translating between client and server protocols, thus freeing the client from the burden of learning many protocols or interfaces. The client object must still know what information to send to a server and what information to expect in return. This has the same weakness as the solution proposed by CORBA; reliance on an intermediary simply shifts the burden of learning new service interfaces one level higher and does not attack the heart of the problem. As newly created objects offer their services at the global level, they must either adhere to a small set of standard interfaces that the intermediaries know (an almost self-evidently naive requirement), or the intermediary must be periodically updated to support new interfaces. Clearly, this is only a short-term solution. We argue that it is the responsibility of the client to learn the new interfaces, and the responsibility of the servers to make learning them easy.

1.3 Service Discovery and Java

Sun's network programming language Java [16] is a possible candidate for providing service discovery. Whereas RPC is a passive service-invocation mechanism in that clients send servers requests and some amount of information to allow the server to carry out the request, Java supports an active service-invocation mechanism. Clients do not merely send requests to servers, they send the actual code for the server to execute.

This model of interaction is extremely flexible. A mobile client would not need to know any service interfaces, since it would define them itself. A client in search of a service would need only find a Java Virtual Machine willing to execute client code. However, Java has two problems: first, there have been many questions regarding the security of such a service invocation model [14]; second, Java does not solve the entire problem—a programmer still must know the interfaces of the local resources that a Java applet might need to use. Standardization of interfaces will solve the latter problem, but it is as unlikely to be realized on a global scale as standardization of RPC interfaces.

1.4 It *Will* Become a Problem

The rapid commercialization of the Internet and the National Information Infrastructure [47] imply that the future of general-purpose computing combines heterogeneity and ubiquitous computing services [36]. Future systems must not only support but *expect* a distributed environment similar to a shopping mall, where virtually all computing needs can be met. We have already witnessed the arrival of diverse user-level applications, from information services [3] to ordering pizza [35]. The de facto standard of distributed information services is the World-Wide Web [49], primarily because of the ease with which anyone can publish information.

In an environment where anyone can simply "hang a shingle" on the net and begin offering services, it is likely that everyone will. Anything one can imagine and far more that one cannot will be available. A discussion of pay-per-service lies beyond the scope of this paper, but it seems reasonable that there will be an abundance of free or virtually free services. Just as hotels and convention centers offer perks like free meals, discounts on local goods and services, and access to facilities like swimming pools and nautilus equipment, these types of establishments will begin to offer on-line services to patrons with portable computers and PDAs. Initial services could include such things as communications, information services, file systems, memory services (distributed virtual memory on demand; fast paging), CPU cycles (service requests might take the form of interpreted code), and display services (for wall-sized presentations, or to augment a machine with a small or non-existent screen).

In this environment, a mobile user will frequently encounter unfamiliar services, many of which the user will want to obtain. One should only need to know the description of a service to be able to use it; this implies an amount of adaptability in the client machine. Client systems must be able to take advantage of new services that suddenly become available, even if the designers of the system did not foresee such services, i.e. even if the client has no *stub* for such a service. The system must not be restricted by static interfaces as in typical RPC mechanisms; it must be possible for a client to invoke a service knowing only a descriptive service name, using interfaces determined dynamically.

1.5 Overview

This paper motivates the service discovery paradigm as a measure of support for nomadic environments, and suggests an architecture for an implementation. We propose the use of a high-level language to represent service interfaces, similar to most interface description languages like OSF's DCE Interface Description Language [32] or Xerox PARC's ILU Interface Specification Language [50]. Interfaces are defined by servers and learned by clients at the time of service invocation through a *service inquiry* on the part of a client. Servers are located through a directory service such as the X.500 Directory, CORBA's Interface Repository, or the ODP trading function.

A client interested in a service performs the following steps:

1. locate a directory object[1],
2. submit a service inquiry to the directory object to obtain a list of service names and providers,
3. select and connect to a service provider,
4. submit a service inquiry to the service provider to obtain the service interface description, and
5. interact with the service provider in the manner needed to execute the service.

The primary components are the service advertisement and the client-server interface.

Service advertisement. The service advertisement informs prospective clients of the details of the service and where to obtain it. A client uses this information to distinguish between servers providing similar services. For example, a "print" server will want to mention "postscript" somewhere in the advertisement if it accepts PostScript. The structure needs to be flexible enough to cover a large range of service types as well as heterogeneous machine types. An ASCII representation is one obvious choice.

Client-server interface. Once the client chooses a server, it will contact the server via its registered address and send a service inquiry, referencing the name of the service given in the service advertisement. The server responds with a description of the interface, looking much like a typical IDL file. It differs from an IDL file in that the data types and some of the variable names are well-known keywords. A client therefore can learn the syntax and low-level semantics of a procedure call at run time. There are many precedents for such a language—the Abstract Syntax Notation One (ASN.1) of X.400/X.500 [6, 7], the AI-oriented knowledge representation language KQML [13], or standard IDL languages such as in ILU, DCE, or CORBA.

The rest of this paper describes an environment that supports service discovery, illustrates in more detail the limitations of RPC, and discusses several issues in implementing service discovery.

2 WHAT NOMADIC COMPUTING *SHOULD* BE LIKE

You are mobile. You are at a conference. Your presentation is on your machine. You have been working on the slides continuously since you left town and have not printed them out. Perhaps you are expecting to be able to connect to a projector directly and give a virtual slide presentation through PowerPoint or something similar. Perhaps you expect to print out the slides when you are done and simply transfer to transparencies the usual way.

Whatever your choice of methods, you do not want your interaction with the local environment to be dependent on the details of a specific type of physical interface: this includes cable interfaces and disk formats. You do not want to have to access the local printer system by transferring your files to floppy then reading the floppy from a

1. "Object" is used in the "active object" sense, as in [12]. It is more or less equivalent to the word "server."

```
LA: Service-Inquiry = print
PR: Service-List = lpr-ascii, lpr-gif, lpr-postscript, lpr-ps, lpr-tex, lpr-text
    Service-Specification
        Service-Name = lpr-ascii, lpr-text
        Service-Description = prints out an ascii Document, ignores non-printing characters
        Service-Interface
            Service-Input = Document-Length:INTEGER, Document:ASCII
            Service-Output = Null
        Service-Keys = ascii, print, text
    Service-Specification
        Service-Name = lpr-tex
        Service-Description = translates and attempts to print out a tex file
        Service-Interface
            Service-Input = Document-Length:INTEGER, Document:ASCII
            Service-Output = Status:ERROR,SUCCESS
        Service-Keys = tex, latex, pictex, print
    Service-Specification
        Service-Name = lpr-gif
        Service-Description = interprets Document as a gif file
        Service-Interface
            Service-Input = Document-Length:INTEGER, Document:IMAGE
            Service-Output = Status:ERROR,SUCCESS
        Service-Keys = gif, print
    Service-Specification
        Service-Name = lpr-postscript, lpr-ps
        Service-Description = interprets Document as a postscript file
        Service-Interface
            Service-Input = Document-Length:INTEGER, Document:POSTSCRIPT
            Service-Output = Status:ERROR,SUCCESS
        Service-Keys = postscript, print, ps
```

Figure 1. .

machine on the local net, and you do not want to access the projection machinery via serial cable. Printing your document to a floppy and transferring to one of the local machines, hoping that they have the appropriate version of the program that you are using, is not a reliable option—although it is common practice today[2]. If you are a PowerBook user and the local environment is Unix, you are out of luck; most Unix workstations do not even have floppy drives. If you are a Windows fanatic and the locals are devoted Macintosh fans, you may find a compatible piece of software, but one would not be advised to risk it. It is likewise unacceptable to rely upon a physical serial connection to a projection machine; your laptop may have the wrong connector or no connector whatsoever.

Here are two illustrative scenarios:

- You want to print something out.
- You want to project your presentation virtually.

The next sections describe each of these scenarios in more detail.

2.1 Printing a Document

Here is the ideal scenario[3]. The local environment supports service discovery. You can connect to the local network using something akin to SLIP or PPP. The local administrator has created a number of guest login IDs, one for each of the conference attendees, with a randomly-generated pseudo-pronounceable password. At registration time you are given your login ID and password:

```
login: guest287
password: garmfrod
```

You ask at the check-in for the address of the local service trader as well as the location of the nearest PostScript printer. You are told to head to the modem pool for local connections and the printer.

You follow the signs to the modem pool and arrive in a room with a dozen twisted pair connections coming out of the wall. You plug into your modem (virtually every laptop user these days has a modem) and run your SLIP/PPP software. You login as guest287. You now have restricted access to the local network. Perhaps you can run Netscape. Perhaps you can telnet to your home environment and access your mail. At the very least, you can print out your slides. The sign on the wall says:

```
LOCAL SERVICE TRADER: trader@local.net
THIS PRINTER: printer@local.net
```

Since you have the address of the printer, you need not go through the trader. You type:

```
localaccess printer@local.net print
/u/blj/presentation.ps
```

The program localaccess talks to *printer@local.net* and attempts to invoke a print service for the file /u/blj/presentation.ps (it can distinguish /u/blj/presentation.ps as an argument of type FILE and not a keyword by the pathname prefix). The interaction looks something like the code in Figure 1 (LA is the localaccess program running on

2. Often known as “sneakernet.”

```
LA: Service-Inquiry = gif projector
AD: Match-List = remote-display, lpr-gif, gcc
    Inquiry-Match = gif+projector
        Service-Name = remote-display
        Service-Provider = display@local.net
        Service-Description = controls the remote projectors in the conference wing
        Service-Keys = gif, monitor, postscript, projector, remote display, screendump
    Inquiry-Match = gif
        Service-Name = lpr-gif
        Service-Description = interprets Document as a gif file
        Service-Provider = printer@local.net
        Service-Keys = gif, print
    Inquiry-Match = project
        Service-Name = gcc
        Service-Description = GNU project C and C++ compiler
        Service-Provider = compiler@local.net
        Service-Keys = C, C++, compiler
```

Figure 2. .

your mobile host, PR is the printer server on the local network, indents inserted for readability).

At this point, the localaccess application must choose between the various services offered by the print server. The document to be printed out is called /u/blj/presentation.ps and since ".ps" is known by the software to indicate a POSTSCRIPT file, the application searches for "postscript" and "ps" among the choices. Two services are found, *lpr-postscript* and *lpr-ps*, but since they share a specification this implies they are equivalent aliases for the same service. The *Document* variable has type POSTSCRIPT which the system knows to be the same type as the argument typed by the user (presentation.ps) so the system binds these two. The value of *Document-Length* is derived from the file. All variables are accounted for. The service invocation can commence without aid from the user. The system has one more exchange with the print server:

```
LA: Service-Request = lpr-postscript
    Document-Length = 34572
    Document
        Raw-Start
        <sends file data>
        Raw-End
PR: SUCCESS
```

The document prints out at the local printer and you head to your hotel room to make more changes.

This example is meant to illustrate the use of the *Service-Inquiry* and *Service-Request* mechanisms. Clearly, the printer server could simply offer one service called *print* with an additional argument called *Type* which could take on the values *ascii*, *gif*, *postscript*, *ps*, *tex*, or *text*. However this would not illustrate the use of the mechanism quite as well.

In this scenario, the mobile client has a descriptive notion of the service it wants (*print /u/blj/presentation.ps*) and knows the location of where to get it (*printer@local.net*); it has a semantic model of what it wants. What the client lacks is the syntactic model—the arguments and data types desired by the server. These are learned dynamically, allowing the client machine to use services of which the user had no previous knowledge.

2.2 Virtual Projection

The preceding example is simplified by the fact that the fictional system administrator had previously set up numerous restricted accounts so that the print server could assume any packets on the local network came from trusted machines. Also, the name and address of the print server were known beforehand. The scenario becomes more complicated when authentication and authorization are explicit and the server is unknown.

In this scenario, the administrator has just installed a projection server the morning of the conference, so the presenter does not know its address, and the server does not implicitly trust messages sent to it.

You have a program called *screendump* which will send a GIF version of whatever is on the screen to wherever you tell it, so that you can use something as simple as ghostscript or ghostview to show your slides. Failing that, you can always photocopy your slides and give your presentation manually. You type:

```
localaccess trader@local.net gif projector
```

Since the location of the server controlling the projector is unknown, you must go through the default server *trader@local.net*, to which all servers post service advertisements. This server performs the trading function [21, 22], allowing clients to connect to servers based solely on a description of the service they desire. The trader performs a lookup on whatever keyword/s is/are provided and returns a list of potential matches. Since neither "gif" nor "projector" are keywords or obvious files, they are treated as service descriptions. The localaccess system sends these to the trader to perform a lookup. The exchange looks like the code in Figure 2.

The local software determines that the first match—the one that hit on both keywords—is the best server to choose. It contacts the remote-display server, as shown in Figure 3

At this point, the localaccess system knows the syntax for the service invocation, but unlike the previous example where the name of the document was given by the user, the system does not have definitions for any of these variables. In this situation, it can only ask the user. The system pops up a dialogue box asking you to define the variable *Projector*. The projector has the name "Room 1200" stenciled on it, so you try that. The system pops up a dialogue box asking to define

3. To keep the discussion at a high enough level to be both informative and readable, we will use sendmail-style addresses to denote objects on the network, and gloss over the details of contacting the DNS for name resolution.

```
LA: Service-Inquiry = remote-display
RD: Service-Specification
    Service-Name = remote-display
    Service-Description = controls the remote projectors in the conference wing
    Service-Interface
        Service-Input = Projector:ASCII, Image:IMAGE, Image-Length:INTEGER,
               Authorization:auth@local.net:BINARY
        Service-Output = Null
    Service-Keys = gif, monitor, postscript, projector, remote display, screendump
```

Figure 3. .

```
LA: Service-Request = remote-display
    Projector = Room 1200
    Authorization:auth@local.net
        Raw-Start
        <sends token obtained from auth@local.net>
        Raw-End
    Image-Length = 875607
    Image
        Raw-Start
        <sends GIF data>
        Raw-End
```

Figure 4. .

the variable *Image*. You connect the output of your screendump program to this variable. The system knows how to derive *Image-Length* from *Image*, so it does not need to ask you about that variable. The keyword *Authorization* has semantic connotations: the system connects to the server *auth@local.net*, which asks for a username and password. You enter the guest ID and password you were given, which satisfies the authentication server. It returns a token to your localaccess system, which attaches it to the variable *Authorization:auth@local.net*.

Now the output of your screendump program goes directly to the remote-display server. Every time a new screen dump appears it sends a packet like the one shown in Figure 4.

Perhaps the *Authorization* token has a time expiration on it—several hours later it will be no good and a dialogue box will pop up if you continue to use the ticket, for example if you are giving a tutorial or a second presentation.

3 RPC FAILS TO SUPPORT SERVICE DISCOVERY

3.1 Binding to a Server

As has been discussed before, most RPC mechanisms require that the client know something of the server implementing the service before attempting to invoke the service. For example, before a client can connect to a server in DCE, the client must know the following things:

- The service id—a generated unique id from which the type of service rendered cannot be deduced.
  This: “002FD6B8-17F7-1B74-BFA9-02608C2C83B2” is an example. One cannot learn the service id except by obtaining a copy of the interface file.
- Either the authentication id of the server, or a shared secret between the client and the server—in this case a unique pathname in the CDS Namespace. No more than one client-server pair may rendezvous at a given CDS shared secret.
- The service interface—a collection of message structures to be passed between the client and server, each containing the service interface id, a message type id, and message-specific data.

Less restrictive but equally problematic, Sun’s RPC mechanism requires a service provider to be on the same host as its portmapper. This arrangement compels every host to offer the same set of services, or a client to know beforehand what host to contact. These types of requirements work fine in a closed system where the clients and servers are under the same administration and likely written by the same programmer. However, as systems get large it is unreasonable to expect clients to have global knowledge.

As described before, many systems including Prospero [30], Cygnus [9], CORBA [31], and ODP [21] have solutions for this problem. They allow a client to look up a service based on a general description. The lookup returns a (possibly empty) set of servers that can handle the service requested.

3.2 Using Service Interfaces

By definition, RPC involves a procedure call. The procedure call makes an abstraction of and hides the entire remote service. However, to use the service behind the procedure call it must be called, and more importantly it must be explicitly coded into the program. This is the behavior of *static interface* systems such as RPC; a service’s interface must be known not only before the transaction begins, but must have been known and used by the programmer at the time he/she wrote the client program. Both semantics and syntax are required at the time of compilation[4].

In a wide-area or mobile environment where one expects service requests to frequently cross administrative and organizational boundaries, the usefulness of such a paradigm breaks down. Client systems will frequently come in contact with newly discovered services requiring interfaces unknown to the client at compile time. Only a system supporting *dynamic interfaces* will support service discovery.

4 ISSUES IN IMPLEMENTING SERVICE DISCOVERY

These are a few of the outstanding issues about the nomadic environment described, and are the topics of our current research. They include security issues such as protecting a user from malicious servers, failure issues such as dealing with fault-tolerance and denial of service, and connectivity issues such as maintaining a level of service while moving through different organizational domains.

Security. The simplest method in a global environment makes every principal responsible for its own security. This is not a bad idea, as AT&T's experiences show that a server should not necessarily trust its portmapper [2, 11]. Similarly, it should not be up to a directory object to provide security for the client and server objects with which it conducts transactions. A potential security hole exists: if the client subsystem can respond with whatever information a server object requests, the enduser could become compromised. We do not want a client to believe any average passerby is a Kerberos server, so the client does not automatically authenticate for the enduser. There is other material the client subsystem may access that an enduser might not want to transmit to a receiver posing as a server object. The partial solution lets the user decide what information can and cannot be handed out, and to simply not make sensitive information available. However, there will doubtlessly be information appropriate to send to a server desiring some form of confirmation or authentication, but inappropriate to broadcast to objects posing as servers.

Failure. How should the client detect and respond to failure conditions, including malicious servers? There has been much research in the area of making distributed services fault-tolerant, much of it centered around reliable communications [4, 24, 27, 28, 34, 42]. One can detect when a server crashes in an open environment on the global scale much less easily than in a closed system. Further complicating the matter is the possibility of malicious servers that do not execute the service but respond positively, or poorly-written ones that perform the service but forget to respond.

Connectivity. Above and beyond the problem of packet routing [1, 33], mobile client systems might need to periodically verify whether they are still connected to the nearest directory object. There are two paradigms for managing nomadic, possibly disconnected, computers. It can be up to the system to locate the mobile party and pass on information when it suits the central system, as in the Xerox PARCTAB personal communication system [41, 43]. Alternatively, it can be up to the mobile client to check in with the system periodically, such as when the client changes location or reestablishes network connectivity, as in the Coda file system [26, 40] or Little Work mobile system [18, 20]. It seems most reasonable to place the responsibility of location determination on the client. One knows one's location in a cellular environment [48], and today's personal digital assistants (PDAs) can be outfitted with an optional global locator attachment. It would be simple if time-consuming to query a global directory for registered directory objects satisfying certain location criteria; a "map" of the user's surrounding area could be cached for future use. Global location is becoming a hot area; for example, Proximus provides a web-based map engine which will produce a map of any area, given an Internet domain name or US mailing address [37].

4. One can always argue that RPC could be extended to a more dynamic form by allowing servers to send out their IDL files in response to service interface inquiries, allowing clients to compile the requisite stubs at the last minute. This just-in-time RPC would solve the problem but would no longer be RPC; it is nothing like a procedure call. In a procedure call, the argument types and ordering are known in advance. By contrast, service discovery requires that systems adapt to the requirements of the environment—that they correctly handle arguments not explicitly prepared for or expected.

5 SUMMARY

Today, nomadic computing is restricted to merely making one's home environment portable; it is fundamentally non-interactive. Making nomadic computing fundamentally interactive has the potential to revolutionize the way portable computers are used, but current paradigms of distributed computing stand in the way. What is needed is the ability for a mobile client to interact in unfamiliar environments to obtain previously unheard-of services from previously unknown servers. This activity is called *service discovery*, the ability for a client program to inquire about services using textual descriptions and bind dynamically to servers providing the services discovered. A method for learning the service interfaces dynamically is a necessary component. This paper motivates the adoption of service discovery as a standard of support for nomadic computing, illustrates its use, and discusses several client-side implementation details.

References

[1] A. Aziz. "A scalable and efficient intra-domain tunneling mobile-IP scheme." *Computer Communication Review*, vol. 24, no. 1, January 1994.

[2] S. M. Bellovin. "There be Dragons." Tech. Rep., AT&T Bell Laboratories, August 1992. ftp://ftp.research.att.com/dist/internet_security/dragon.ps.

[3] T. Berners-Lee, R. Cailliau, and J.-F. Groff. "The World-Wide Web." *Computer Networks and ISDN Systems*, vol. 25, no. 4-5, November 1992.

[4] K. Birman and T. Joseph. "Reliable communication in the presence of failures." *ACM Transactions of Computer Systems*, vol. 5, no. 1, February 1987.

[5] A. D. Birrell and B. J. Nelson. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39–59, February 1984.

[6] CCITT. *Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1)*. International Telegraph and Telephone Consultative Committee, 1988.

[7] CCITT. *Recommendations X.500-X.521: The Directory*. International Telegraph and Telephone Consultative Committee, 1988.

[8] R. N. Chang and C. V. Ravishankar. "A service acquisition mechanism for the client/service model in Cygnus." Tech. Rep. CSE-TR-84-91, University of Michigan, 1991.

[9] R. N. Chang. *A Network Service Acquisition Mechanism for the Client/Service Model*. PhD thesis, University of Michigan, 1990.

[10] D. R. Cheriton. "The V distributed system." *Communications of the ACM*, vol. 31, no. 3, March 1988.

[11] B. Cheswick. "An evening with Berferd, in which a cracker is lured, endured, and studied." Tech. Rep., AT&T Bell Laboratories. ftp://ftp.research.att.com/dist/internet_security/berferd.ps.

[12] R. S. Chin and S. T. Chanson. "Distributed object-based programming systems." *ACM Computing Surveys*, vol. 31, no. 3, March 1988.

[13] DARPA Knowledge Sharing Initiative: External Interfaces Working Group. "Specification of the KQML agent-communication language." Tech. Rep., June 1993. ftp://ksl.stanford.edu/pub/knowledge-sharing/papers/kqml-spec.ps.

[14] D. Dean, E. W. Felten, and D. S. Wallach. "Java security: From HotJava to Netscape and beyond." In *Proc. IEEE Symposium on Security and Privacy*, Oakland CA, May 1996.

[15] R. P. Draves, M. B. Jones, and M. R. Thompson. "MIG–The Mach Interface Generator." Tech. Rep. (CMU unpublished report), Carnegie Mellon University, July 1989. ftp://mach.cs.cmu.edu/usr/mach/public/doc/unpublished/mig.ps.

[16] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Sun Microsystems, http://java.sun.com/java.sun.com/newdocs.html#dev, 1996.

[17] G. Hamilton, M. L. Powell, and J. G. Mitchell. “Subcontract: A flexible base for distributed programming.” In *Proc. Fourteenth ACM Symposium on Operating Systems Principles (SOSP-14)*, December 1993.

[18] P. Honeyman, L. Huston, J. Rees, and D. Bachman. “The Little Work project.” In *Proc. Third Workshop on Workstation Operating Systems*, April 1992.

[19] Y.-M. Huang. *Constructive Specification and Synthesis of Agents for Custom and Cross RPC*. PhD thesis, University of Michigan, 1994.

[20] L. B. Huston and P. Honeyman. “Disconnected operation for AFS.” In *Proc. USENIX Mobile & Location-Independent Computing Symposium*, August 1993.

[21] ITU. *Draft Recommendation X.903: Basic Reference Model of Open Distributed Processing*. International Telecommunication Union, 1992.

[22] ITU. *Draft Recommendation X.9tr: ODP Trading Function*. International Telecommunication Union, 1994.

[23] B. L. Jacob and T. N. Mudge. “The trading function in action.” In *Proc. SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[24] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. “An efficient reliable broadcast protocol.” *Operating Systems Review*, vol. 23, no. 4, pp. 5–19, October 1989.

[25] Y. A. Khalidi, M. Talluri, M. N. Nelson, and D. Williams. “Virtual memory support for multiple page sizes.” In *Proc. Fourth Workshop on Workstation Operating Systems*, October 1993, pp. 104–109.

[26] J. J. Kistler and M. Satyanarayanan. “Disconnected Operation in the Coda File System.” In *Proceedings of the 1991 Symposium on Operating System Principles*, October 1991, pp. 213–225.

[27] B. Liskov. “Distributed programming in Argus.” *Communications of the ACM*, vol. 31, no. 3, March 1988.

[28] J. E. B. Moss. “Nested transactions: An introduction.” *Concurrency Control and Reliability in Distributed Systems*, 1987.

[29] S. Mullender, G. v. Rossum, A. Tanenbaum, R. v. Renesse, and H. v. Staveren. “Amoeba: A distributed operating system for the 1990s.” *IEEE Computer*, vol. 23, no. 5, May 1990.

[30] B. C. Neuman, S. S. Augart, and S. Upasani. “Using Prospero to support integrated location-independent computing.” In *Proc. USENIX Mobile & Location-Independent Computing Symposium*, August 1993.

[31] OMG. *The Common Object Request Broker: Architecture and Specification, Rev 1.2*. Object Management Group, December 1993. OMG Document Number 93-12-43.

[32] OSF. *DCE Application Development Guide*. Open Software Foundation, 1991.

[33] C. Perkins, A. Miles, and D. Johnson. “IMHP: A mobile host protocol for the Internet.” *Computer Networks and ISDN Systems*, vol. 27, no. 3, December 1994.

[34] E. Pitoura and B. Bhargava. “Revising transaction concepts for mobile computing.” In *Proc. 1994 Workshop on Mobile Computing Systems and Applications*, December 1994.

[35] Pizza Hut. *http://www.pizzahut.com/*.

[36] L. Press. “Commercialization of the Internet.” *Communications of the ACM*, vol. 37, no. 11, pp. 17–21, November 1994.

[37] Proximus Corporation. *http://www.proximus.com/*.

[38] R. Rashid, A. Tevanian, M. Young, D. Young, R. Baron, D. Black, W. Bolosky, and J. Chew. “Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures.” *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 896–908, August 1988.

[39] C. V. Ravishankar and R. N. Chang. “An attribute-based service-request mechanism for heterogeneous distributed systems.” Tech. Rep. CSE-TR-08-88, University of Michigan, 1988.

[40] M. Satyanarayanan, J. J. Kistler, L. B. Mummert, M. R. Ebling, P. Kuman, and Q. Lu. “Experience with Disconnected Operation in a Mobile Computing Environment.” In *Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing*, August 1993.

[41] B. Schilit, N. Adams, and R. Want. “Context-aware computing applications.” In *Proc. 1994 Workshop on Mobile Computing Systems and Applications*, December 1994.

[42] A. Z. Spector, D. Daniels, D. Duchamp, J. L. Eppinger, and R. Pausch. “Distributed transactions for reliable systems.” In *Proc. Tenth ACM Symposium on Operating Systems Principles*, December 1985, pp. 127–146.

[43] M. Spreitzer and M. Theimer. “Providing location information in a ubiquitous computing environment.” In *Proc. Fourteenth ACM Symposium on Operating Systems Principles (SOSP-14)*, December 1993.

[44] Sun Microsystems. *Sun RPC man pages – rpc, rpcinfo, rpcgen, portmap*.

[45] A. S. Tanenbaum, R. v. Renesse, H. v. Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. v. Rossum. “Experiences with the Amoeba distributed operating system.” *Communications of the ACM*, vol. 33, no. 12, pp. 46–63, December 1990.

[46] M. Theimer, A. Demers, and B. Welch. “Operating system issues for PDAs.” In *Proc. Fourth Workshop on Workstation Operating Systems*, Napa CA, October 1993.

[47] M. K. Vernon, E. D. Lazowska, and S. D. Personick, Eds. *R&D for the NII: Technical Challenges*. Interuniversity Communications Council, Inc., February 1994.

[48] J. Z. Wang. “A fully distributed location registration strategy for universal personal communications systems.” *IEEE Journal on Selected Areas in Communication*, vol. 11, no. 6, August 1993.

[49] WWW Conference. “Special issue: Selected papers of the first World-Wide Web conference.” *Computer Networks and ISDN Systems*, vol. 27, no. 2, November 1994.

[50] Xerox. *Inter-Langauge Unification*. Xerox PARC, ftp://ftp.parc.xerox.com/pub/ilu/ilu.html.