Publishing RPC Services

by

Peter Szilagyi

B.S., Massachusetts Institute of Technology (1997)

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

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1998

© Massachusetts Institute of Technology 1998. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
November 25, 1997

Certified by ....
David K. Gifford
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Department Committee on Graduate Theses

Eng.

Publishing RPC Services

by

Peter Szilagyi

Submitted to the Department of Electrical Engineering and Computer Science
on November 25, 1997, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

We present a system for publishing RPC services on a computer network. Unlike existing systems, ours is designed to permit RPC services to be used across organizational boundaries, where the potential client programmers are unknown to the service programmers. Our purpose is to bring the World Wide Web's sense of community and ease of publication to program services.

Thesis Supervisor: David K. Gifford
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

I would like to thank my thesis advisor, Prof. David K. Gifford, for his superhuman patience with the glacial progress of my thesis work. I would also like to thank Dr. Olin Shivers, my post-thesis employer, for his encouragement and similar patience. Thanks to Mark Sheldon for discussions and references regarding the idea of interface discovery based on programming language types. Most of all, I would like to thank Nicole Burris for her ubiquitous support and tolerance of the evil thesis-mode Pete. Finally, I would like to thank my friend Jim Klossner, on who's AIX box I ran `uuidgen` and checked my IDL interface definitions.

Contents

1 **Introduction** 9
- 1.1 Definitions of RPC Publishing Terms 9
  - 1.1.1 RPC Service 9
  - 1.1.2 Server Computer 10
  - 1.1.3 Service Instance 10
  - 1.1.4 Instance Publication 11
  - 1.1.5 Instance Registry 11
  - 1.1.6 Interface Discovery 12
- 1.2 Why Publish RPC Services? 12
  - 1.2.1 No Service Exists for your Application 13
  - 1.2.2 You Want to Simplify a Service 13
  - 1.2.3 Adding Functionality to an Existing Service 16
  - 1.2.4 Your Interface Changes Frequently 16
  - 1.2.5 Community Development can be Fast and Modular 16
  - 1.2.6 RPC has Advantages over Data Protocols 17
  - 1.2.7 Summary 17
- 1.3 How We Approximate Publishing RPC Services Now 17
  - 1.3.1 Downloading Software 18
  - 1.3.2 Packaged Software 18
  - 1.3.3 Java Applets 19
  - 1.3.4 Component Object Technology 20
- 1.4 How We *Should* Publish RPC Services 21
  - 1.4.1 Support Network Interface Discovery 22
  - 1.4.2 Automate Downloading and Installation 22
  - 1.4.3 Provide Network Names for Services 22
  - 1.4.4 Check Interface Usage Mechanically 22
  - 1.4.5 Reuse Existing Interfaces through Subtyping and Supertyping 23
  - 1.4.6 The Component Technologies are Available Now 23
  - 1.4.7 The Component Technologies have not been Combined 24
  - 1.4.8 Summary 24
- 1.5 Organization 25

2 An Example: A Student Information Service **27**
2.1 Meet Jane Fromaine, Project Programmer . . . 27
2.2 See Jane Read the Online Documentation . . . 27
2.3 See Jane Discover the Bursar's and Registrar's Services . . . 29
2.3.1 How does a Browser Display Published RPC Instances? . . . 30
2.4 See Jane Click to Generate Client Headers and Stubs . . . 30
2.5 See Jane Prepare her Project Makefile . . . 32
2.6 See Jane Write and Compile her Program . . . 33

3 The Design of an RPC Publishing System **35**
3.1 Environment and Solution Components . . . 35
3.1.1 The Web is Used for Naming and Discovery . . . 35
3.1.2 DCE RPC is Used at Runtime . . . 36
3.2 Requirements for RPC Publishing . . . 37
3.2.1 Communities Develop RPC Services . . . 37
3.2.2 A Network's Services are Browseable and Searchable . . . 37
3.2.3 Programs can Browse and Search . . . 38
3.2.4 Components Have Network Names . . . 38
3.2.5 No Runtime Overhead . . . 38
3.2.6 Trivial Tasks are Automated . . . 39
3.2.7 The System Works with the Existing Web . . . 39
3.3 Satisfying the Requirements of RPC Publishing . . . 39
3.3.1 A Single Namespace Binds Human and Program Services . . . 39
3.4 The Architecture of RPC Publishing . . . 40
3.4.1 Instances may be Specific RPC Objects . . . 43
3.4.2 Servers are Essentially `dcerpc:` URL's . . . 46
3.4.3 Services Include IDL Specifications . . . 48
3.4.4 The Registry Lists a Server's Published Instances . . . 50
3.4.5 An Example of the RPC Publishing System URL's . . . 51
3.4.6 Discovery may use Extensions to the Web . . . 53
3.4.7 Ordinary DCE RPC Binding Occurs at Runtime . . . 56
3.5 Alternative Designs . . . 56
3.5.1 Making Java Applets Publishable . . . 58
3.5.2 Using a Local Proxy to Present Services . . . 58
3.5.3 Integrate a CORBA ORB with the Web . . . 60
3.5.4 Use Java RMI as an RPC System . . . 60
3.5.5 Use Microsoft's DCOM as an RPC System . . . 60

4 The Future of RPC Publishing **63**
4.1 Applications of RPC Publishing . . . 63
4.1.1 Collecting Stock Quotes from the Internet . . . 63
4.1.2 Searching the Web . . . 64
4.2 Contribution . . . 65
4.3 Further Work . . . 65
4.3.1 An Implementation Would be Nice . . . 65
4.3.2 A Java Implementation Would be Nicer . . . 65
4.3.3 Declaring Interfaces as Supertypes . . . 66
4.3.4 Interface Discovery Based on Programming Language Types . . . 66

List of Figures

2-1 Hypothetical RPC Browser Displaying Search Results 31

3-1 The Architecture of RPC Publishing 41
3-2 A Generic RPC Service Instance 42
3-3 An RPC Publishing System Server 45
3-4 A Generic RPC Service 49
3-5 Registry IDL Specification 52
3-6 A Programmer Discovering a Service 54
3-7 A Program Binding to a Service 57
3-8 A Possible `DocumentedApplet` Class 58

Chapter 1

Introduction

In this chapter we define publication of RPC services and we motivate the need for it. We then consider how machine-oriented services are distributed on today's Internet and how current practice differs from our ideal. Finally, we organize the remainder of this thesis.

1.1 Definitions of RPC Publishing Terms

In this section we clarify some common terms in the context of RPC services. We also define some new terms specific to RPC publishing.

1.1.1 RPC Service

A *service* is simply some functionality provided by a computer network. A *program service* is a service defined with enough precision to be useful to computer programs. In general, this does not imply a particular RPC protocol. An *RPC service* is, as usual, a program service that uses a recognized RPC protocol. We will focus on *DCE RPC services* [12] in this thesis.

The old `finger` service [13], for example, is not really even a *program* service because its output data format is not sufficiently structured. The current `finger` service [27], on the other hand, is much more structured, and might therefore be called a program service.

Services that are defined in terms of explicit RPC protocols are RPC services. For example, NFSv3 is usually implemented as a Sun RPC service [5, 24, 25, 23]. In the gray area lie services that

do not use any existing RPC system, but use a precise, program-oriented, non-human-readable protocol. For example, the Internet `time` protocol [18] provides the number of seconds since midnight on January 1, 1900 in binary form.

We sometimes call an RPC service a *service type* or a *service class* to distinguish it from a *service instance*, described below. Further, we extend the DCE RPC terminology and call an RPC service an *RPC interface* to indicate that a published RPC service has essentially the same functionality as an IDL interface specification.

1.1.2 Server Computer

A *server computer* is a network host that provides a service or services. The term *server* is sometimes used to refer to software, running on a server computer, that provides a service. (That is, a *server process* or *server program*.) We will use the term *server* for server computers only.

Servers can be named physically (e.g., by a DNS domain name) or logically (e.g., by a service name). For example, DCE makes it possible to bind to a server using a symbolic name that does not physically identify the server. In fact, in the case of automatic binding, the DCE RPC runtime may be able to try to bind to different physical server computers during a single RPC call, transparently [12, page 106].

Thus, a *server* is a computer that may be named in a variety of ways, including a DNS domain name or IP address. In fact, the server might not be identified uniquely by the RPC publishing system.

1.1.3 Service Instance

A *service instance* is a particular implementation of an RPC service. A service instance is provided by *server process* running on a server or a latent *server program* that will be run as necessary (in the fashion of the popular Internet superserver [19, `inetd`]). A single server may support multiple instances of an RPC service.

We use the *service instance* terminology to distinguish an instance of a service from the service itself. For example, `finger` is a well-known service available on TCP port 79. There is an *instance*

of the `finger` service running on the server `psrg.lcs.mit.edu`.[1]

1.1.4 Instance Publication

We define the *publication of an RPC service instance* as the distribution of all information needed to use that instance, including human-readable documentation and machine-readable specifications. An instance publication includes pointers to the service type, server computer, and service instance.

The form of an instance name varies depending on the underlying RPC mechanism. For an IP-based service, we might use a DNS name for the server and an IP port for both the service and instance, like `psrg.lcs.mit.edu:79` for the `finger` service. For DCE RPC, we would use the Global Directory Service (GDS) [12] to name services, as in `/.../lcs.mit.edu/subsys/registry`.

An RPC publication includes testing and debugging aids as well as locative information and documentation, especially when the service instance manipulates precious, persistent state. These testing and debugging aids might include test scaffolding code, an interface to instrumentation on the server, etc.

1.1.5 Instance Registry

In our proposed system, the *instance registry* is a well-known service instance provided by every server. The registry documents the server's capabilities, allowing programmers to discover instances on the server using browsers and search tools. Our instance registry is built on the registration mechanism of the underlying RPC system.

Sun (ONC) RPC, for example, uses a *portmapper*, which is a well-known UDP service (`sunrpc`) on port 111. This portmapper is also a well-known RPC service (`portmapper`) with the RPC program number 100000. The portmapper is a registry for server processes that maps logical service names (program numbers) to physical connection points (IP ports). In our terminology, this would be a *service* registry, not an *instance* registry.

DCE RPC provides a registry that can map service names (UUID's) to generalized physical connection points called *endpoints*. Hence, the registry mechanism is called the *endpoint mapper*.

[1] Although `finger` is not an RPC service, we hope it illustrates the point.

However, the DCE endpoint mapper can also map *object* names (also UUID's) to their endpoints.[2]

We associate DCE objects with RPC service instances in our proposed system. Therefore, we use the DCE endpoint mapper as an RPC instance registry. We specify an RPC service instance by specifying its server computer, service type, and a DCE object UUID for the endpoint mapper.

1.1.6 Interface Discovery

Lastly, we define *interface discovery* as the interactive process of finding published RPC services and instances. Both people and programs undertake interface discovery to find services and instances, although programs can only browse and search in limited ways. Interface discovery can proceed from two perspectives:

- The seeker may *search* for a particular interface or, more generally, some kind of functionality.
- The seeker may *browse* a server or network to see what RPC services it provides.

A human seeker's goal might be very general. For example:

> Where can I find an RPC service for getting stock quotes, and what instances of that service are available?

On the other hand, a program must be much more specific:

> Does `http://stockquote.com` provide an instance of the `http://nasdaq.com/quote/service/specification.idl` service?

Thus, a wide variety of exploration activities fall under our definition of interface discovery, ranging from searching to browsing and from general to precise.

1.2 Why Publish RPC Services?

In this section we provide examples of why one might publish precisely specified RPC services. Publishing an RPC service allows many people to use it, even if it hasn't been widely standardized or implemented.

[2] DCE server programs can use the runtime function `rpc_ep_register` to associate themselves with both a service *and* any number of objects [12, pages 114, 126]. Using the endpoint mapper, clients can locate those specific objects, if they wish.

1.2.1 No Service Exists for your Application

One reason to publish an RPC service is the absence of a well-known standard that meets your needs. If there is no standard for some specific RPC functionality that you need, then it falls on you and your community to *find* a standard.

As far as we know, for example, there is no *RPC* standard for exporting real-time meteorological information.[3] So, if you wanted to make continuous weather maps from the data collected by university weather stations, you might build an interface to your weather station and publish the interface as a first cut. Other weather stations could then implement your interface and give you feedback on how to improve it. Before long, a workable standard might evolve.

1.2.2 You Want to Simplify a Service

Another possibility is that your application needs only *some* of the functionality of a well-known service. Therefore, your implementation supports only a subset of the operations of an existing service, or your implementation does not implement the existing operations completely. In this case, you may want to be precise about what your service provides to avoid confusion and incompatibilities.

In programming language terminology, let us associate services with *types* and instances with *values*. We can think of the operations in a service as *methods*. Every instance of a service implements the operations of the service, so every value in the corresponding type is subject to the methods of the type.

When you provide a simplified service of the kind described above, you would like your clients to be able to call on the *more general* standard service as well. After all, the standard service implements all the operations your service provides, and then some. Thus, any instance that supports the standard service will support your service, at least conceptually. (But see below.)

If we call the type of the standard interface S, and we call the type corresponding to your interface I, then we say that S *is more general than* I, or S *is a subtype of* I. (The terminology is a little

[3] Thanks to my thesis advisor, Prof. Gifford, for this idea.

bit confusing, but please read on.) This relationship is denoted

$$S \sqsubseteq I$$

to indicate that S *is smaller than* I, or S *contains fewer values than* I. A value of type S can be used wherever a value of type I is expected, because S provides all the methods of I, and then some. The reverse is not true; there may exist values of type I that are not subject to all methods of type S. After all, I specifies a subset of the methods of S.

Thus, despite the confusing terminology, I *is a supertype of* S. Put another way, your restricted interface I provides at least as many instances as the existing service S, since every instance of S is automatically an instance of I.

This relationship between your interface and the existing interface presents some practical and theoretical problems.

- Popular RPC systems do not include a notion of subtyping on interface specifications; it is a programming language issue. For example, DCE RPC has no hierarchy of types at all; every interface is associated with an unstructured UUID, and a server either does or does not implement an interface. This leaves you with three alternatives, none of which is appealing.

  1. Specify and implement your superinterface separately from the well-known standard. Provide documentation that explains that your interface is *conceptually* a supertype of the standard interface, and provide references to the standard interface.

     If you do this, of course, clients written to use your interface will *not* automatically work with the existing service upon which it is based (S). Conceptually, they should work, because your interface is *less defined*, i.e., it is a supertype of S.

  2. Implement the subinterface, i.e., the well-known standard. For those operations not in the superinterface, provide stubs that return some kind of error indication, preferably an `UNIMPLEMENTED` exception, if your RPC system supports it. Again, document this behavior as well as you can.

     This solution has the disadvantage of being *type unsafe* in the sense that clients of the existing interface S will not know that they cannot use servers that implement your

interface I. There is, therefore, the potential for error, especially if a client erroneously depends on some semantic property of S that doesn't apply to I.

3. Get the cooperation of the specifiers and implementors of the standard. Specifically, get them to agree that your interface is a valid and useful supertype of the standard interface. They will specify that all implementations of the standard interface also implement your interface. Because your interface is a supertype of the standard one, this will be relatively little work.

   This alternative clearly does not scale, since the standard is, by assumption, much larger in scope than your interface. However, this is the only alternative that will make clients of I work with servers of the more specific interface S while preventing clients of S from trying to use servers of the more general interface I.

Thus, using an existing RPC system is technically infeasible if you want to support the idea of creating new interfaces by simplifying existing interfaces. Nevertheless, we will use DCE RPC for practical reasons.

- Even popular programming languages do not support the declaration of supertypes of existing types. Java, for example, includes all the supertypes of an interface in the interface declaration. This means that, once again, you will need the cooperation of the standard interface publisher to declare a superinterface of an existing interface.

  Thus, even if you used Java and Java RMI [8] as the basis for your system, you would need to cooperation of the publisher of the subtype.

These problems, which make true supertyping seem impossible to provide with existing technology, suggest that we leave the problem to further work. See Section 4.3 for this and other interesting problems that could be explored by further work.

As a practical matter, programmers do not seem to miss supertyping, while they use *subtyping* heavily. The next section describes the pressing need for subtyping when reusing existing interfaces.

1.2.3 Adding Functionality to an Existing Service

Similarly, your application may *extend* a standard by providing additional functionality. The comments above apply to this section as follows.

- Existing RPC systems still do not support subtyping on interfaces, but now it is not so important, because you are in control. You can specify the new interface as a subtype of the existing interface, and you can insist that all implementations of your interface also implement the well-known standard interface.
- Programming language solutions will work as long as the programming language is as expressive as Java. You can declare your new interface to be a subtype of some existing interfaces.

As an example, suppose you wanted to extend NFS with virtual file operations. That is, your NFS server will implement some `read` and `write` calls by proxy, and you want to expose that to clients in a controlled way. This functionality could by used to forward clients from one NFS server to another, providing some of the functionality of AMD [17] in the NFS server.

Your extended NFS interface would include all the normal NFS calls. In addition, it would include calls to determine if a particular file was forwarded and from where. Naïve NFS clients would work just fine, and clients that knew about your extended interface would be able to snap forwarding pointers for improved efficiency.

1.2.4 Your Interface Changes Frequently

As the last general situation in which you might publish an RPC service, consider an experimental service that changes frequently. Such a service is clearly not a candidate for standardization. Nevertheless, it is important for an experimental system to reach a wide audience for feedback. In this situation, precisely defined RPC interfaces can flag changes in the interface automatically, catching and updating out-of-date clients.

1.2.5 Community Development can be Fast and Modular

Although distributed development is hard and has many disadvantages, it is certainly the case that

- massively parallel development can sometimes be made fast, and
- different developers maintain modular pieces of large systems.

Witness the impressive development of free software systems that use distributed development, such as Linux, Emacs, and LaTeX.

1.2.6 RPC has Advantages over Data Protocols

The advantages and disadvantages of the RPC paradigm compared to data-oriented protocols are well-known.[12, page 58]. The advantages include

- the familiar, synchronous, easily understood and used procedure call syntax and semantics,
- automatic data encoding, translation, and decoding, i.e., marshaling and unmarshaling,
- tight adherence to the client-server model's request-response discipline, and
- automatic binding to servers, at least in the case of DCE RPC.

These advantages reflect the large investment of techniques and tricks in the RPC runtime system, which is reused by all RPC programs.

1.2.7 Summary

In summary, there are many reasons to publish RPC services. They are good for developing community standards, they prevent confusion about exactly what subset or extension of a well-known standard you are using, and they keep clients synchronized with rapidly changing interfaces. Further, the general advantages of publication and high-level RPC protocols hold, including community development and reuse of RPC techniques, respectively. Generally speaking, published RPC services have all the usual advantages of precisely specified interfaces [16].

1.3 How We Approximate Publishing RPC Services Now

In this section we describe the problems with contemporary software distribution mechanisms in the context of publishing RPC services. These forms of distribution include simple downloading,

packaged software, applets, and other component software.

1.3.1 Downloading Software

Downloading software is uncertain and error-prone, even when it can provide mechanically checkable interface specifications. For example, take Inquir [4], an RPC-like service distributed via FTP.

Inquir is a centralized database of user information provided by the Laboratory for Computer Science (LCS) for the LCS community. LCS also provides a companion TCP service that allows users to modify their Inquir entries. LCS distributes Inquir in the form of documentation and client programs that implement both RPC and user interfaces to the Inquir service. These are published on `ftp.lcs.mit.edu`.

The problems with Inquir are that (1) it requires human intervention to download, compile, and install and (2) its interface is specified informally, not in a machine-readable language. Both of these problems typify Unix free-software distributions. In particular, Unix is plagued by its great strength: programs operate on unstructured strings, with great flexibility, but almost no type checking.

Manual operations are simple for human clients, but programs require absolute precision from a publishing system. We seek to automate downloading, compilation, and installation.

1.3.2 Packaged Software

Software packaging systems like RPM [1] provide the automatic downloading, compilation, and installation management we need, but they do not provide interface specifications and enforcement.

RPM is the Red Hat Linux Package Manager. In addition to automating the installation process for Linux software, RPM distinguishes documentation files and configuration files from other files inside packages. Thus, RPM supports extracting and browsing documentation mechanically. This is certainly a step up from downloading and installing software manually.

However, it is up to particular packages to provide machine-checkable interfaces. If a package contains ordinary Unix programs, we run into the same problem we had with Inquir: Unix programs cannot be specified conveniently because such programs have unstructured interfaces. Even

if an RPM package contains subroutine libraries written in a language that *does* support interfaces, RPM does not provide explicit support for interface specifications.[4]

Thus, RPM and its kin in the Windows world are not designed to enforce programmatic interfaces. In general, they are designed for installing user-oriented software rather than RPC services.

1.3.3 Java Applets

Applet programs are a special case of *component software*, which is simply software designed to plug into a particular environment. Many applet environments have been provided, but the most popular one by far is the Java applet API [6]. Java applets have several advantages over packaged software.

- Java applet support is already incorporated into popular Web browsers. Simply browsing a Web page containing an applet serves to download, install, and run a piece of software automatically. These steps, although trivial for a human, are nevertheless tedious; their automation is an essential part of transparent RPC service publishing.
- The Java programming language [10] incorporates precise specifications in the forms of *interfaces* and *classes*. Because a Java applet is also a Java class, it has the capability to export arbitrary Java language interfaces.[5] These interfaces can be accessed with the Java reflection API [7].
- Java programs have access to a developing RPC service called Java Remote Method Invocation (JRMI) [8]. This RPC protocol could serve to support an RPC publishing system.

Thus, Java applets solve several problems associated with downloading and packaged software. Java applet systems automate more tedious, transport-level work, they realize precise specifications, and they provide an RPC system.

However, several problems remain in the Java world, which affect the usefulness of Java applets as a mechanism for RPC service publishing:

[4] We would have to add some precise conventions to RPM to export interfaces from packages.
[5] A Java class can implement any number of interfaces.

- Although Java has the potential for expressing RPC interface specifications, applets are user interface components, not program modules. In particular, applets do not *necessarily* export interfaces that would be useful as RPC interface specifications, nor do they provide detailed documentation on how to use the underlying protocol. Therefore, we cannot use Java applets, by themselves, as an RPC service and instance publishing mechanism.

- Because Java applets are intended to run in secure environments on fundamentally insecure platforms, they are crippled in several important ways, and yet there is still some question about how secure the Java environment really is [26]. In particular, Java applets cannot load other Java classes or communicate over the network, except with their servers.

- We have no way to name Java interfaces on the network. This capability is essential so that RPC service and interface descriptions can refer to one another. For our purposes, this is a limitation in the language (because of the syntax of the `import` declaration) and in the virtual machine (because it is not specified to handle URL's or other names for packages).

- An additional but minor problem is that Java applets do not explicitly include human-readable documentation, which is an essential part of publishing program services. Section 3.5.1 discusses how to fix this and some of the other problems with Java applets for RPC service publishing.

Without these basic capabilities, Java applets alone are not sufficient as a vehicle for publishing RPC services.

1.3.4 Component Object Technology

Component object technology solves the downloading problem just as Java applets do, and, in addition, components usually have strict interfaces that can be used as RPC interface specifications. Some components even include user documentation, which could be emphasized to become programmer documentation for RPC services.

OLE objects can export type information via the ITypeInfo and IProvideClassInfo interfaces, offer to present themselves using the IViewObject2 interface, and allow users to configure them through their IPropertyPage interface [3]. OLE objects can even make themselves scriptable

through OLE automation and the IDispatch interface. To document itself an OLE object could, for example, provide a rich help link in its property page. Users could then select the documentation right when they the need it most, i.e., when they are configuring the object for their purposes.

The central problem with distributed object technology, from our perspective, is that it does not interoperate with existing Web technology to provide interface discovery. Existing search engines cannot look inside objects to find their IPropertyPage interface and then extract their Help information; after all, existing search engines only understand HTML and text. Only object-smart search engines know where to look for the text, *if* it is available.

Similarly, a Web browser pointed at an ActiveX control knows only to activate it; the Web browser would have to be extended to look inside it and browse it. Because component objects are opaque boxes of code and data, they are impervious to tools that do not understand their object model in detail.

Of course, it would be a small step to extend search engines and Web browsers to speak to component objects on their own terms; after all, distributed object technology is intended to make objects drop easily into container documents and applications. However, if we do not want to modify search engines and Web browsers we have to stick to text.

So, distributed object technology provides interfaces and client stubs, but not interface discovery or browseable documentation. This technology is exemplified by the Object Management Group's CORBA [2] specification, Microsoft's DCOM combined with OLE [3], and the Java Beans [6] portable component architecture from Sun.

1.4 How We *Should* Publish RPC Services

Here we describe how we might publish RPC services in a perfect world. We describe the relationship of an RPC publishing environment to existing programming environments and we outline the salient features of those environments that we would like to reproduce. We also observe that most of the capabilities we desire exist in isolation, so "all" we need to do is put existing technologies together to achieve our ideal RPC service publishing system.

1.4.1 Support Network Interface Discovery

Contemporary programming environments allow programmers to browse and search library documentation as they work. We would like to extend this functionality to community programming environments. Specifically, we would like programmers to be able to browse and search documentation for RPC services on the network. They may want to search for particular functionality, or they may want to see what services are provided by a particular server.

1.4.2 Automate Downloading and Installation

Once they have found the service they are interested in, programmers should be able to "drop" that service into their programming environment and begin using it in their programs without further conscious installation. Generally speaking, we would like it if any code and data associated with a service was automatically downloaded (if necessary) and installed (to the extent necessary). That is, we would like it if programmers could *name* services anywhere on the network and have the system resolve the names by itself.

1.4.3 Provide Network Names for Services

Programmers should be able to name RPC instances, services, and servers anywhere, including their documentation. In the Web environment, one can name any document anywhere in one's own document; the same should be true for program-oriented content.

1.4.4 Check Interface Usage Mechanically

Once an RPC service has been incorporated into a program, uses of that service should be checked automatically. The extent of the checking we can expect is limited not only by the state of the art in program verification, but also by the availability of languages and language implementations that perform rigorous type checking. Nevertheless, we would like the use of an RPC service to be checked at least as rigorously as the use of a local subroutine library.

1.4.5 Reuse Existing Interfaces through Subtyping and Supertyping

To make the most of a publishing system, it must be possible, not only to call services from across the network, but also to design new services based on existing services on the network. In particular, it must be possible to declare new services as *extensions* or *subtypes* of existing services, and it must be possible to declare new services as *restrictions* or *supertypes* of existing services. See Section 1.2.3 on page 16 for our motivation for requiring *extensional* reuse and Section 1.2.2 on page 13 for our reasons for requiring *restrictional* reuse.

(This requirement is more ambitious than it sounds, probably too ambitions. The sections cited above observe that most existing programming languages do not support restrictional reuse. That is, it is not possible to declare a new interface to be a supertype of an existing interface, e.g., in C++ or Java.)

1.4.6 The Component Technologies are Available Now

Except for the so-called "restrictional reuse" issue mentioned above, most of the tools to support our requirements already exist.

- We have Web browsers and search engines that operate on text. Since we can deal with the textual representations of interfaces, text-oriented browsers and searchers suit the interface discovery task well.
- We can express interfaces precisely using modern programming languages like Java and Modula-3 [11].
- We have ubiquitous textual publication media, such as FTP and HTTP, which we can use to transport RPC service specifications.
- Similarly, we have sophisticated RPC systems like DCE RPC and Java RMI, which we can use to make the RPC calls.
- We even have specification languages for RPC interfaces, including the DCE IDL, Java itself, and the Xerox ILU IDL, which we can use to specify RPC services for the purpose of generating stubs.

So, why don't we already have a ubiquitous interface discovery system?

1.4.7 The Component Technologies have not been Combined

The gaps between the tools listed above are small but significant.

- There is no standard place to look on a server for its published RPC services. We recall that browsers and search engines are programs, so if we hope to build exploration tools that take advantage of the structure of RPC services, we need a standard place for them to look for published RPC services.

  (Of course, particular RPC systems have their standard places to look, but those RPC systems do not provide "services" in the sense we mean. In particular, they do not provide documentation.)

- There is no standard, programmatic representation that binds the documentation, specification, and client code of an RPC service. We have already seen that there are representations that handle some of these aspects, but no single representation handles all three.

The goal of this thesis is to fill in the gaps outlined above, producing a design for a prototypical RPC publishing system.

1.4.8 Summary

The way we *would* publish services is by providing interface discovery tools to programmers, which would lead them to precise, machine-checkable, network-nameable interface specifications. When a programmer calls an RPC service instance, the calls are type-checked automatically. All of this applies not only to the programmer writing applications, but also to the programmer writing new services that restrict or extend existing services.

Although we have technology that provides each of these properties, we do not have a synthesis of those technologies that provides a community RPC service publishing system. The design of a prototype of such a system is the purpose of this thesis.

1.5 Organization

The remainder of this thesis is organized as follows:

- Chapter 2 presents a detailed example of how the program development process would progress using a hypothetical RPC publishing system in a slightly idealized world.
- Chapter 3 describes the existing RPC and Web technologies we are using and explains precisely how our RPC publishing system combines them.
- Chapter 4 lists some interesting applications of a system for publishing RPC interfaces, summarizes the contribution of this thesis, and suggests possible directions for future work.

Chapter 2

An Example: A Student Information Service

In this chapter we present an example of the application development process using our proposed RPC publishing system. Our example is Podunk University's student information service (PU SIS).[1] The example shows how our proposed system lets applications cross organizational boundaries. In this case, various departments of Podunk University supply services for the SIS front end application.

2.1 Meet Jane Fromaine, Project Programmer

Jane Fromaine is assigned to the PU SIS project. She is an undergraduate student participating in the work/study program. Jane is an experienced ANSI C and DCE RPC programmer, but she is new to the RPC publishing system.

2.2 See Jane Read the Online Documentation

Jane begins by reading the online help for the RPC publishing system browser.

[1] This example is based loosely on MIT's student information service. No offense intended!

- Jane finds that she should use `rpscc` (the RPC publishing system C compiler) in place of `cc` (the usual C compiler). The `rpscc` compiler cooperates with the browser to provide transparent access to remote RPC services.

  The `rpscc` compiler simply calls `cc` with additional `-I` and `-L` arguments, which point to C header files and object libraries generated by the browser. The browser generates this code from RPC service and instance specifications Jane finds on the Web.

- When Jane resolves a URL (by clicking on it), the browser checks it for any RPC publications. If it finds any, the browser downloads and compiles the corresponding IDL specifications using the `idl` program, which is a standard part of OSF's DCE.

  For a service named *service*, the browser generates a header file called *service*`.h`, suitable for inclusion in Jane's C program with `#include` <*service*`.h`>. The browser also compiles `lib`*service*`.a`, a library of DCE client stub code. Jane links against this library with `rpscc -l`*service*.

  For an RPC *instance*, the browser first processes the corresponding RPC *service* and determines what DCE *object* the instance is associated with. (See Section 3.4.1 on page 43 for details on the representation of an RPC instance.) Then the browser generates code for the instance that calls on the service's code.

  For each operation provided by the instance's service, the stub code for the instance defines a procedure *instance_operation* that

  1. binds to the instance's server using the instance's DCE object UUID, obtaining a binding handle, and

  2. calls service's operation on that binding handle.

  (So the *instance_operation* procedure will have one less argument than the service's *operation* procedure, namely the binding handle.) The instance's code is stored in *instance*`.h` and `lib`*instance*`.a`.[2]

[2] These names are relatively short, so there is some danger of name collisions in a large program. Therefore, service and instance names should be chosen more carefully than this example chapter, as a whole, implies.

- Jane can replace the automatic compile-by-browsing behavior. Instead of compiling every RPC instance and service, the browser can present a "Generate Client Headers and Stubs" button.

  Naturally, wanting total control over her computing resources, Jane elects to turn off automatic client stub compilation. She doesn't want to waste her computer's time generating stubs she doesn't need! She will simply press the button when she wants to use an instance.

- Jane can use a "List RPC Instances" tool to list all the URL's in an ordinary Web page that correspond to published RPC instances. Jane can use this tool in concert with an ordinary Web search engine to find RPC services quickly.

Now that Jane has read the documentation, she is ready to use the RPC publishing system. She heads out on the local Podunk University Web.

2.3 See Jane Discover the Bursar's and Registrar's Services

Jane begins by searching `podunk.edu` for the Bursar's and Registrar's services with the Podunk University Search Engine (PUSE). Jane refines her query to `+"Podunk University" +authenticat* +student account registration schedules`, which returns 50 documents. Then she clicks on the "List RPC Instances" button.

Jane finds the Bursar's accounting service and the Registrar's schedule and registration services. These are surrounded by the physical education department's towel server (which uses the Bursar's service for billing and only distributes towels to registered students and faculty), the music department's lesson scheduler (which needs to interact with the Registrar's scheduler), and the information systems department's Kerberos IV authentication server (which provides authentication services for the entire university).

These are the six URL's:

```
http://www.podunk.edu/bursar/accounts
ftp://ftp.podunk.edu/registrar/registration
http://www.podunk.edu/registrar/schedules
```

```
http://www.pe.podunk.edu/rpc-services/towel-o-matic
ftp://liszt.podunk.edu/music-department/lessons
http://turing.podunk.edu/is/krb4
```

Notice that these look like ordinary URL's. They have been chosen as readable, sensible names. The RPC publishing system browser traverses their internal structure mechanically.

2.3.1 How does a Browser Display Published RPC Instances?

Figure 2-1 shows how an RPC publishing system browser might present Jane's search results. The browser is a Web browser with some modifications, implemented as plug-in modules for an existing browser.

The browser window has the usual features, including a title bar on top, a tool and menu bar immediately below the title bar, a scroll bar on the right, and a status line at the bottom. On the menu bar, you can see the "List RPC Instances" button that Jane pressed.[3] The status line indicates that RPC instances are listed in the main panel.

Each instance in the main panel is underlined to indicate that it is an ordinary URL that could be explored normally. Each instance also has a "Generate Headers and Stubs" button, presented by the browser in response to Jane pressing the "List RPC Instances" button.

2.4 See Jane Click to Generate Client Headers and Stubs

Now that she has found the right RPC instances, Jane proceeds to import them into her program. She simply clicks on the "Generate Headers and Stubs" button. The browser reads the IDL specification file associated with each URL and runs the DCE RPC `idl` program to generate the headers and client stub code described above.

At no time does Jane need to know where the RPC instances reside, whether they are replicated, or any similar implementation details. She needs only to read the instance documentation pages.

[3] The "List RPC Instances" button is on the menu bar for presentation purposes. A real browser would probably have this button on the "View" submenu.

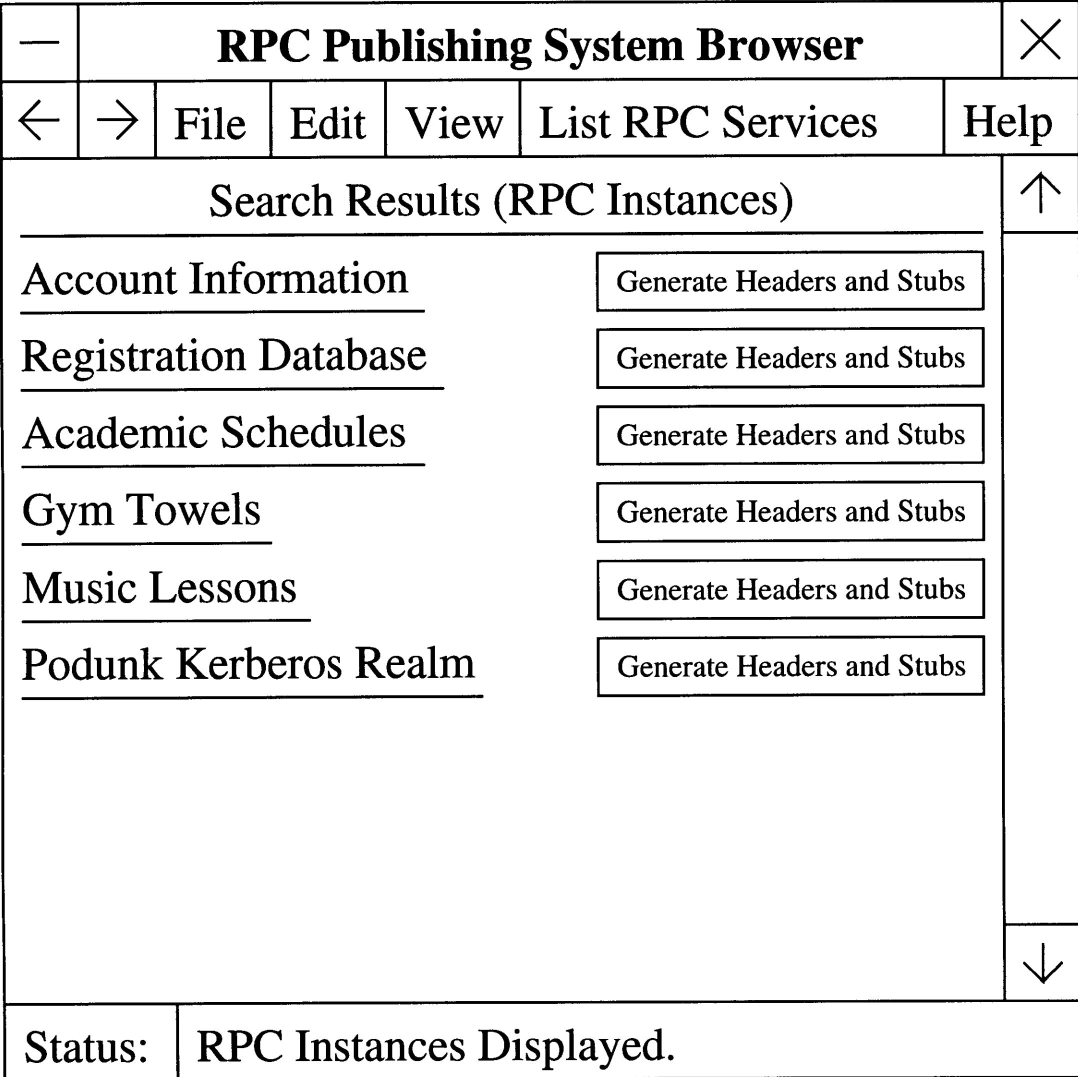

Figure 2-1: Hypothetical RPC Browser Displaying Search Results

2.5 See Jane Prepare her Project Makefile

Jane clicked on the "Generate Headers and Stubs" button impulsively. She continues by reading the *documentation* for the RPC instances she has imported. The documentation reminds her to write `#include <service.h>` at the tops of her program. In this case, Jane includes `accounts.h`, `registration.h`, and `schedules.h`. The documentation also reminds her to link with `-laccounts-lregistration-lschedules`, which she does by adding to the definition of `LIBS` in her makefile.[4]

Jane also recalls that she needs to use the `rpscc` compiler, so she sets `CC = rpscc` in her makefile. Then she thinks about it, and realizes that she is used to using GCC and her makefile depends on certain GCC features. She reads the manual for `rpscc` and discovers that she can have `rpscc` use `gcc` with the `-cc` option. Therefore, Jane recants and decides to set `CC = rpscc -cc gcc` in her makefile.[5]

These practices are

1. mentioned in the online Web documentation for the published RPC services she has found,
2. mentioned in the documentation for her RPC publishing service browser, and
3. suggested by Jane's experience as a C programmer. (An experienced C programmer immediately asks herself "What header file do I include to use this software module?" and "What library do I link against?".)

Jane finds the documentation perfectly digestible, because it describes a small change to her makefiles. She already knows how to do development this way.

Now Jane sets `SRCS = sis.c` in her slightly modified makefile and sets off to actually write `sis.c`, the Podunk University Student Information Service user interface program.

[4] Longer names would be used in practice to avoid name collisions, but for our example these names are good enough.

[5] This is just an example. The real `rpscc`, if it is ever implemented, and whatever it is called, may want to use a different option for choosing the underlying C compiler. Although the GCC manual seems to indicate that `-cc` is a compatible choice, other C compilers may use it for something else. There are many ways to deal with this problem, some of them are complete solutions that will work with any C compiler, and none of them are interesting in this context.

2.6 See Jane Write and Compile her Program

Jane adds `#include` directives to `sis.c`. She inserts a dummy `main` function, compiles her program, and runs it to test her makefile and the `rpscc` compiler. Everything works the first time! The reason is that Jane did not need to modify her stock makefile much, and it is already well tested and used.

Now Jane feels comfortable and ready to develop the application. She uses the `menu` library for her user interface, which is a text-based menu system that extends the `curses` library for visual terminals.[6] Since this is the prototype version, Jane leaves out any user configuration hooks. Those will be added later, when the core functionality works.

As an experienced C programmer, Jane quickly sorts out some compiler warnings related to the difference between C and IDL types. (For example, the IDL character type is `idl_char`, where the C type is `char`.) Before long, her prototype is working.

[6] Podunk University is old-fashioned; they have many DEC VT100 terminals in their computer labs. They want their SIS application to be extremely portable and useful on any student terminal, so they want Jane to avoid graphical user interfaces if possible.

Chapter 3

The Design of an RPC Publishing System

In this chapter, we describe the demands we place on an RPC publishing system and we describe how we meet those demands. We then describe the architecture of our system in detail. Finally, we describe alternative designs and our reasons for rejecting them.

3.1 Environment and Solution Components

Our RPC publishing system uses existing Web and DCE RPC technologies to achieve its goals. This chapter describes these components.

3.1.1 The Web is Used for Naming and Discovery

We rely heavily on the Web's URL naming system for global names. We also rely on existing Web browsers and search engines as tools of discovery based on the URL naming system.

We make no *direct* use of the Web's format for human-readable documents (HTML), type system (MIME), or protocol (HTTP).

The URL Naming System Provides Network Names

We need global names for the same reason that the Web does, i.e., so that publishers can collaborate by naming each other's work across administrative (organizational) boundaries.

The URL Naming System is Extensible by Construction

We want to extend the URL naming system so that it can be used to name RPC instances, servers, and services. This is possible because URL's are designed with some internal structure. In particular, they have a protocol component at the beginning that determines what the rest of the name means.[1]

Web Browsers and Search Engines are Useful for Discovery

Web browsers and Internet search engines already abound, and we would like to make as much use of them as possible. In particular, a powerful combination of browsing and searching has become the usual paradigm for exploring the Web. We embrace that paradigm for the additional purpose of discovering RPC services.

3.1.2 DCE RPC is Used at Runtime

The Open Software Foundation (OSF) has designed and implemented a Distributed Computing Environment (DCE) that is widely available. This standard includes a Remote Procedure Call (RPC) system, which we have adopted as the RPC system for our RPC publishing system.

DCE RPC is a sophisticated RPC system. We chose it for its IDL specification language, maturity, and powerful binding mechanisms. However, other RPC systems can be substituted for DCE RPC in our design, such as Java RMI, DCOM, and CORBA.

IDL Specifications Provide Essential Information

The DCE RPC Interface Definition Language (IDL) has several useful properties. The most interesting ones for our purposes are that

[1] This structure is similar to the structure of Monikers in OLE [3].

1. an IDL specification includes all information needed to mechanically check client and server implementations,
2. an IDL specification is simply an ASCII text document, so it is very portable, and
3. the tools to manipulate IDL specifications are part of the Distributed Computing Environment.

Authenticated RPC is Already Supported

Although we do not deal with security issues in this thesis, the question immediately arises: once we have published RPC services, how do we make them secure? OSF DCE already deals with many security issues and, in particular, authenticated RPC is already available. We would like to make use of this technology in the future.

3.2 Requirements for RPC Publishing

Our system is designed to satisfy certain requirements, which were outlined in Section 1.4 on page 21. Here we consider those requirements again, in more detail.

3.2.1 Communities Develop RPC Services

An RPC publishing system must enable communities to develop RPC services cooperatively. In particular, it must be possible for any service provider to specify their services and any client to use those services without the intervention of a third party, esp. a standards organization.

We require this decentralized administration because we believe that this is one of the essential aspects of the World Wide Web.

3.2.2 A Network's Services are Browseable and Searchable

An RPC publishing system must support interface discovery by browsing across the network. It must be possible to follow links from a server to all of its services and their interfaces. Further, it must be possible to obtain a list of servers to begin the browsing process.

An RPC publishing system must also support interface discovery through searching. We require only that it be possible for a programmer to search the documentation of published RPC servers, services, and interfaces. The publisher will be responsible for making the documentation useful and amenable to popular search techniques.

In particular, programmers will expect to be able to search and find servers that supply a particular service. This is simply an extension of the functionality of the existing Web, where users search to find documents pertaining to a particular topic.

It is not necessary for that programmers be able to search the interfaces of RPC services using constraints on the types of operations. This is a topic for further work discussed in Section 4.3.4.

3.2.3 Programs can Browse and Search

Our RPC publishing system must not be restricted to human use. It must also allow *programs* to discover interfaces. All of the typical browsing operations must be available, or potentially available, in programmatic form.

3.2.4 Components Have Network Names

Network-wide names will be available for RPC instances, servers, and services. A single name will be sufficient to identify both the WWW and RPC parts of an instance, server, service, or other component. These names will be compact and readable by humans, at least to the extent that URL's are human-readable.

3.2.5 No Runtime Overhead

When an RPC client binds to an RPC service, there will be *no* runtime overhead imposed by the RPC publishing service itself. The client will pay only the cost of the RPC binding and calling in the underlying RPC system.[2]

[2] Note that many RPC services are associated with naming systems, which may impose additional overhead as part of the RPC system. For example, the DCE RPC binding process may resolve DNS or X.500 names. Such embedded name resolutions are always considered part of the underlying RPC system overhead.

3.2.6 Trivial Tasks are Automated

Human intervention will *not* be necessary to download, install, or configure any software, if these actions are necessary. A single name will serve to link a client to a published RPC service or instance. The development software will be responsible for determining if and when any file transfers will take place.[3]

3.2.7 The System Works with the Existing Web

Existing Web browsers and search engines will be useful for discovering published RPC services and instances. Although some functionality will not be available through ordinary Web browsers, they will provide the same functionality for RPC services that they already provide for the World Wide Web.

In other words, the RPC publishing system will *extend* the Web rather than alter it in incompatible ways. We desire *backwards compatibility.*

3.3 Satisfying the Requirements of RPC Publishing

The requirements given in Section 3.2 determine the structure of our RPC publishing system. The most important feature dictated by the requirements is a unified global namespace.

3.3.1 A Single Namespace Binds Human and Program Services

The key to our RPC *publishing* service is a uniform naming system. All components in the system are named using URL's primarily because URL's are uniform names for network objects. Despite their shortcomings, URL's have proven adequate as global names.

Happily, using URL's as names also simplifies interaction with the existing Web. People and programs tend to recognize URL's even when they have unusual protocol specifiers like `dcerpc:`.

Ordinary Web browsers can manipulate RPC publishing system documents right up until they are asked to resolve `dcerpc:` URL's. Similarly, Web spiders and search engines have no problems

[3] Naturally, the user will be able to intervene when they wish. For example, if the software has installed too much software automatically and does not have a good heuristic for flushing its cache, the user may need to provide input.

handling RPC publishing system documents, because URL's are used as names.

Hopefully, future Web browsers will provide extension mechanisms that let them interpret new protocol specifiers, such as `dcerpc:`.

3.4 The Architecture of RPC Publishing

In this section we describe how our RPC publishing service is organized. Figure 3-1 summarizes the system's structure. The figure depicts a programmer and a program, both indicated by bugs ( ). The programmer, in the upper left, writes the program, in the upper right. At runtime, the program goes out and binds to the service instance it knew at compile time.

To write the program, the programmer makes use of a Web search engine (denoted by an eye ( )) and direct browsing to find the specification for an RPC service (denoted by a sheet ( )). This specification includes everything the programmer needs to prepare her program to use the RPC service instance. In particular, it specifies the RPC server (denoted by a cylinder ( )) and the instance within the server (denoted by a pyramid ( )).

At runtime, when the program uses the service instance, it binds and makes RPC calls in the underlying RPC system (DCE RPC). The binding is done via the DCE Directory Service (DS), which is used by the procedure `rpc_ns_binding_import_begin` [12]. The RPC publishing system is not directly involved in this phase of deployment, except that it provides the GDS name of the initial directory entry and, optionally, the object UUID used for binding.

In general, a program could use the services of the RPC publishing system to find other RPC service instances. The program could either browse or search and, having found an instance in which it was interested, the program could write another program to make use of the RPC service instance. Alternatively, the program could compile new modules that used the RPC instance and load them dynamically to make use of them. Figure 3-1 does *not* depict this more sophisticated potential behavior.

Let us consider the parts of this architecture individually.

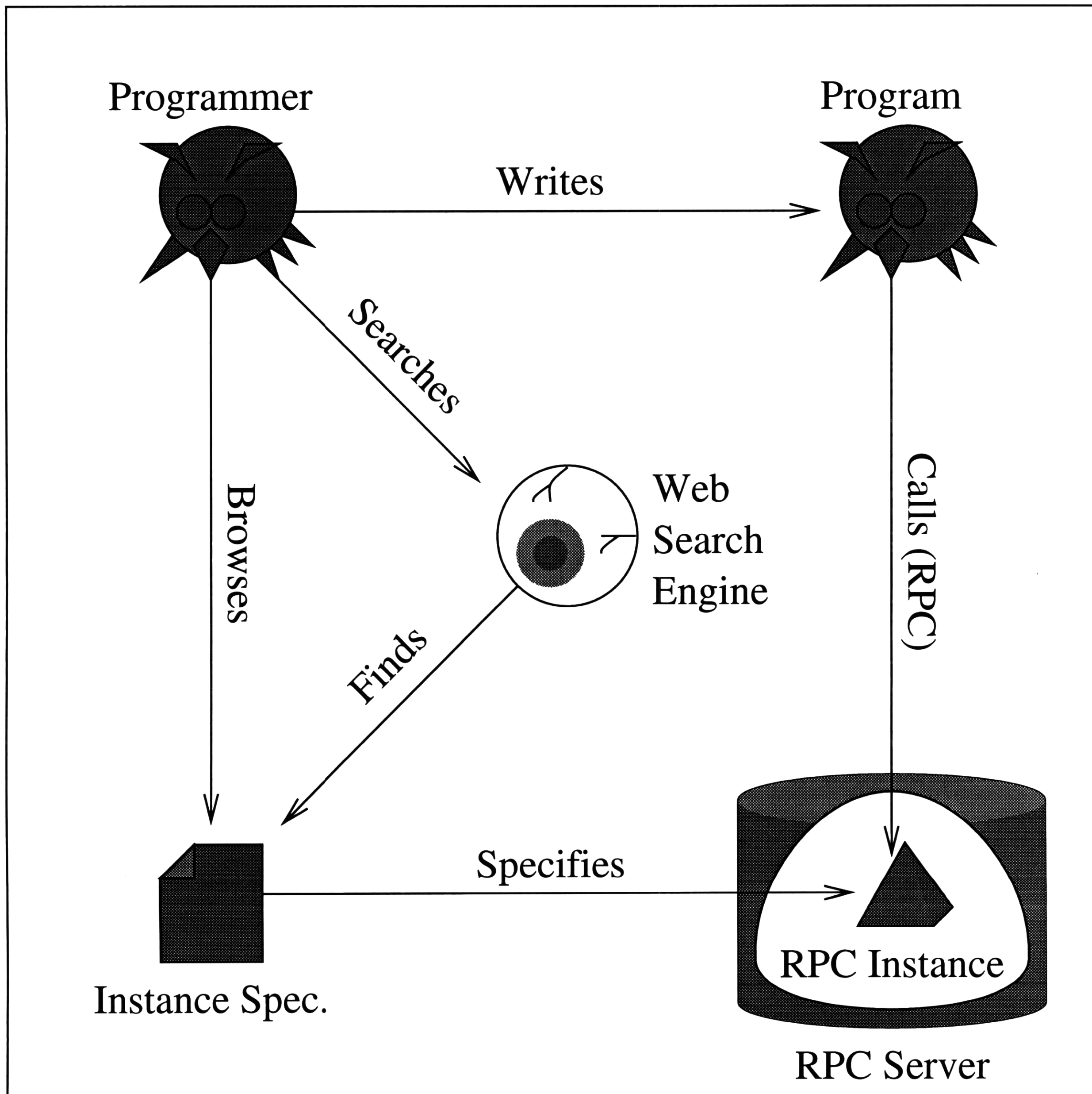

Figure 3-1: The Architecture of RPC Publishing

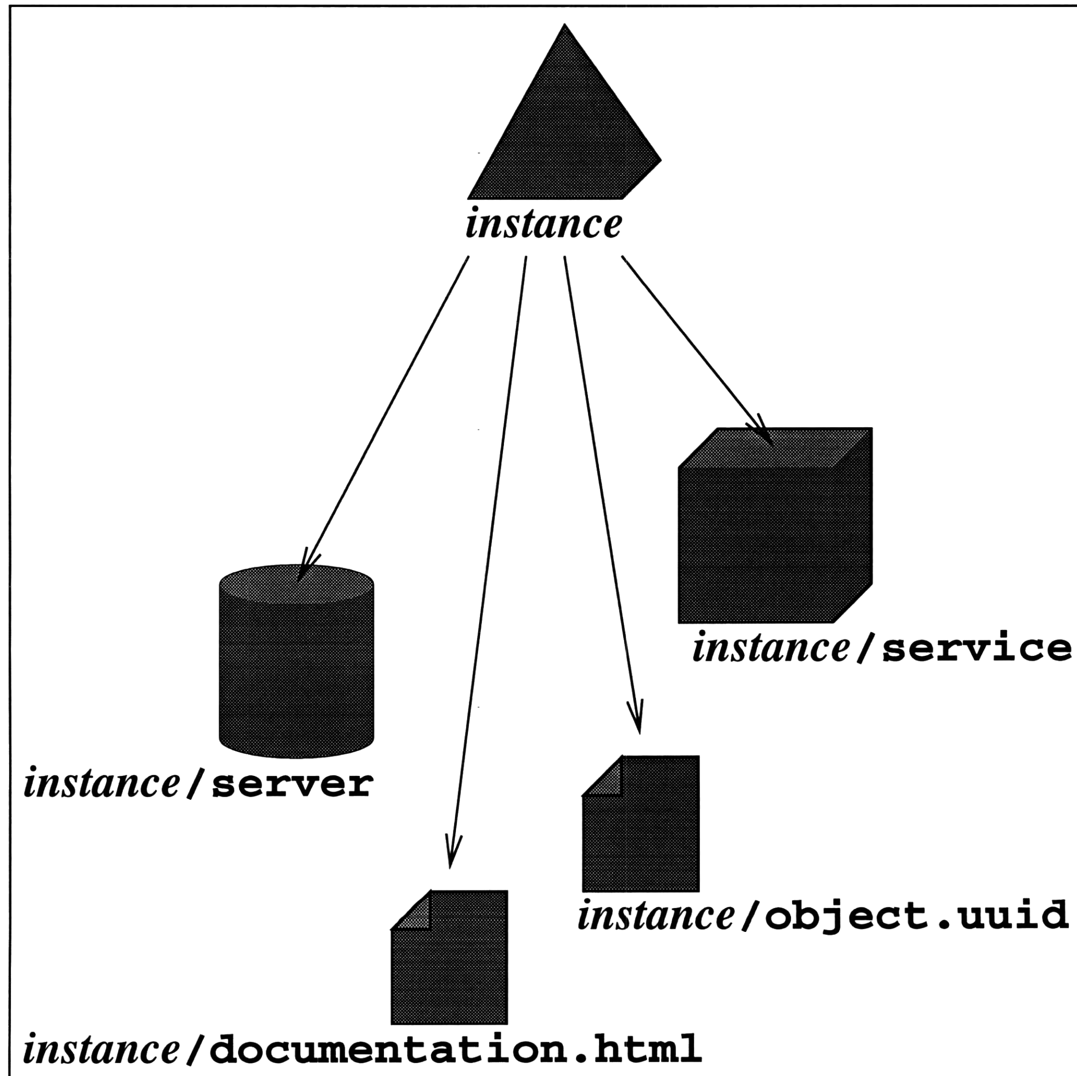

Figure 3-2: A Generic RPC Service Instance

3.4.1 Instances may be Specific RPC Objects

The structure of an RPC service instance is illustrated in Figure 3-2. It is based on the domain equation

$$\mathsf{Instance} = \mathsf{Service} \times \mathsf{Server} \times \mathsf{Maybe\ Object} \times \mathsf{Documentation}$$

where

$$\begin{aligned} \mathsf{Maybe}\ \tau &= \tau + \mathsf{Null} \\ \mathsf{Null} &= \{\mathrm{nil}\} \\ \mathsf{Object} &= \mathsf{UUID} \\ \mathsf{Documentation} &= \mathsf{HTML} \end{aligned}$$

As you can see, an instance may represent a specific DCE RPC object, although it need only represent a particular server process. We describe the parts of the right hand side of the equation in turn.

Service A description of the type of service that the instance provides. The most important part of the Service is its Interface. In general, the Service specifies how to use a certain class of RPC instances.

The Service corresponds roughly to a DCE RPC service, although an RPC publishing Service contains some additional information, specifically, some documentation. We will describe the Service type more precisely below, including another informal domain equation, in Section 3.4.3.

For a particular instance URL *'instance'*, the service URL is *instance*`/service`, i.e., the instance URL prepended to the literal string `/service`.

Server A description of the computer or process that implements the instance. In the context of DCE, this is specified by an arbitrary Directory Service (DS) entry, which may specify a single process running on a single host, a group of other entries, or a *profile* that prioritizes other entries.

Again taking the instance URL to be '*instance*', the server URL is *instance*/server. This partial URL leads to the DCE RPC server itself as well as documentation for the server. We will describe the Server type more precisely in Section 3.4.2, including another informal domain equation like the one above.

For an instance URL *instance*, the DCE RPC server URL is *instance*/server.url. The server URL is of the form dcerpc://*cell*/subsys/*server*, as described below.

Maybe Object An identifier for the particular instance on the server, to distinguish it from other instances that may be implemented by the same server. We use the same terminology that OSF DCE uses. An Object is simply a UUID in DCE RPC.

If there is an object UUID associated with the instance named by the URL '*instance*', the UUID is stored at the URL *instance*/object.uuid, as shown in the figure. If no particular object is associated with an instance, the file *instance*/object.uuid contains the string NULL.

On a given server, the object UUID distinguishes between multiple endpoints for the same service. If there is only one endpoint on the server, no object UUID is needed. Hence, the object UUID is optional.

Documentation Human-readable documentation for this instance in particular, as opposed to documentation for the service class or server. (Those components may have their own documentation.)

The format of the documentation need not be specified, but HTML is a popular choice. So is plain text, and so is Microsoft Word format.

The documentation for the service instance named by the URL '*instance*' is to be found at the derivative URL *instance*/documentation.html, assuming that it is provided in HTML form at least. One might also try *instance*/documentation.txt and *instance*/documentation.doc for the other forms mentioned above.

Notice that the components of an instance (service, server, and documentation) are lexically related. Given the documentation for an instance, one can recover the name of the instance itself and, from that, the service and server.

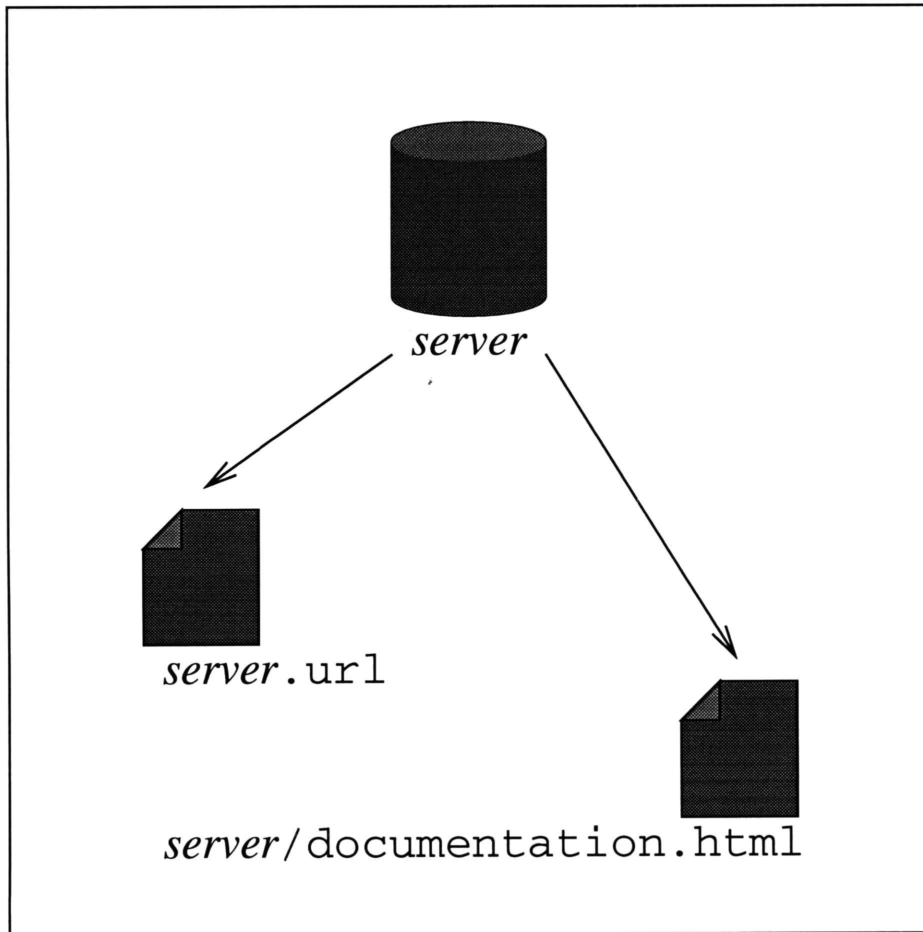

Figure 3-3: An RPC Publishing System Server

This property is very important, because it means that one can discover the HTML documentation for an instance on the Web using an ordinary Web browser or search engine and then, simply and automatically, e.g., obtain the IDL specification for its service class. Without the simple lexical relationship between these components, ordinary Web technology would be less useful as a part of the RPC publishing system.

3.4.2 Servers are Essentially `dcerpc:` URL's

As shown in Figure 3-3, an RPC publishing system server is simply a DCE RPC server and some documentation. The DCE RPC server is specified as a `dcerpc:` URL, described below. The documentation is typically in HTML form, as described already in Section 3.4.1 and elsewhere. The figure reflects the following informal domain equation:

$$\text{Server} = \text{URL} \times \text{Documentation}$$

For a particular server named by the URL *server*, the DCE RPC URL is stored in a file called *server*`.url`. This URL is of a special form described below, and it contains all the information needed at runtime to bind to the DCE RPC server.

As with RPC service instances, it is important that the URL's for the DCE RPC URL and documentation are lexically related. Having found the documentation for a service with AltaVista, for example, one can get to the DCE RPC URL immediately. If a browser knows about the RPC publishing system, it can move from the documentation to the server mechanically.

The DCE RPC Server URL Contains Binding Information

In the context of RPC services, we usually think of a server as the host that an RPC client binds to at runtime to obtain a service.

However, in the case of DCE RPC, it is more appropriate to think of the server as the locative information passed to the DCE RPC function `rpc_ns_binding_import_begin`. This information is most emphatically *not* a host name, but a more general specification. Precisely, it is a name to be resolved by the DCE Directory Service (DS) to a *directory entry*, which can specify a single server, a group of entries, or a *profile* that prioritizes other entries.

If we were using an RPC system without a directory service, e.g., ONC RPC, we could specify the server host name in the RPC URL. In the case of ONC RPC, we could use the following form for the server URL:

`oncrpc://`*hostname*`/`*prognum*`/`*version*

We would also have to change other details. For example, ONC RPC IDL specifications conven-

tionally end in `.x`, not `.idl`, and are called "RPC Language" files. Also, ONC RPC does not really support multiple server processes for the same service ID and version, so we would have to give up interchangeable server processes on a single server computer. But these are minor details.

Our RPC Publishing System Uses Explicit Binding

Ordinarily, DCE RPC programs can make use of *automatic binding*, where the RPC runtime takes complete responsibility for finding a server given an interface UUID. The RPC runtime searches for a server in some standard places, obviating the need for the programmer to worry about exactly where that binding information is coming from, and giving the system administrator control over the details of binding.

This technique is not suitable for our RPC publishing service, however. The applications we want to support are distributed across organizational boundaries, eliminating the possibility for centralized administration of the binding process.

Fortunately, DCE RPC provides an alternative binding mechanism called *explicit binding*. In this form of binding, the programmer provides a specific DS name from which to start searching for a server that implements a given interface. By making use of Global Directory Service (GDS) names, the server can provide such a name, which can be used anywhere. GDS names take this form:

`/.../`*cell*`/subsys/`*server*

The cell may be specified as a DNS domain name or an X.500 name.

The Structure of the `dcerpc:` URL

As discovered by the DCE-Web project [14], popular Web browsers do not support the URL system's inherent flexibility. That is, browsers do not usually provide any way to extend themselves with new protocols to replace HTTP, FTP, and such standard protocols specified by URL prefixes like `http:` and `ftp:`. This problem is discussed further in Section 3.5.2 on page 58.

Nevertheless, we would like to have *global names* for RPC servers. Therefor, we define the additional `dcerpc:` protocol specifier, which is used by RPC publishing system tools to communicate with RPC servers.

DCE RPC URL's take this form:

`dcerpc://`*cell*`/subsys/`*server*

The part of the URL after the `dcerpc:` protocol specifier is the significant part of a global DCE GDS name that specifies a directory entry for use with `rpc_ns_binding_import_begin` or a similar runtime binding procedures. The structure shown, where the server is a subdirectory of `subsys`, is conventional in OSF DCE.

Thus, the DCE RPC URL does *not* directly name a MIME object that can be downloaded and viewed. Although a sophisticated browser could provide a user interface to a `dcerpc:` URL, we do not assume such a browser. Instead, we think of a `dcerpc:` URL simply as a name for a DCE RPC server. The `dcerpc:` protocol specifier doesn't *really* specify a protocol, but rather a namespace for DCE RPC servers.

3.4.3 Services Include IDL Specifications

Figure 3-4 shows the structure of a generic RPC service. It is named by an arbitrary URL *service*, shown at the top of the figure. As you can see, a service links to two other documents, whose names are formed by lexical changes to *service*:

- The *documentation* link points to human-readable documentation describing the semantics of the service. The documentation can be provided in any human-readable MIME type, although we assume that it is provided in HTML form. (Presumably, documentation would be somewhere in the `text/*` MIME type.)

  The HTML documentation is obtained by following the URL *service*`/documentation.html`.

- The *specification* link points to machine-readable specifications for the service. Since we have chosen OSF DCE as our underlying distributed computing environment, the natural choice for the service specification is an IDL interface declaration. The name for the IDL specification is *service*`/specification.idl`, according to the DCE filename convention.

  The IDL definition contains the universally unique identifier (UUID) associated with the interface. The designer of the interface would have run `uuidgen` to generate the UUID for use as a nonce.

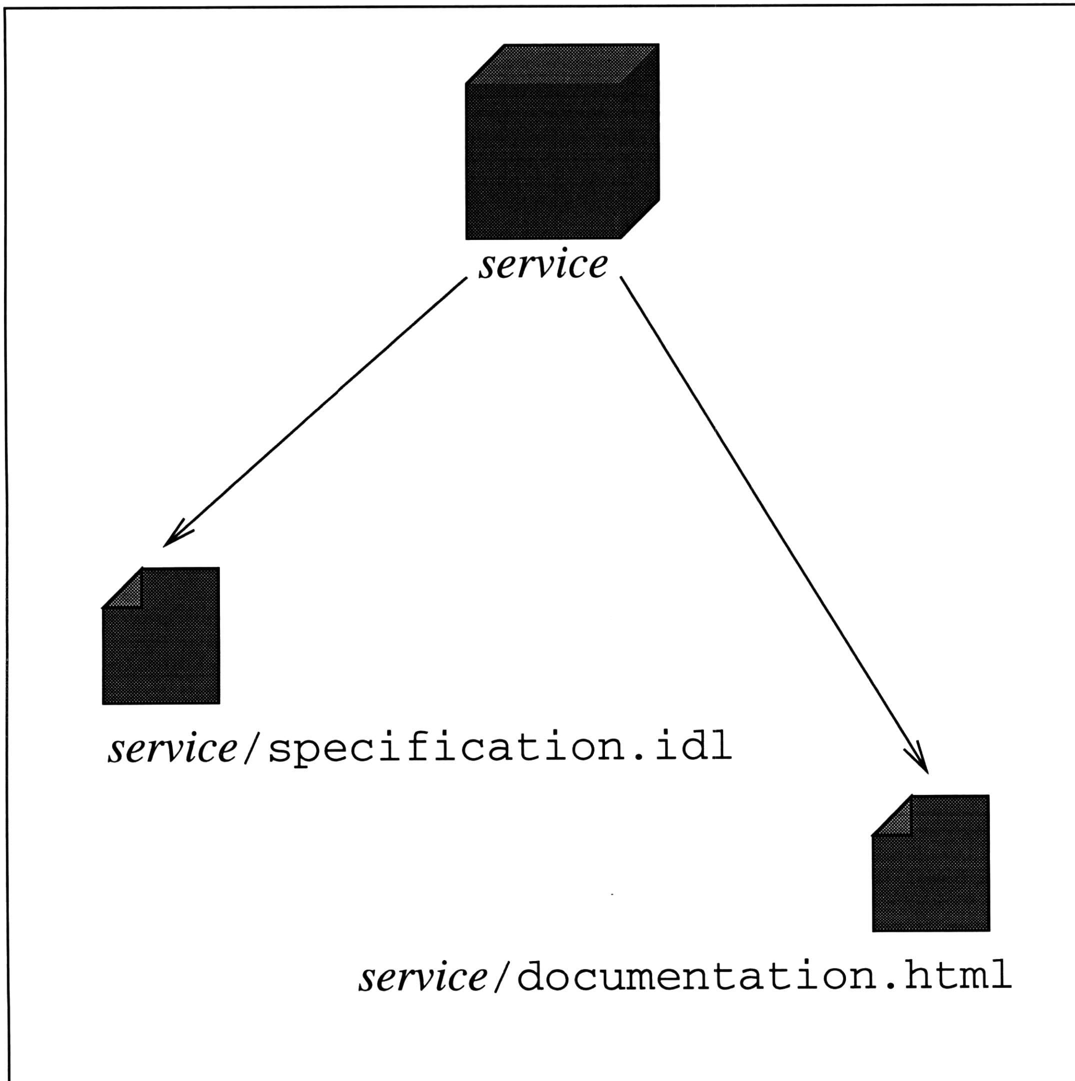

Figure 3-4: A Generic RPC Service

This structure for a service is motivated by the equations

$$\begin{aligned} \text{Service} &= \text{Specification} \times \text{Documentation} \\ \text{Specification} &= \text{IDL} \end{aligned}$$

This structure is what makes an instance machine-browseable.

Notice that the URL names of the HTML documentation and the IDL specification can be determined each from the other. This is important because it allows a person to discover the documentation using a human-oriented search service and immediately retrieve the machine-readable specification. Similarly, an IDL-aware search engine might find the specification without finding the documentation, but if the IDL file is part of a service then the HTML documentation is found immediately by replacing the `specification.idl` filename with `documentation.html`.

3.4.4 The Registry Lists a Server's Published Instances

There is a special service that can always be used to list published DCE RPC services. This service is called the *registry*, and it has a well-known UUID and IDL specification. The registry is very handy for browsing an administrative domain for useful services, from the inside *or* the outside of the domain.

The RPC publishing system registry is much like the ONC RPC portmapper. It is a service that is available at both the transport level and the higher level of RPC services. Just as `portmap` is a TCP/IP service, the registry is visible on the Web in the form of its HTML documentation. Similarly, `portmap` is a registered ONC RPC service, and the registry has a DCE RPC IDL specification.

By convention, the registry service instance on each Web server, if any, is called `http://`*server*`/services`. Thus, the IDL specification (shown in Figure 3-5) is located in `http://`*server*`/services/service/specification.idl` (because the registry is just another published RPC service). The registry instance documentation is located in `http://`*server*`/services/documentation.html`. The registry instance documentation should contain ordinary Web links to all the services enumerated by the RPC instance, so users of ordinary Web browsers can get some use out of the registry.

It might be convenient to generate the registry Web page on demand using a script. Such a script could generate the HTML list of services directly from the RPC publishing system list. Then the documentation need not be synchronized manually with the RPC service.

There is a Unique Registry Service

A possible IDL specification for the RPC publishing system's registry service is shown in Figure 3-5. As you can see, this IDL specification determines the registry service's UUID:

```
adcbc010-3b07-11d1-b3a6-08005afc8e16
```

Clients can assume that this UUID is always associated with the registry service, because I have reserved it for the purpose.[4]

As you can see, the registry service exports a single operation, `registry_instances`, which returns a list of URL's pointing to published RPC services.

Using the interface in Figure 3-5 and the DCE `idl` program, one can immediately generate a C language header file for the registry service. In addition, the `idl` program compiles client and server stub code that facilitates client and server program implementations.

The registry could, of course, be more sophisticated. It could provide search facilities, for example. The registry specified in Figure 3-5 is only version 1.0.

3.4.5 An Example of the RPC Publishing System URL's

After all that abstraction, it may be helpful to consider an hypothetical example of instance, server, service, and registry names. For this purpose, suppose we here at the Programming Systems Research Group (PSRG) decided to publish an RPC service for our Content Routing System (CRS) [9].

Since we (hypothetically) put all our public information regarding CRS in `http://www.psrg.lcs.mit.edu/crs/`, we put the DCE IDL specification for our CRS RPC service in `http://www.psrg.lcs.mit.edu/crs/service/specification.idl`. The structure of published RPC services described in Section 3.4.3 dictates that the service documentation go in `http://www.psrg.lcs.mit.edu/crs/service/documentation.html`.

[4] I generated the registry UUID using `uuidgen` on my friend Jim Klossner's computer. Thanks, Jim.

```
/* Interface identification information goes at the top, like this: */
[
  uuid(adcbc010-3b07-11d1-b3a6-08005afc8e16),
  version(1.0)
]

/* The interface is named by a compound declaration, like this one: */
interface registry
{
  /*
    All but the simplest types need to be given names for use in the
    interface declaration, much as in C.  The text in square brackets
    is DCE RPC meta information.  In this case, it informs the RPC
    runtime that this is really a NULL-terminated string:
  */
  typedef [ptr, string] char * url_t;

  /*
    The DCE RPC IDL inherits C's unpleasant array syntax, but we can
    declare the length explicitly.  This type is an open array of
    URL's tagged with its length at runtime:
  */
  typedef [ptr] struct {
    unsigned long length;
    [size_is(length)] url_t urls[];
  } * urls_t;

  /*
    Operation declarations look like ANSI C prototypes.  This one
    returns a pointer, whose referenced data will actually be copied
    over the network by the RPC runtime as necessary:
  */
  urls_t registry_instances(void);
}
```

Figure 3-5: Registry IDL Specification

This naming convention makes `http://www.psrg.lcs.mit.edu/crs/` a likely candidate for our local RPC instance, since it already has a valid `service` component. So, we store `NULL` in `http://www.psrg.lcs.mit.edu/crs/object.uuid` and we some cursory documentation in `http://www.psrg.lcs.mit.edu/crs/documentation.html` with a link to the general CRS documentation in `index.html`.

Here at PSRG, we have no DCE service. But if we did, we would specify a DCE server by storing something like `dcerpc://psrg.lcs.mit.edu/subsys/crs` in `http://www.psrg.lcs.mit.edu/crs/server.url`. The exact DCE RPC URL would depend on how our DCE cell was configured.

Finally, by convention, we would have registry documentation at `http://www.psrg.lcs.mit.edu/services/documentation.html`, which would provide ordinary Web links to all our services, including CRS. The corresponding published RPC service, i.e., the registry service, would be specified at `http://www.psrg.lcs.mit.edu/services/service/specification.idl`, as per the general structure of published RPC instances.

3.4.6 Discovery may use Extensions to the Web

The discovery process begins with search engines and Web browsers, denoted in Figure 3-6 with eyes ( ) and ships ( ) respectively. The double arrows labeled *lexical* indicate that RPC publishing system objects are named so that you can go from the Web content to the RPC content and back. (Refer back to Section 3.4.1, Section 3.4.2, and Section 3.4.3 for details.) Thus, both ordinary Web tools and specialized RPC publishing tools have their places in the system.

- A hypothetical RPC-service-aware search engine can parse the DCE IDL specification for services, providing IDL-oriented search capabilities. For example, a user could search for a service that has a particular named operation satisfying certain type constraints. Building such a search engine would be an interesting topic for further work; see Section 4.3.4 on page 66.

- A text search engine is likely to find the documentation page for a particular instance or service. The documentation URL, in turn, provides a way to get back to the instance or, in this case, the service that it pertains to, including the IDL specification. By following the backlink, you can go from a text search to a complete published RPC interface.

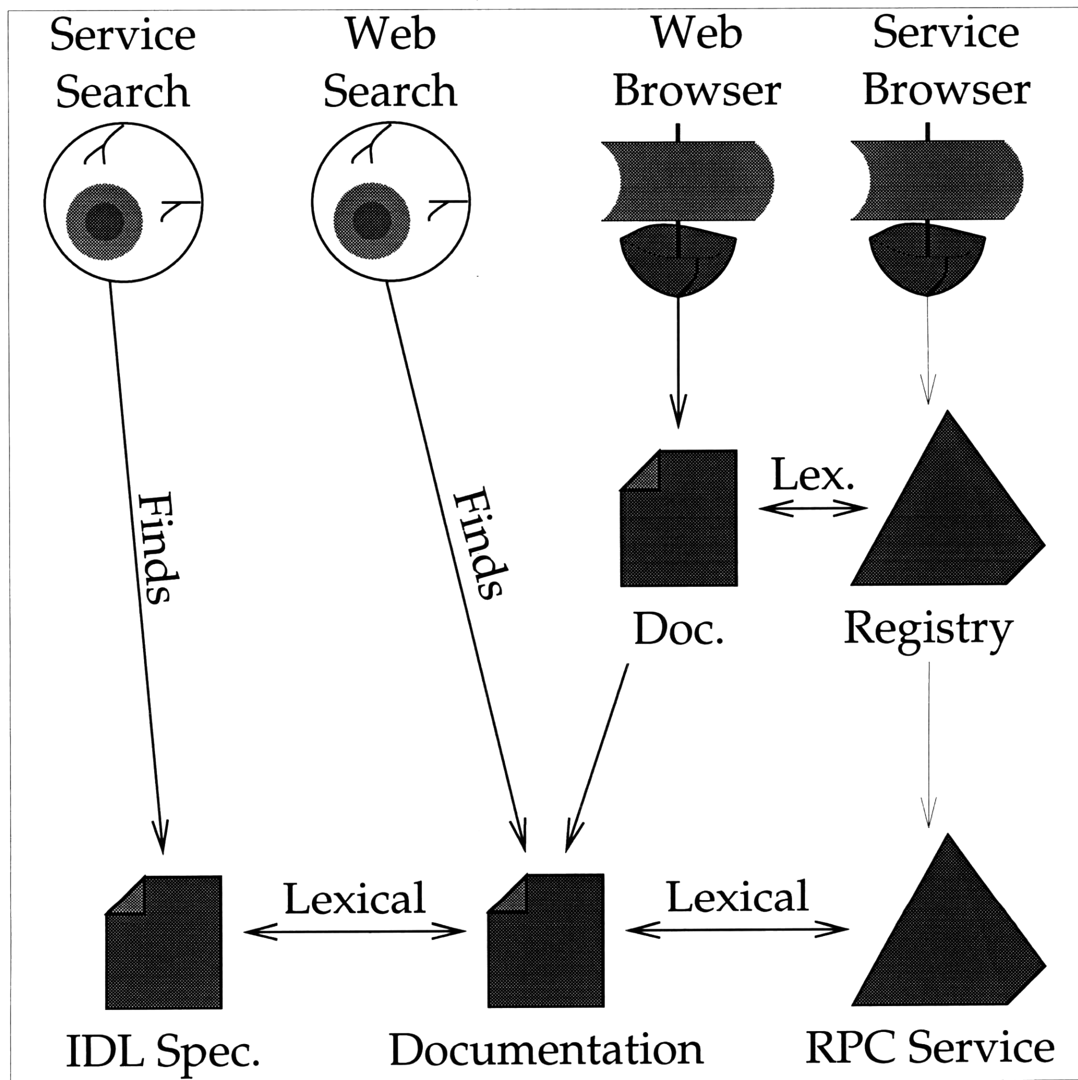

Figure 3-6: A Programmer Discovering a Service

- A service-aware browser, on the other hand, knows how to browse servers. It knows exactly where to look for a given server's registry. More importantly, the specialized browser understands the structure of the registry and the services it points to, so it can present that information to the user in a convenient form. The user may prefer a summary of the documentation, annotated specifications, or a hierarchy of services; all of these can be automated, hypothetically, using the structure of the RPC publishing system.

  Knowing the location of the interface, a specialized service browser automatically downloads it along with other interfaces it includes from the same location. The browser automatically generates C language headers and stub code from the IDL files so they are ready for use by the DCE RPC programmer. The browser also presents the programmer with a URL for the service. Either way, the programmer immediately begins using the service in her program, checked by the specification.

- Finally, an ordinary Web browser can also be used to discover interfaces in a less convenient way. Specifically, if a programmer knows about the registry documentation URL, she can point her Web browser at it and arrive at the documentation for the server.

  The documentation is likely to point at documentation for all the exported service instances. If it does, and the documentation for each service instance points back to the service instance itself, the programmer can easily find and download the interfaces she will need to communicate with this particular server.

  If the registry documentation is less ambitious and does not point to documentation for individual service instances, all is not lost. The programmer can still point her browser at the well-known registry URL, download the IDL interface definition for registries, and use it in a program to obtain pointers to the published services. This is a rather tedious route, so we expect documentation to be rich in cross-links so this route can be avoided.

  The same observations apply to individual services. Once a programmer finds the URL for a service, she can try the documentation and then the IDL interface. The drawbacks of this exploration method are that the programmer has to know the conventions and download manually.

So, even with existing searching and browsing tools, a programmer that knows about RPC service registries explores the published interfaces of a server much as she would explore the Internet for software. With specialized tools, she finds and incorporates published RPC services by pointing and clicking just a few times.

3.4.7 Ordinary DCE RPC Binding Occurs at Runtime

Figure 3-7 shows what happens after the program is compiled and run, as it seeks to connect to the remote RPC service. This is simply a description of the DCE RPC binding mechanism, which is why the global and cell directory services exist in the first place. The entire process takes place inside `rpc_ns_binding_import_begin`.

1. The DCE RPC runtime receives a GDS name from the client code.
2. It consults the DCE Directory Service, possibly tracing out some RPC configuration profiles set up by system administrators. See the *OSF DCE Guide to Developing Distributed Applications* [12] for more information about the structure of RPC configuration profiles. We simply observe that the search will eventually lead to a *server entry* in the GDS, which specifies the machine to contact.
3. The *endpoint mapper* on that machine knows exactly how to contact the RPC service instance, i.e., server process.
4. The RPC runtime stores that information for later use.

Here you can see that other RPC systems would also be suitable for use in our RPC publishing system, but DCE RPC's sophisticated binding mechanism really makes this last step much more powerful. We can just use a *name* for the server elsewhere in our RPC publishing system, without regard for its identity, because the DCE RPC runtime takes care of binding to the server.

3.5 Alternative Designs

In this section we describe alternatives to the design we will outline in Section 3.4. We also give reasons why we did not pursue each of these alternatives.

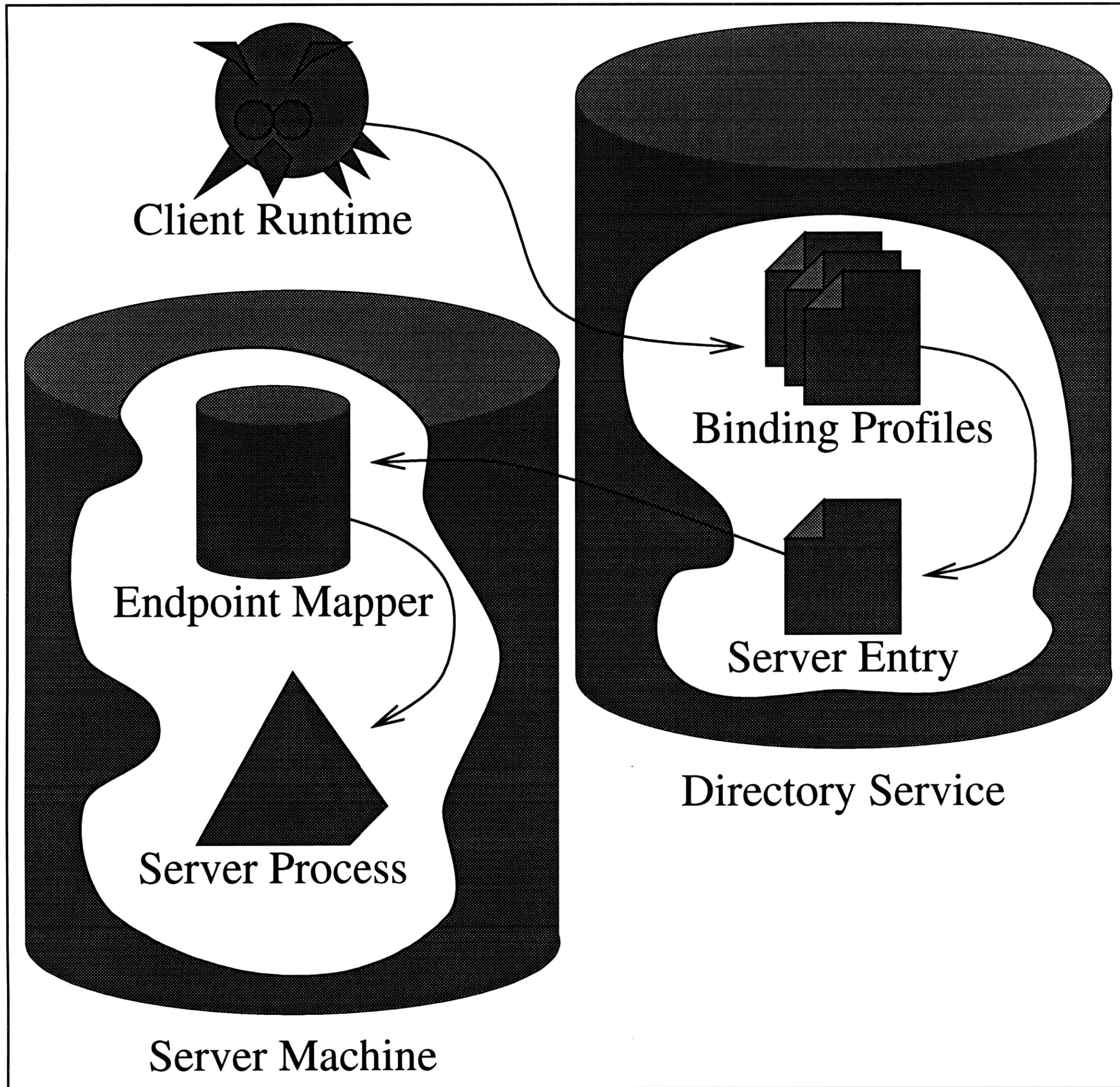

Figure 3-7: A Program Binding to a Service

```
import java.applet.Applet;
import java.net.URL;

interface Documented {
  URL documentation();
}

class DocumentedApplet extends Applet implements Documented {
  URL documentation() {
    return URL("documentation.html");
  }
}
```

Figure 3-8: A Possible `DocumentedApplet` Class

3.5.1 Making Java Applets Publishable

In Section 1.3.3, we mentioned that Java applets, at the present time, have several problems. Assuming that Java is secure and a production-quality RPC system is available, we could fix the other problems as follows.

- We could define a subclass of the `Applet` class that provides the missing documentation. We might call this a `DocumentedApplet`. It might look something like the class presented in Figure 3-8. We might also want to provide a way for the applet to provide its documentation directly rather than through a URL, but the essential intention is expressed clearly by the `DocumentedApplet` class.
- We could modify the Java virtual machine to accept a larger class of names for packages in the `import` declarations. In particular, we could include URL's in the syntax. Java applets could then express their interdependencies across the Web.

Thus, we could include documentation in Java applets and name interfaces over the network.

3.5.2 Using a Local Proxy to Present Services

The DCE Web project [14] provides a very interesting implementation strategy for protocol extensions to the World Wide Web. It uses the ubiquitous capability of Web browsers to request documents through *proxy servers*. The DCE Web local proxy server mediates between DCE-ignorant browsers and DCE-authenticated Web servers.

As described in [15], the DCE Web project chose not to extend the URL naming system with new protocols because popular Web browsers did not support protocol extensions at the time. That is, one could not convince a typical browser to resolve URL's beginning with `foo:` by some external means, such as running a program or connecting to an alternative IP port.[5] However, by extending the HTTP namespace, the DCE Web project got the same effect as a protocol extension.

Although the DCE Web project is security-oriented, the idea of a local proxy server carries over to our RPC publishing system. An RPC publishing proxy server could interpret specially formed URL's, presenting their content via an HTML user interface. Normal Web documents would be forwarded without modification by an RPC publishing system proxy.

Consider, for example, a very simple design where a DCE GDS URL like

`dcerpc://`*cell*`/subsys/`*server*

is mapped into the HTTP URL namespace as

`http://.../`*cell*`/subsys/`*server*.

Even such a simple mapping works well, because `...` is not a legal DNS host name, so the fabricated `http:` URL would never collide with a real `http:` URL.[6]

One important disadvantage of this approach is mentioned by Lewontin in [15]. The local proxy server, while easily localized to a single machine, is difficult to localize to individual users on that machine. So, although it is simple to control access on a single-user machine or in the context of a particular brand of Unix, a portable multi-user solution is elusive. Thus, there is some concern about security with this approach, which is especially important when using an otherwise security-conscious system like DCE.

We avoid the the proxy solution because it obscures the RPC publishing system's semantics. The *meaning* of a `dcerpc:` URL is simply different from the meaning of an `http:` URL; the first is a service, the second is a document. We want to distinguish them thoroughly. Further, we hope that future Web browsers will be easy to extend with new protocols, so it will not be so unthinkable that we introduce the `dcerpc:` protocol (and perhaps the `oncrpc:` protocol) into the URL namespace.

[5] In fact, this is still true today.

[6] This example uses a DNS-based name, but X.500-based GDS names also begin with `...`, so the scheme works for all DCE GDS names.

3.5.3 Integrate a CORBA ORB with the Web

Another alternative is to extend our system to include CORBA as an underlying RPC mechanism. This alternative embraces the object-oriented paradigm and a carefully designed standard for distributed objects. To outline this alternative, we would

- use the CORBA IDL as an interface specification language,
- use existing CORBA services to represent meta-information at runtime, namely the dynamic invocation interface (DII) and dynamic skeleton interface (DSI), and
- provide additional services for obtaining documentation.

Adding support for CORBA would be straightforward, especially as some modern browsers already support it (e.g., Netscape Navigator).

3.5.4 Use Java RMI as an RPC System

The Java Remote Method Invocation (JRMI) [8] system provides an RPC system for Java. JRMI would, for the most part, make a good RPC system on which to base an RPC publishing system. Java has some additional appeal, in that it provides a portable representation of programs and popular Web browsers support Java applets directly.

At the present time, the Java RMI system is much less mature than DCE RPC. Further, DCE RPC provides the option of authentication, which JRMI does not.

In any case, it would be straightforward to modify the RPC publishing system described in this thesis to use JRMI. The architecture of the Java RMI system includes a binding process and a server-side registry not fundamentally different from the ones used in other RPC systems. The `jrmi:` URL protocol specifier would be similar to the `oncrpc:` protocol specifier mentioned in Section 3.4.2.

3.5.5 Use Microsoft's DCOM as an RPC System

The Distributed Component Object Model (DCOM) is an RPC system based on DCE RPC. This RPC system is very attractive at least in part because it is the RPC system used to support Distributed OLE, a very popular distributed object system.

Although DCOM is not itself a rich object system like CORBA, it does provide the basic RPC functionality needed by an RPC publishing system. However, DCE RPC provides certain potential that DCOM does not, including authenticated RPC.

Chapter 4

The Future of RPC Publishing

We have motivated and defined RPC publishing, described an example of how the development process is improved by RPC publishing, and given a detailed paper design for a particular RPC publishing system. Now, we conclude our presentation on RPC publishing. First, we list applications that might benefit from the use of an RPC publishing system. Second, we state our RPC publishing system's contribution to the field of Computer Science. Finally, we and indicate some of the many ways in which our work could be extended in the future.

4.1 Applications of RPC Publishing

As described in Section 1.2 on page 12, there are many reasons to publish RPC services. Let us consider some examples of applications that might be implemented better using published RPC services compared to private RPC services (DCE RPC).

4.1.1 Collecting Stock Quotes from the Internet

A popular use of the Web is to view stock quotes. For many people, the delayed quotes available for free on the Web are adequate. Some people write scripts that automatically retrieve and process these stock quotes.

Getting Stock Quotes is Tedious

Scripts that read stock quotes jump through hoops. They call Web browsers with programmatic interfaces, such as Lynx, to download stock quote pages. These pages are generated by other scripts on their servers, so we have two scripts communicating via human-readable HTML. The script reading the stock quotes then parses the Web page to obtain the embedded stock quote information. Thus, we have an *ad hoc*, unreliable, inelegant process where a database of stock quotes is squeezed through an unsuitable transport (HTML).

Published RPC is Good for Accessing Public Databases

An RPC interface to a stock quote database might improve the speed and efficiency of retrieving stock quotes by orders of magnitude. If the publishers or consumers of this information are interested in efficiency, then they would certainly prefer an RPC interface. Users who write scripts to fetch their stock quotes will also prefer and RPC interface, simply because it makes their scripts much easier to write.

4.1.2 Searching the Web

The AltaVista search engine [22], for example, has access to a large database of content information regarding Web pages all over the Web. Like other search engines, AltaVista provides a rich user interface, but no programmatic interface.

Parsing HTML is a Fragile Process

One could certainly parse the HTML document returned by a query. However, small changes in the layout of the result can fool the parser. For example, if Digital decided to display search results in a special font, one would have to alter one's parser to take the `FONT` tag into account.

Unusual Search Needs Require Programmatic Support

If you are searching the Web for all valid OSF DCE IDL specifications, for example, you have a problem. There is no AltaVista query you can write that will select only valid IDL files. You can make excellent use of AltaVista to narrow your search by searching only for files whose names end

in `.idl` containing the keywords `uuid`, `version`, and `interface`. However, you still need to look at these files to make sure they are syntactically valid, because

Thus, in order to implement this unusual search, you would like to preselect certain documents using a large search engine like AltaVista and then postprocess the results. However, as described above, parsing the HTML returned by an AltaVista query is a tedious, fragile process. You would prefer a programmatic interface to AltaVista, which would allow you to reliably search from inside a program.

4.2 Contribution

Our RPC publishing system contributes the possibility of a community program development environment comparable to integrated development environments (IDE's). The most novel aspect of this environment is the interactive discovery and use of programmatic interfaces. We have based our system on existing Web technology to leverage the growing base of existing Web applications.

4.3 Further Work

This section describes possible extensions of this thesis work.

4.3.1 An Implementation Would be Nice

This document is a paper design. The obvious next step in this work is to implement some program services and a program publishing server, which could be browsed and searched with existing Web technology. The obvious next step after that is to implement a service browser based on an existing Web browser.

4.3.2 A Java Implementation Would be Nicer

One interesting implementation possibility is a Java implementation. Such an implementation would be able to bootstrap our system onto existing Web technology. (That is, if the RPC publishing system were written in Java, it could be packaged as a Java applet to run inside contemporary Web

browsers.) In addition, such an implementation would be relatively simple to produce, leveraging the work already done on the Java API.

4.3.3 Declaring Interfaces as Supertypes

As described in Section 1.2 on page 12, it would be helpful to be able to define a new interface as the *supertype* of an existing interface, *without* the cooperation of the other interface's publisher. One could explore the connection between this type system issue and the publication of RPC services.

4.3.4 Interface Discovery Based on Programming Language Types

It may be useful for programmers to search for interfaces by specifying constraints on the types of operations declared in those interfaces. Although this kind of search is not supported by existing commercial information retrieval (IR) technology, previous research has examined type-based searches of the interfaces to subroutine libraries [20, 21].[1] Future research could explore the feasibility and utility of network interface discovery based on programming language types.

[1] Thanks to Mark Sheldon for these interesting references.

Bibliography

[1] Edward Bailey. *Maximum RPM*. Red Hat Press, August 1997. ISBN 0672311054.

[2] Ron Ben-Natan. *CORBA: A Guide to the Common Object Request Broker Architecture*. McGraw-Hill, October 1995. ISBN 0070054274.

[3] Kraig Brockschmidt. *Inside OLE*. Microsoft Press, second edition, 1995.

[4] Jeremy Brown. Inquir user information database. `ftp://ftp.lcs.mit.edu/pub/inquir/README`, August 1992.

[5] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification, June 1995. RFC 1813.

[6] Patrick Chan and Rosanna Lee. *The Java Class Libraries:* `java.applet`, `java.awt`, `java.beans` *(Vol. 2)*. Addison-Wesley, second edition, October 1997. ISBN 0201310031.

[7] Patrick Chan, Rosanna Lee, and Doug Kramer. *The Java Class Libraries:* `java.io`, `java.lang`, `java.math`, `java.net`, `java.security`, `java.text`, `java.util` *(Vol. 1)*. Addison-Wesley, second edition, March 1998. ISBN 0201310023.

[8] Troy Bryan Downing. *RMI: Developing Distributed Java Applications With Remote Method Invocation and Object Serialization*. IDG Books Worldwide, January 1998. ISBN 0764580434.

[9] Andrzej Duda and Mark Sheldon. Content routing in networks of WAIS servers. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, pages 124–133, Poznan, Poland, June 1994.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, September 1996. ISBN 0201634511.

[11] Samuel P. Harbison. *Modula-3*. Prentice Hall, 1992. ISBN 0-13-596396-6.

[12] Jr. Harold W. Lockhart. *OSF DCE Guide to Developing Distributed Applications*. McGraw-Hill, 1994.

[13] K. Harrenstien. Name/Finger, December 1977. RFC 742.

[14] Steve Lewontin. The DCE-Web: Securing the enterprise Web, July 1996. `http://www.osf.org/www/enhance/swwhitep.htm`.

[15] Steve Lewontin. Secure Web off-the-shelf solutions, July 1996. `http://www.osf.org/www/enhance/swots.htm`.

[16] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[17] Jan-Simon Pendry and Nick Williams. *AMD: The 4.4 BSD Automounter Reference Manual*, March 1991. 5.3 Alpha.

[18] J. Postel and K. Harrenstien. Time protocol, May 1983. RFC 868.

[19] The Regents of the University of California. *UNIX System Manager's Manual*, March 1991. 4.3 Berkeley Distribution.

[20] Mikael Rittri. Using types as search keys in function libraries. In *International Conference on Functional Programming Languages and Computer Architecture*, pages 174–183. ACM and IFIP, September 1989.

[21] Colin Runciman and Ian Toyn. Retrieving re-usable software components by polymorphic type. In *International Conference on Functional Programming Languages and Computer Architecture*, pages 166–173. ACM and IFIP, September 1989.

[22] Richard Seltzer, Eric J. Ray, and Deborah S. Ray. *The AltaVista Search Revolution — How to find anything on the Internet*. McGraw-Hill, 1997.

[23] R. Srinivasan. Binding protocols for ONC RPC version 2, August 1995. RFC 1833.

[24] R. Srinivasan. Remote procedure call protocol version 2, August 1995. RFC 1831.

[25] R. Srinivasan. XDR: External data representation standard, August 1995. RFC 1832.

[26] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 116–127, Paris, France, October 1997.

[27] D. Zimmerman. The Finger user information protocol, December 1991. RFC 1288.