Remote Procedure Calls

Paul Krzyzanowski
Rutgers University

October 30, 2012

1 Introduction, or what's wrong with sockets?

Sockets are a fundamental part of client-server networking. They provide a relatively easy mechanism for a program to establish a connection to another program, either on a remote or local machine and send messages back and forth (we can even use read and write system calls). This interface, however, forces us to design our distributed applications using a read/write (input/output) interface which is not how we generally design non-distributed applications. In designing centralized applications, the procedure call is usually the standard interface model. If we want to make distributed computing look like centralized computing, input/output-based communications is not the way to accomplish this.

In 1984, Birrell and Nelson devised a mechanism to allow programs to call procedures on other machines. A process on machine A can call a procedure on machine B. When it does so, the process on A is suspended and execution continues on B. When B returns, the return value is passed to A and A continues execution. This mechanism is called the **Remote Procedure Call** (**RPC**). To the programmer, it appears as if a normal procedure call is taking place. Obviously, a remote procedure call is different from a local one in the underlying implementation.

2 Steps in a remote procedure call

Let us examine how local procedure calls are implemented. This differs among compilers and architectures, so we will generalize. Every processor provides us with some form of *call* instruction, which pushes the address of the next instruction on the stack and transfers control to the address specified by the call. When the called procedure is done, it issues a *return* instruction, which pops the address from the top of the stack and transfers control there. That's just the basic processor mechanism that makes it easy to implement procedure calls. The actual details of identifying the parameters, placing them on the stack, executing a call instruction are up to the compiler. In the called function, the compiler is responsible for ensuring any registers that may be clobbered are saved, allocating stack space for local variables, and then restoring the registers and stack prior to a return.

None of this will work to call a procedure that is loaded on a remote machine. This means that the compiler has to do something different to provide the illusion of calling a remote procedure. This makes remote procedure calls a **language-level** construct as opposed to sockets, which are an **operating system level** construct. We will have to simulate remote

procedure calls with the tools that we do have, namely local procedure calls and sockets for network communication.

The entire trick in making remote procedure calls work is in the creation of **stub functions** that make it appear to the user that the call is really local. On the client, a stub function looks like the function that the user intends to call but really contains code for sending and receiving messages over a network. The sequence of operations that takes place, shown in Figure 1 (from p. 693 of W. Richard Steven's UNIX Network Programming), is:

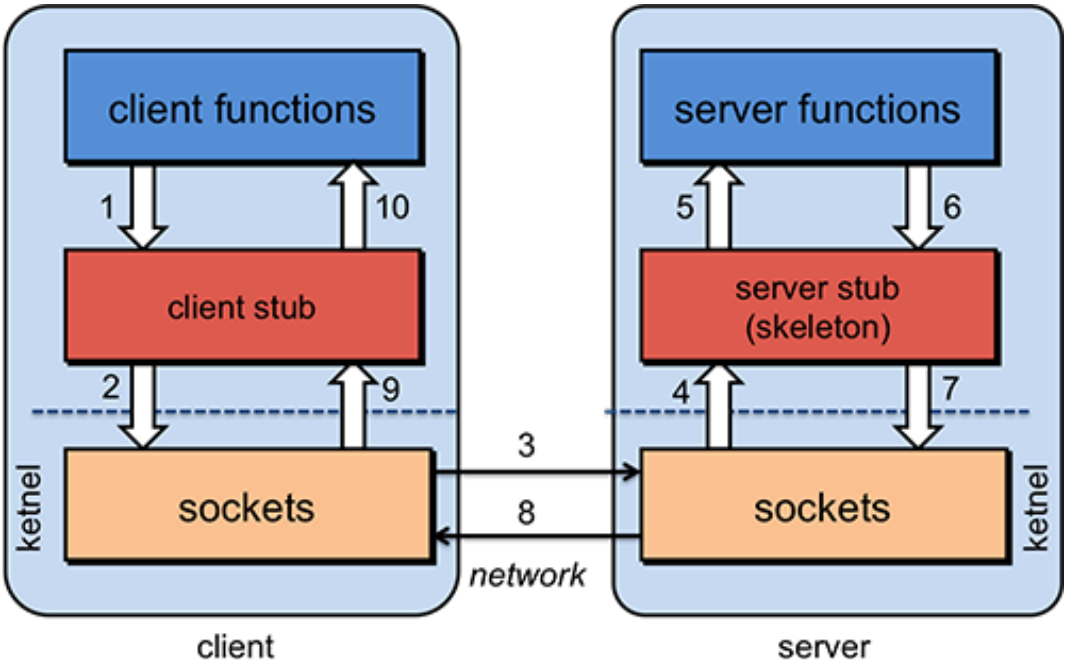

Figure 1: Figure 1. Steps in executing a remote procedure call

1. The client calls a local procedure, called the **client stub**. To the client process, this appears to be the actual procedure, because it is a regular local procedure. It just does something different since the real procedure is on the server. The client stub packages the parameters to the remote procedure (this may involve converting them to a standard format) and builds one or more network messages. The packaging of arguments into a network message is called **marshaling** and requires **serializing** all the data elements into a flat array-of-bytes format.

2. Network messages are sent by the client stub to the remote system (via a system call to the local kernel using *sockets* interfaces).

3. Network messages are transferred by the kernel to the remote system via some protocol (either connectionless or connection-oriented).

4. A **server stub**, sometimes called the **skeleton**, receives the messages on the server. It **unmarshals** the arguments from the messages and, if necessary, converts them from a standard network format into a machine-specific form.

5. The server stub calls the server function (which, to the client, is the remote procedure), passing it the arguments that it received from the client.
6. When the server function is finished, it returns to the server stub with its return values.
7. The server stub converts the return values, if necessary, and marshals them into one or more network messages to send to the client stub.
8. Messages get sent back across the network to the client stub.
9. The client stub reads the messages from the local kernel.
10. The client stub then returns the results to the client function, converting them from the network representation to a local one if necessary.

The client code then continues its execution.

The major benefits of RPC are twofold. First, the programmer can now use procedure call semantics to invoke remote functions and get responses. Secondly, writing distributed applications is simplified because RPC hides all of the network code into stub functions. Application programs don't have to worry about details such as sockets, port numbers, and data conversion and parsing. On the OSI reference model, RPC spans both the session and presentation layers (layers five and six).

3 Implementing remote procedure calls

Several issues arise when we think about implementing remote procedure calls.

3.1 How do you pass parameters?

Passing by value is simple: just copy the value into the network message. Passing by reference is hard. It makes no sense to pass an address to a remote machine since that memory location likely points to something completely different on the remote system. If you want to support passing by reference, you will have to send a copy of the arguments over, place them in memory on the remote system, pass a pointer to them to the server function, and then send the object back to the client, copying it over the reference. If remote procedure calls have to support references to complex structures, such as trees and linked lists, they will have to copy the structure into a pointerless representation (e.g., a flattened tree), transmit it, and reconstruct the data structure on the remote side.

3.2 How do we represent data?

On a local system there are no data incompatibility problems; the data format is always the same. With RPC, a remote machine may have different byte ordering, different sizes of integers, and a different floating point representation. The problem was dealt with in the IP protocol suite by forcing everyone to use big endian byte ordering for all 16 and 32 bit fields in headers (hence the use of *htons* and *htonl* functions). For RPC, we need to come up with a "standard" encoding for all data types that can be passed as parameters if we are to communicate with heterogeneous systems. ONC RPC, for example, uses a format called XDR (eXternal Data Representation) for this process. These data representation formats

can use implicit or explicit typing. With **implicit typing**, only the value is transmitted, not the name or type of the variable. ONC RPC's XDR and DCE RPC's NDR are examples of data representations that use implicit typing. With **explicit typing**, the type of each field is transmitted along with the value. The ISO standard ASN.1 (Abstract Syntax Notation), JSON (JavaScript Object Notation), Google Protocol Buffers, and various XML-based data representation formats use explicit typing.

3.3 What machine and port should we bind to?

We need to locate a remote host and the proper process (port or transport address) on that host. One solution is to maintain a centralized database that can locate a host that provides a type of service. This is the approach that was proposed by Birell and Nelson in their 1984 paper introducing RPC. A server sends a message to a central authority stating its willingness to accept certain remote procedure calls. Clients then contact this central authority when they need to locate a service. Another solution, less elegant but easier to administer, is to require the client to know which host it needs to contact. A name server on that host maintains a database of locally provided services.

3.4 What transport protocol should be used?

Some implementations allow only one to be used (e.g. TCP). Most RPC implementations support several and allow the user to choose.

3.5 What happens when things go wrong?

There are more opportunities for errors now. A server can generate an error, there might be problems in the network, the server can crash, or the client can disappear while the server is running code for it. The transparency of remote procedure calls breaks here since local procedure calls have no concept of the failure of the procedure call. Because of this, programs using remote procedure calls have to be prepared to either test for the failure of a remote procedure call or catch an exception.

3.6 What are the semantics of remote calls?

The semantics of calling a regular procedure are simple: a procedure is executed exactly once when we call it. With a remote procedure, the **exactly once** aspect is quite difficult to achieve. A remote procedure may be executed:

- 0 times if the server crashed or process died before running the server code.
- once if everything works fine.
- once or more if the server crashed after returning to the server stub but before sending the response. The client won't get the return response and may decide to try again, thus executing the function more than once. If it doesn't try again, the function is executed once.
- more than once if the client times out and retransmits. It is possible that the original request may have been delayed. Both may get executed (or not).

RPC systems will generally offer either **at least once** or **at most once** semantics or a choice between them. One needs to understand the nature of the application and function of the remote procedures to determine whether it is safe to possibly call a function more than once. If a function may be run any number of times without harm, it is *idempotent* (e.g., time of day, math functions, read static data). Otherwise, it is a *nonidempotent* function (e.g., append or modify a file).

3.7 What about performance?

A regular procedure call is fast: typically only a few instruction cycles. What about a remote procedure call? Think of the extra steps involved. Just calling the client stub function and getting a return from it incurs the overhead of a procedure call. On top of that, we need to execute the code to marshal parameters, call the network routines in the OS (incurring a mode switch and a context switch), deal with network latency, have the server receive the message and switch to the server process, unmarshal parameters, call the server function, and do it all over again on the return trip. Without a doubt, a remote procedure call will be much slower. We can easily expect the overhead of making the remove call to be thousands of times slower than a local one. However, that should not deter us from using remote procedure calls since there are usually strong reasons for moving functions to the server.

3.8 What about security?

This is definitely something we need to worry about. With local procedures, all function calls are within the confines of one process and we expect the operating system to apply adequate memory protection through per-process memory maps so that other processes are not privy to manipulating or examining function calls. With RPC, we have to be concerned about various security issues:

- Is the client sending messages to the correct remote process or is the process an impostor?
- Is the client sending messages to the correct remote machine or is the remote machine an impostor?
- Is the server accepting messages only from legitimate clients? Can the server identify the user at the client side?
- Can the message be sniffed by other processes while it traverses the network?
- Can the message be intercepted and modified by other processes while it traverses the network from client to server or server to client?
- Is the protocol subject to replay attacks? That is, can a malicious host capture a message an retransmit it at a later time?
- Has the message been accidentally corrupted or truncated while on the network?

4 Programming with remote procedure calls

Many popular languages today (C, C++, Python, Scheme, et alia) were not designed with built-in syntax for remote procedures and are therefore incapable of generating the necessary

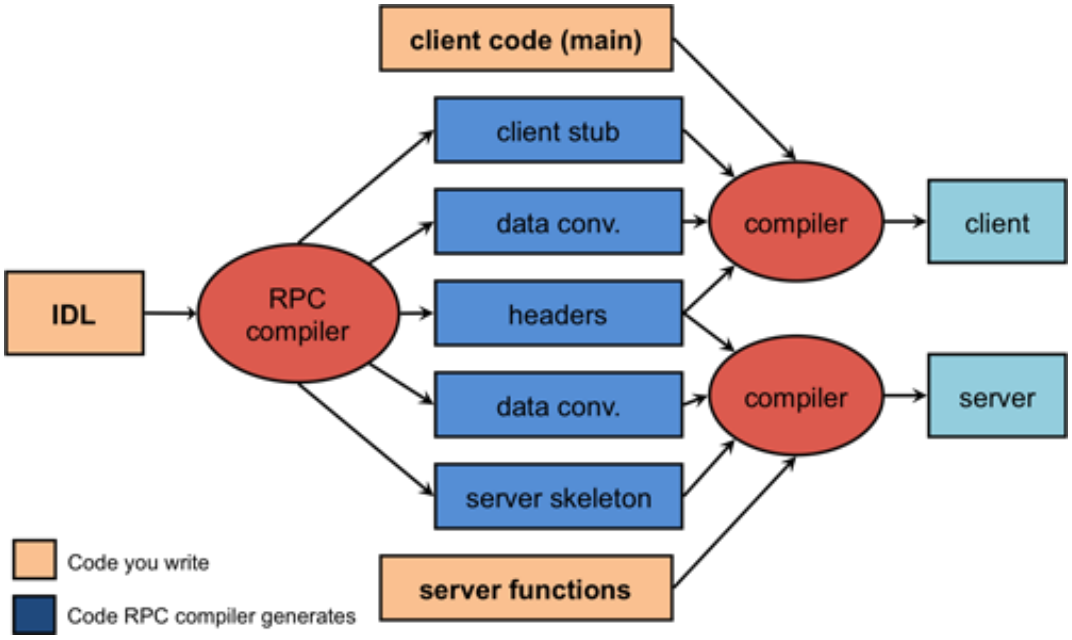

Figure 2: Figure 2. Compilation steps for remote procedure calls

stub functions. To enable the use of remote procedure calls with these languages, the commonly adopted solution is to provide a separate compiler that generates the client and server stub functions. This compiler takes its input from a programmer-specified definition of the remote procedure call interface. Such a definition is written in an **interface definition language**.

The interface definition generally looks similar to function prototype declarations: it enumerates the set of functions along with input and return parameters. After the RPC compiler is run, the client and server programs can be compiled and linked with the appropriate stub functions (Figure 2). The client procedure has to be modified to initiallyize the RPC mechanism (e.g. locate the server and possibly establish a connection) and to handle the failure of remote procedure calls.

5 Advantages of RPC

- You don't have to worry about getting a unique transport address (picking a unique port number for a socket on a machine). The server can bind to any available port and then register that port with an RPC name server. The client will contact this name server to find the the port number that corresponds to the program it needs. All this will be invisible to the programmer.

- The system can be independent of transport provider. The automatically-generated server stub can make itself available over every transport provider on a system, both TCP and UDP. The client can choose dynamically and no extra programming is required since the code to send and receive messages is automatically generated.

- Applications on the client only need to know one transport address: that of the name server that is responsible for telling the application where to connect for a given set of server functions.
- The function-call model can be used instead of the send/receive (read/write) interface provided by sockets. Users do't have to deal with marshaling parameters and then parsing them out on the other side.

6 RPC API

Any RPC implementation needs to provide a set of supporting libraries. THese include:

Name service operations Register and look up binding information (ports, machines). Allow an application to use dynamic (operating system assigned) ports.

Binding operations Establish client/server communications using the appropriate protocol (establish communication endpoints).

Endpoint operations Register endpoint information (protocol, port number, machine name) to the name server and listen for procedure call requests. These functions are often called from the automatically-generated main program, the server stub (skeleton).

Security operations The system should provide mechanisms for the client and server to be able to authenticate each other and to provide a secure communication channel between the two.

Internationalization operations (possibly) These are very rarely a part of an RPC package but may include functions to convert time formats, currency formats, and language-specific strings through string tables.

Marshaling/data conversion operations Functions to serialize data into a flat array of bytes for transmitting onto a network and functions to reconstruct it.

Stub memory management and garbage collection Stubs may need to allocate memory for storing parameters, particularly to simulate pass-by-reference semantics. The RPC package needs to allocate and clean up any such allocations. They also may need to allocate memory for creating network buffers. For RPC packages that support objects, the RPC system needs a way of keeping track whether remote clients still have references to objects or whether an object can be deleted.

Program ID operations Allow applications to access identifiers (or handles) of sets of RPC interfaces so that the the set of interfaces offered by a server can be communicated and used.

Object and function ID operations Allow the ability to pass references to remote functions or remote objects to other processes. Not all RPC systems support this.

7 The first generation of Remote Procedure Calls

7.1 ONC RPC (formerly Sun RPC)

Sun Microsystems was one of the first commercial offerings of RPC libraries and an RPC compiler. It was offered on Sun computers in the mid 1980s and supported Sun's Network File System (NFS). The protocol was pushed as a standard by Open Network Computing, a consortium headed by Sun and AT&T. It is a very lightweight (and light featured) RPC system that is available on most POSIX and POSIX-like operating systems, including Linux, SunOS, OS X, and various flavors of BSD. It is commonly referred to as Sun RPC or ONC RPC.

ONC RPC provides a compiler that takes the definition of a remote procedure interface and generates the client and server stub functions. This compiler is called **rpcgen**. Before running this compiler, the programmer has to provide the **interface definition**. The interface definition contains the function declarations, grouped by version numbers and identified by a unique **program number**. The program number enables clients to identify the interface that they need. Version numbers are useful to allow clients that have not updated to the latest code to still be able to connect to a newer server by having that server still support old interfaces.

Parameters over the network is marshaled into an implicitly-typed serialized format called **XDR** (eXternal Data Representation). This ensures that parameters can be sent to heterogenous systems that may use different byte ordering, different size integers, or different floating point or string representations. Finally, Sun RPC provides a run-time library that implements the necessary protocols and socket routines to support RPC.

All the programmer has to write is a client procedure (client.c[1]), the server functions (server.c[2]), and the RPC interface definition (date.x[3]). When the RPC interface definition (a file suffixed with .x; for example, a file named date.x) is compiled with *rpcgen*, three or four files are created. These are for the date.x example:

date.h Contains definitions of the program, version, and declarations of the functions. Both the client and server functions should include this file.

date_svc.c C code to implement the server stub.

date_clnt.c C code to implement the client stub.

date_xdr.c Contains the XDR routines to convert data to XDR format. If this file is generated, it should be compiled and linked in with both the client and server functions.

The first step in creating the client and server executables is to compile the data definition file date.x[4]. After that, both the client and server functions may be compiled and linked in with the respective stub functions generated by *rpcgen*.

In older versions of, the transport protocol would be either the string "tcp" or the string "udp" to specify the respective IP service RPC. This is still supported and must be used with the Linux implementation of RPC. To make the interface more flexible, UNIX System V

[1] rpc/client.c
[2] rpc/server.c
[3] rpc/date.x
[4] rpc/date.x

release 4 (SunOS since version 5) network selection routines allow a more general specification. They search a file (/etc/netconfig) for the first provider that meets your requirements. This last argument can be:

"netpath" search a NETPATH environment variable for a sequence of preferred transport providers)

"circuit_n" find the first virtual circuit provider in the NETPATH list), "datagram_n" (find the first datagram provider in the NETPATH list)

"visible" find the first visible transport provider in /etc/netconfig)

"circuit_v" find the first visible virtual circuit transport provider in /etc/netconfig)

"datagram_v" find the first visible datagram transport provider in /etc/netconfig).

Each remote procedure call was initially restricted to accepting a single input parameter along with the RPC handle. The system was only later modified to support multiple parameters. Support for single parameter RPC is still the default for some versions of *rpcgen*, such as the one on Apple's OS X. Passing multiple parameters had to be done by defining a structure that contains the desired parameters, initializing it, and passing that structure.

A call to a remote procedure returns a pointer to the result rather than the desired result. The server functions have to be modified to accept a pointer to the value declared in the RPC definition (.x file) as an input and return a pointer to the result value. On the server, the pointer must be a pointer to static data. Otherwise, the area that is pointed to will be undefined when the procedure returns and the procedure's frame is freed). On the client, the return of a pointer allows us to distinguish between a failed RPC (null pointer) versus a null return from the server (indirect null pointer).

The names of the RPC procedures are the names in the RPC definition file converted to lower case and suffixed by an underscore followed by a version number. For example, BIN_DATE becomes a reference to the function bin_date_1. Your server must implement bin_date_1 and the client code should issue calls to bin_date_1.

7.2 What happens when we run the program?

The server

When we start the server, the server stub code (program) runs and puts the process in the background (don't forget to run *ps* to find it and kill it when you no longer need it [5]). It creates a socket and binds any local port to the socket. It then calls a function in the RPC library, svc_register, to register the program number and version. This contacts the **port mapper**. The port mapper is a separate process that is usually started at system boot time. It keeps track of the port number, version number, and program number. On UNIX System V release 4, this process is called *rpcbind*. On Linux, OS X, and BSD systems, it is known as *portmap*.

[5] This is the default behavior. A command-line flag to rpcgen disables the server automatic running as a daemon.

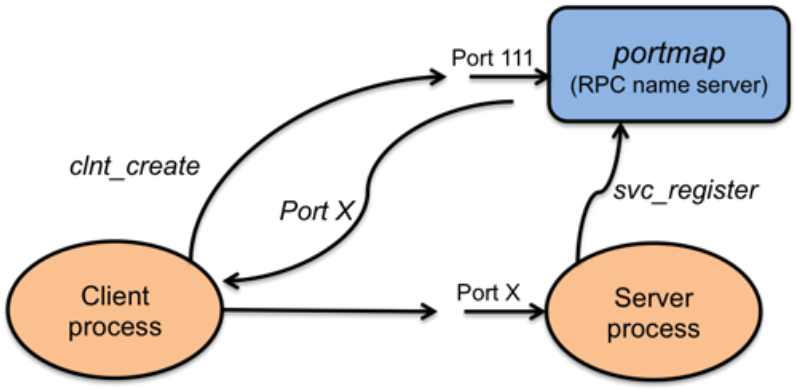

Figure 3: Figure 3. Function lookup in ONC RPC

The client

When we start the client program, it first calls *clnt_ create* with the name of the remote system, program number, version number, and protocol. It contacts the port mapper on the remote system to find the appropriate port on that system.

The client then calls the RPC stub function (*bin_ date_ 1* in this example). This function sends a message (e.g., a datagram) to the server (using the port number found earlier) and waits for a response. For datagram service, it will retransmit the request a fixed number of times if the response is not received.

The message is then received by the remote system, which calls the server function (*bin_ date_ 1*) and returns the return value back to the client stub. The client stub then returns to the client code that issued the call.

8 RPC in the Distributed Computing Environment (DCE RPC)

The Distributed Computing Environment (DCE) is a set of components designed by the Open Software Foundation (OFS) for providing support for distributed applications and a distributed environment. After merging with X/Open, this group became The Open Group. The components provided as part of the DCE include a distributed file service, a time service, a directory, service, and several others. Of interest to us here is the DCE remote procedure call. It is very similar to Sun RPC. Interfaces are written in an interface definition language called **Interface Definition Notation** (**IDN**). Like Sun RPC, the interface definitions look like function prototypes.

One deficiency in Sun RPC is the identification of a server with a "unique" 32-bit number. While this is a far larger space than the 16-bit number space available under sockets, it is still not comforting to come up with a number and hope that it is unique. DCE RPC addresses this deficiency by not having the programmer think up a number. The first step in writing an application is getting a unique ID with the **uuidgen** program. This program generates a prototype IDN file containing an interface ID that is guaranteed never to be used again. It is a 128-bit value that contains a location code and time of creation encoded in it. The user then edits the prototype file, filling in the remote procedure declarations.

After this step, an IDN compiler, **dceidl**, similar to rpcgen, generates a header, client stub, and server stub.

Another deficiency in Sun RPC is that the client must know the machine on which the server resides. It then can ask the RPC name server on that machine for the port number corresponding to the program number that it wishes to access. DCE supports the organization of several machines into administrative entities called **cells**. Every machine knows how to communicate with a machine responsible for maintaining information on cell services: the **cell directory server**.

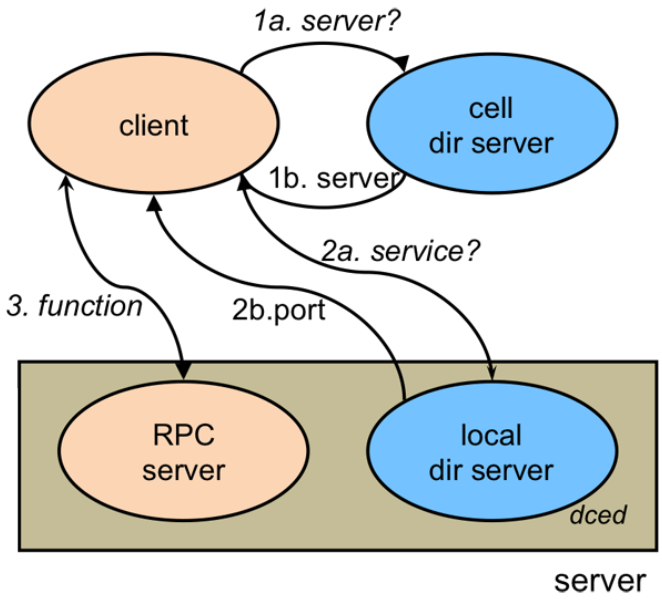

Figure 4: Figure 4. Function lookup in DCE RPC

With Sun RPC, a server only registers its {program number to port mapping} with a local name server (*portmapper*). Under DCE, a server registers its endpoint (port) on the local machine with the RPC daemon (name server) and also registers its {program name to machine} mapping with its cell directory server. When a client wants to establish communication with an RPC server, it first asks its cell directory server to locate the machine on which the server resides. The client then talks to the RPC daemon on that machine to get the port number of the server process. DCE also supports more complicated searches that span cells.

DCE RPC defines a network encoding for marshaled messages called **NDR**, for **Network Data Representation**. Unlike a single specification for representing various data types, NDR supports a **multi-canonical** format. Several encoding methods are allowed and the client can pick whichever one it chooses, one that ideally will not require it to convert from native types. If this differs from the server's native data representation, the server will still have to convert but a multi-canonical format avoids the case of having both the client and server convert into an external format when they both share the same native formats. For example, if a standard dictates a big endian network data format but both the client and server support little endian, the client has to convert each datum from little endian to big endian and the server, upon receiving the message, has to convert each datum back into little endian. A multi-canonical network data representation will allow the client to send a

network message containing data in little endian format.

9 Second generation RPCs: object support

As object oriented languages began to gain popularity in the late 1980's, it was evident that both the Sun (ONC) and DCE RPC systems did not provide any support for instantiating remote objects from remote classes, keeping track of instances of objects, or providing support for polymorphism [6]. The existing RPC mechanisms still functioned but they did not support object oriented programming techniques in an automated, transparent manner.

9.1 Microsoft DCOM (COM+)

In April 1992, Microsoft released Windows 3.1 which included a mechanism called OLE (Object Linking and Embedding). This allowed a program to dynamically link other libraries to allow facilities such as embedding a spreadsheet into a Word document (this was not a Microsoft innovation; they were catching up to a feature already offered by Apple). OLE evolved into something called COM (Component Object Model). A COM object is a binary file. Programs that use COM services have access to a standardized interface for the COM object but not its internal structures. COM objects are named with globally unique identifiers (GUIDs) and classes of objects are identified with class IDs. Several methods exist to create a COM object (e.g., *CoGetInstanceFromFile*). The COM libraries look up the appropriate binary code (a DLL[7] or executable) in the system registry, create the object, and return an interface pointer to the caller.

DCOM (Distributed COM) was introduced with Windows NT 4.0 in 1996 and is an extension of the Component Object Model to allow objects to communicate between machines. It was later renamed to COM+. Since DCOM is meant to support access to remote COM objects, a process that needs to create an object would need to supply the network name of the server as well as the class ID. Microsoft provides a couple of mechanisms for accomplishing this. The most transparent is to have the remote machine's name fixed in the registry (or DCOM class store), associated with the particular class ID. This way, the application is unaware that it is accessing a remote object and can use the same interface pointer that it would for a local COM object. Alternatively, an application may specify a machine name as a parameter.

A DCOM server is capable of serving objects at runtime. A service known as the **Service Control Manager** (**SCM**), part of the DCOM library, is responsible for connecting to the server-side SCM and requesting the creating of the object on the server. On the server, a **surrogate process** is responsible for loading components and running them. This differs from RPC models such as ONC and DCE in that a service for a specific interface was not started a priori. This surrogate process is capable of handling multiple clients simultaneously.

To support the identification of specific instances of a class (individual objects), DCOM provides an object naming capability called a **moniker**. Each instance of an object can create its own moniker and pass it back to the client. The client will then be able to refer to it later or pass the moniker to other processes. A moniker itself is an object. Its *IMoniker*

[6] Polymorphism means that you can create multiple functions with the same name but that take different parameters. A function *foo(int)* is different code from a funciton *foo(char)*.

[7] a DLL is a dynamically linked library. That is, it is a library that is linked into the process at run time rather than statically into the program during compilation and linking.

interface can be used to locate, activate, and access the bound ob-ject without having any information about where the object is located.

Several types of monikers are supported:

File moniker This moniker uses the file type (e.g., ".doc") to determine the appropriate object (e.g., Microsoft Word). Microsoft provides support for persistence: storing an object's data in a file. If a file represents a stored object, DCOM will use the class ID in the file to identify the object.

URL moniker This abstracts access to URLs via Internet protocols (e.g. http, https, ftp) in a COM interface. Binding a URL moniker allows the remote data to be retrieved. Internally, the URL moniker uses the WinInet API to retrieve data.

Class moniker This is used together with other monikers to override the class ID lookup mechanism.

Beneath DCOM: ORPC

Microsoft DCOM on its own does not provide remote procedure call capabilities. It is a set of libraries that assumes that RPC is available to the system. Beneath DCOM is Microsoft's RPC mechanism, called **Object RPC** (**ORPC**). This is a slight extension of the DCE RPC protocol with additions to support an **Interface Pointer Identifier** (**IPID**), versioning information, and extensibility information. The Interface Pointer Identifier is used to identify a specific instance of a class where the call will be processed. It also provides the capability for referrals – remote object references (IPIDs) can be passed around.

The marshaling mechanism is the same **Network Data Representation** (**NDR**) as in DCE RPC with one new type representing a marshaled interface (IPID). To do this marshaling, DCOM (and ORPC) needs to know methods, parameters, and data structures. This is obtained from an interface definition language,called **MIDL**, for **Microsoft Interface Definition Language**. As might be expected, this is identical to DCE's IDN with extensions for defining objects. The MIDL files are compiled with an IDL compiler that generates C++ code for marshaling and unmarshaling. The client side is called the proxy and the server side is called the stub. Both are COM objects that can be loaded by the COM libraries as needed.

Remote reference counting

A major difference for a server between object oriented programming and function-based programming is that functions persist while objects get instantiated and deleted from their classes (the code base remains fixed but data regions are allocated each time an object is created). For a server, this means that it must be prepared to create new objects and know when to free up the memory (destroy the objects) when the objects are no longer needed (there are no more references left to the object).

Microsoft DCOM does this explicitly rather than automatically. Object lifetime is controlled by **remote reference counting**. A call is made to **RemAddRef** when new reference to an object is added and to **RemRelease** when a reference is removed. The object itself is deleted on the server when the reference count reaches zero.

This mechanism works fine but is not foolproof. If a client terminates abnormally, no messages were sent to decrement the reference count on objects that the client was using. To handle this case, the server associates an expiration time for each object and relies on

periodic messages from the client to keep the object reference alive. This mechanism is called **pinging**. The server maintains a ping frequency (**pingPeriod**) and a timeout period (**numPingsToTimeOut**). The client runs a background process that sends a ping set: the IDs of all remote objects on a specific server. If the timeout period expires with no pings received, all references are cleared.

DCOM summary

Microsoft DCOM is a significant improvement over earlier RPC systems. The Object RPC layer is an incremental improvement over DCE RPC and allows for object references. The DCOM layer builds on top of COM's access to objects (via function tables) and provides transparency in accessing remote objects. Remote reference management is still somewhat problematic in that it has to be done explicitly but at least there is a mechanism for supporting this. The moniker mechanism provides a COM interface to support naming objects, whether they are remote references, stored objects in the file system, or URLs. The biggest downside is that DCOM is a Microsoft-only solution. It also doesn't work well across firewalls (a problem with most RPC systems) since the firewalls must allow traffic to flow between certain ports used by ORPC and DCOM.

9.2 CORBA

Even with DCE fixing some of the shortcomings in Sun RPC, certain deficiencies still remain. For example, if a server is not running, a client cannot connect to it to call a remote procedure. It is an administrator's responsibility to ensure that the needed servers are started before any clients attempt to connect to them. If a new service or interface is added to the system, there is no means by which a client can discover this. In some environments, it might helpful for a client to be able to find out about services and their interfaces at runtime. Finally, object oriented languages expect polymorphism in function calls (the function may behave differently for different types of data). Traditional RPC has no support for this.

CORBA (**Common Object Request Broker Architecture**) was created to address these, and other, issues. It is an architecture created by an industry consortium of over 300 companies called the Object Management Group (OMG). The specification for this architecture has been evolving since 1989. The goal is to provide support for distributed heterogeneous object-oriented applications. Objects may be hosted across a network of computers (a single object is not distributed). The specification is independent of any programming language, operating system, or network to enable interoperability across these platforms.

Under CORBA, when a client wishes to invoke an operation (method) on an object, it makes a request and gets a response. Both the request and response pass through the object request broker (ORB). The ORB represents the entire set of interface libraries, stub functions, and servers that hide the mechanisms for communication, activation, and storage of server objects from the client. It lets objects discover each other at run time and invoke services.

When a client makes a request, the ORB:

- Marshals arguments (at the client).
- Locates a server for the object. If necessary, it creates a process on the server end to handle the request.

- If the server is remote, transmits the request (using RPC or sockets).
- Unmarshals arguments into server format (at the server).
- Marshals return value (at the server
- Transmits the return if the server is remote.
- Unmarshals results at client.

An Interface Definition Language (IDL) is used to specify the names of classes, their attributes, and their methods. It does not contain the implementation of the object. An IDL compiler generates code to deal with the marshaling, unmarshaling, and ORB/network interactions. It generates client and server stubs. The IDL is programming-language neutral. Bindings exist for many lanugages, including for C, C++, Java, Perl, Python, Ada, COBOL, Smalltalk, Objective C, and LISP. A sample IDL is shown below:

```
Module StudentObject {
    Struct StudentInfo {
        String name;
        int id;
        float gpa;
    };
    exception Unknown {};
    interface Student {
        StudentInfo getinfo(in string name)
            raises(unknown);
        void putinfo(in StudentInfo data);
    };
};
```

IDL data types include:

- Basic types: long, short, string, float, ...
- Constructed types: struct, union, enum, sequence
- Typed object references
- The *any* type: a dynamically typed value

Programming is most commonly accomplished via object reference and requests: clients issue a request on a CORBA object using the object reference and invoking the desired methods within it. For example, using the IDL shown above, one might have code such as:

```
Student st = ... // get object reference
try {
    StudentInfo sinfo = st.getinfo("Fred Grampp");
} catch (Throwable e) {
    ... // error
}
```

Beneath the scenes, the IDL compiler generates stub functions. This code results in a call to a stub function, which then marshals parameters and sends them to the server. The client and server stubs can be used only if the name of the class and method is known at compile time. Otherwise, CORBA supports dynamic binding: assembling a method invocation at run time via the **Dynamic Invocation Interface** (**DII**). This interface provides calls to set the class, build the argument list, and make the call. The server counterpart (for creating a server interface dynamically) is called the **Dynamic Skeleton Interface** (**DSI**)[8]. A client can discover names of classes and methods at run time via the interface repository. This is a name server that can be queried to discover what classes a server supports and which objects are instantiated.

CORBA standardized the functional interfaces and capabilities but left the actual implementation and data representation formats to individual ORB vendors. This led to the situation where one CORBA implementation might not necessarily be able to communicate with another. Applications generally needed some reworking to move from one vendor's CORBA product to another.

In 1996, CORBA 2.0 added interoperability as a goal in the specification. The standard defined a network protocol called **IIOP** (the **Internet Inter-ORB Protocol**) which would work across any TCP/IP based CORBA implementations. In fact, since there was finally a standardized, documented protocol, IIOP itself could be used in systems that do not even provide a CORBA API. For example, it could be used as a transport for an implementation of Java RMI (RMI over IIOP).

The hope in providing a well-documented network protocol such as IIOP along with the full-featured set of capabilities of CORBA was to usher in a wide spectrum of diverse Internet services. Organizations can host CORBA-aware services. Clients throughout the Internet will be able to query these services, find out their interfaces dynamically, and invoke functions. The pervasiveness of these services could be as ubiquitous as HTML web access. This did not quite happen. HTTP-based web services won out.

CORBA summary

Basically, CORBA builds on top of earlier RPC systems and offers the following additional capabilities:

- Static or dynamic method invocations (RPC only supports static binding).
- Every ORB supports run time metadata for describing every server interface known to the system.
- An ORB can broker calls within a single process, multiple processes on the same machine, or distributed processes.
- Polymorphic messaging: an ORB invokes a function on a target object. The same function may have different effects depending on the type of the object.
- Automatically instantiate objects that are not running
- Communicate with other ORBs.

[8]Some systems use the term stub to refer to the client stub and skeleton to refer to the server stub. CORBA is one of them. Microsoft uses the term proxy for the client stub and stub for the server stub.

- CORBA also provides a comprehensive set of services (known as COS, for COrba Services) for managing objects:
- Life-Cycle Services: provides operations for creating, copying, moving, and deleting components.
- Persistence Service (externalization): provides an interface for storing components on storage servers.
- Naming Service: allows components to locate other components by name.
- Event Service: components can register/unregister their interest in specific events.
- Concurrency Control Services: allows objects to obtain locks on behalf of transactions.
- Transaction Service: provides two-phase commit coordination (more on this protocol later).
- Query Service: allows query operations on objects.
- Licensing Service: allows metering the use of components.
- Properties Service: allows names/properties to be associated with a component.

The price for the capabilities and flexibility is complexity. While CORBA is reliable and provides comprehensive support for managing distributed services, deploying and using CORBA generally has rather steep learning curve. Integrating it with languages is not always straightforward. Unless one could really take advantage of CORBA's capabilities, it is often easier to use a simpler and less powerful system to invoke remote procedures. CORBA enjoys a decent level of success, but only in pools of users rather than an Internet-wide community. CORBA suffered in being late to standardize on TCP/IP-based protocols and deploying Internet-based services.

9.3 Java RMI

CORBA aims at providing a comprehensive set of services for managing objects in a heterogeneous environment (different languages, operating systems, networks). Java, in its initial inception, supported the downloading of code from a remote site but its only support for distributed communication was via sockets. In 1995, Sun (the creator of Java) began creating an extension to Java called Java RMI (Remote Method Invocation). Java RMI enables a programmer to create distributed applications where methods of remote objects can be invoked from other Java Virtual Machines (JVMs).

A remote call can be made once the application (client) has a reference to the remote object. This is done by looking up the remote object in the naming service (the RMI registry) provided by RMI and receiving a reference as a return value. Java RMI is conceptually similar to RPC butt supports the semantics of object invocation in different address spaces.

One area in which the design of Java differs from CORBA and most RPC systems is that RMI is built for Java only. Sun RPC, DCE RPC, Microsoft's COM+ and ORPC, and CORBA are designed to be language, architecture, and (except for Microsoft) operating system independent. While those capabilities are lost, the gain is that RMI fits cleanly into the language and has no need for standardized data representations (Java uses the same byte ordering everywhere). The design goals for Java RMI were that it should:

- Fit the language, be integrated into the language, and be simple to use.
- Support seamless remote invocation of objects.
- Support callbacks from servers to applets.
- Preserve safety of the Java object environment.
- Support distributed garbage collection.
- Support multiple transports.

The distributed object model is similar to the local Java object model in the following ways:

1. A reference to an object can be passed as an argument or returned as a result.
2. A remote object can be cast to any of the set of remote interfaces supported by the implementation using the Java syntax for casting.
3. The built-in Java *instanceof* operator can be used to test the remote interfaces supported by a remote object.

The object model differs from the local Java object model in the following ways:

1. Classes of remote objects interact with remote interfaces, never with the implementation class of those interfaces.
2. Non-remote arguments to (and results from) a remote method invocation are passed by copy, not by reference.
3. A remote object is passed by reference, not by copying the actual remote implementation.
4. Clients must deal with additional exceptions.

Interfaces and classes

All remote interfaces extend the interface java.rmi.Remote. For example:

```
public interface bankaccount extends Remote
{
    public void deposit(float amount)
        throws java.rmi.RemoteException;

    public void withdraw(float amount)
        throws OverdrawnException,
        java.rmi.RemoteException;
}
```

Note that each method must declare java.rmi.RemoteException in its *throws* clause. This exception is thrown at the client by the RMI system whenever the remote method invocation fails. One then creates an implementation for these remote interfaces.

Remote Object Class

Not visible to the programmer, the java.rmi.server.RemoteObject class provides remote semantics of Object by implementing the *hashCode*, *equals*, and *toString*[9]. The functions needed to create objects and make them available remotely are provided by java.rmi.server.RemoteServer and subclasses. The java.rmi.server.UnicastRemoteObject class defines a unicast (single) remote object whose references are valid only while the server process is alive.

Stubs

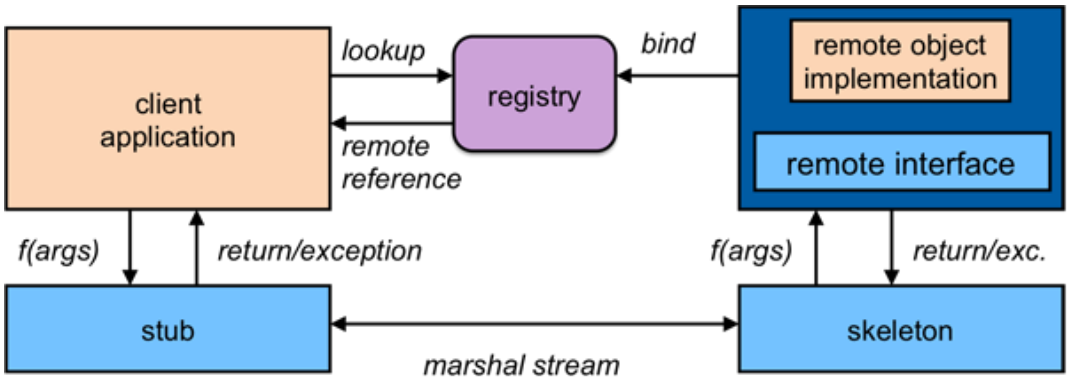

Figure 5: Figure 5. Java RMI flow

Java RMI works by creating stub functions. The stubs are generated with the **rmic** compiler. Since Java 1.5, Java supports the dynamic generation of stub classes at runtime. The compiler, *rmic*, is still present and supported and provides various compilation options.

Locating objects

A bootstrap name server is provided for storing named references to remote objects. A remote object reference can be stored using the URL-based methods of the class java.rmi.Naming. For example,

```
BankAccount acct = new BankAcctImpl();
String url = "rmi://java.sun.com/account";
// bind url to remote object
java.rmi.Naming.bind(url, acct);

// look up account
acct = (BankAccount)java.rmi.Naming.lookup(url);
```

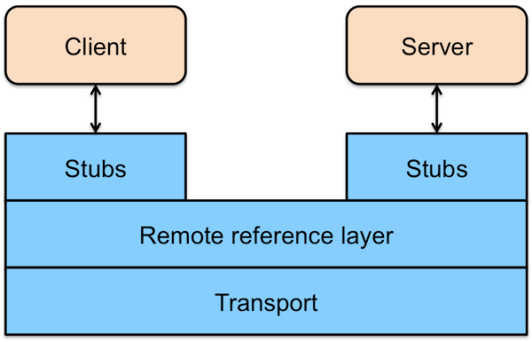

Figure 6: Figure 6. Java RMI logical view

RMI architecture

RMI is a three-layer architecture (Figure 6). The top layer is the *stub/skeleton layer*. It transmits data to the *remote reference layer* via marshal streams. Marshal streams employ a method called **object serialization**, which enables objects to be transmitted between address spaces (passing them by copy, unless they are remote objects which are passed by reference). Any class that will be used as a parameter to remote methods must implement the serializable interface. This ensures that the object data can be converted (*serialized*) into a byte stream (marshaled) for transport over the network. Serialization is a core aspect of marshaling: converting data into a stream of bytes so that it can be sent over a network or stored in a file or database.

The client stub performs the following steps:

1. initiates a call to remote object
2. marshals arguments
3. informs remote reference layer that the call should be invoked
4. unmarshals return value or exception
5. informs remote reference layer that the call is complete.

The server stub (skeleton):

1. unmarshals arguments
2. makes up-call to the actual remote object implementation
3. marshals the return value of the call (or exception)

The stub/skeleton classes are determined at run time and dynamically loaded as needed.

The remote reference layer deals with the lower-level transport interface. It is responsible for carrying out a specific remote reference protocol that is independent of the client stubs and server skeletons.

Each remote object implementation chooses its own remote reference subclass. Various protocols are possible. For example:

[9]the toString method returns the reference of the object as a string.

- Unicast point-to-point
- Invocation to replicated object groups
- Support for a specific replication strategy
- Support for a persistent reference to a remote object (enabling activation of the remote object)
- Reconnection strategies

The RMI transport layer is the transport-specific part of the protocol stack. It:

- Sets up connections, manages connections
- Monitors connection liveness
- Listens for incoming calls
- Maintains a table of remote objects that reside in the address space
- Sets up a connection for an incoming call
- Locates the target of a remote call and passes the connection to the dispatcher.

RMI distributed garbage collection

RMI creates a distributed environment where a process running on one Java Virtual Machine (JVM) can access objects that are resident on a process running on a different JVM (and quite possibly a different system). This means that a server process needs to know when an object is no longer referenced by a client and can be deleted (garbage collected). Within a JVM, Java uses reference counting and schedules objects for garbage collection when the reference count goes to zero. Across JVMs, with RMI, Java supports two operations: **dirty** and **clean**. The local JVM periodically sends a **dirty** call to the JVM of the server when an object is still in use. This *dirty* call is re-issued periodically based on the lease time given by the server. When a client has no more local references to the remote object, it sends a **clean** call to the server. Unlike DCOM, the server does not have to count per-client usage of the object and is simply informed when an object is no longer needed. If it does not receive either *dirty* or *clean* message before the lease time for the object expires, the object is then scheduled for deletion.

10 Third generation RPCs and Web Services

As use of the Internet skyrocketed after the introduction of the web browser, the browser became the dominant model for accessing information. Much of the design priority was to provide user access via a browser rather than programmatic access to accessing or manipulating the data.

Programmers craved remotely hosted services that programs rather than users can use. Web pages were designed to focus on the presentation of content. Parsing out the presentation aspects is often cumbersome. Traditional RPC solutions can certainly work over the Internet but a problem with them was that they usually relied on dynamic port assignment (contact a name server to find out what port a server offering a specific set of interfaces

is using). This was at odds with best practices of firewall configuration, which restricts available ports and may even inspect the protocol to ensure, for example, that HTTP traffic is indeed in a valid HTTP format.

Web services emerged as a set of protocols that allow services to be published, discovered, and used in a technology-neutral form. That is, the service should not be dependent on the client's language, operating system, or machine architecture.

A common implementation of web services is to use the web server as a conduit for service requests. Clients access the service via an HTTP protocol sent to a web server on the server. The web server is configured to recognize either part of the URL pathname or filename suffix and pass the request to a specific browser plug-in module. This module can then strip out the headers, parse the data (if needed), and call any other functions or modules as needed. A common example of this flow is a browser's support for Java servlets, where HTTP requests are forwarded to a JVM running a user's server code.

10.1 XML-RPC

XML-RPC was designed in 1998 as an RPC messaging protocol for marshaling procedure requests and responses into human-readable XML. The XML format uses explicit typing and is transported over the HTTP protocol, which alleviates the traditional enterprise firewall issues of having to open additional ports for RPC server applications.

An example of an XML-RPC message is:

```
<methodCall>
    <methodName>
        sample.sumAndDifference
    </methodName>
    <params>
        <param><value><int> 5 </int></value></param>
        <param><value><int> 3 </int></value></param>
    </params>
</methodCall>
```

This is a request to run the method named *sample.sumAndDifference* with two integer parameters: 5 and 3. The basic data types supported by XML-RPC are: *int*, *string*, *boolean*, *double*, and *dateTime.iso8601*. In addition, a *base64* type encodes arbitrary binary data and *array* and *struct* allow one to define arrays and structures, respectively.

XML-RPC was not designed with any specific langauge in mind and was not designed with a complete package of software for handling remote procedures. Stub generation, object management, and service lookup are not in the protocol. There are now libraries for many different languages, including Apache XML-RPC for Java, python, and perl. some of these provide a very clean procedure-like interface (e.g., python and perl).

XML-RPC is a simple spec (around seven pages long) without ambitious goals: it only focuses on messaging and does not deal with garbage collection, remote objects, name service, and other aspects of remote procedures. Nevertheless, even without broad industry support, the simplicity of the protocol led to widespread adoption of this protocol.

10.2 SOAP

The XML-RPC specification was used as a basis for creating **SOAP**, which used to stand from the *Simple Object Access Protocol*. The acronym has since been dropped since the protocol is no longer simple and is not limited to accessing objects. The protocol was created in 1998 by a consortium of organizations with strong support from Microsoft and IBM. The latest version is from 2007. SOAP specifies an XML format for stateless message exchange and includes RPC-style procedure call invocations as well as more generalized messaging that may include multiple responses.

It continues where XML-RPC left off, offering user-defined data typs, the ability to specify a recipient, specify different types of messaging, and many more capabilities. SOAP is mostly implemented as XML messages over HTTP. Like XML-RPC, SOAP is a messaging format. It does not define garbage collection, object references, stub generation, or even a transport protocol.

SOAP became one of the popular building blocks for web services. However, it just provided a standardized messaging structure. To complement it, one needs a way to define services so that the proper SOAP messages could be created. **WSDL**, the **Web Services Description Language**, serves as a method of describing web services. It is an XML document that can be fed to a program that will generate software that will send and receive SOAP messages. Essentially, WSDL serves as an interface definition language that will generate stubs (although, depending on the language, they may or may not be as transparent to the programmer as a function call).

A WSDL document comprises four parts:

1. **Types**: defines the data types used by the web service.
2. **Messages**: describes the data elements, or parameters, used by messages.
3. **Port Type**: describes the operations offered by the service. This includes the operations and input and output messages (part 2) that each operation uses.
4. **Binding**: defines the message format and protocol details for each port. For example, it may define RPC-style messaging. Note that the term *port* here is an abstract messaging endpoint (defined in part 3), not a TCP or UDP port number.

Even though they use XML, neither SOAP nor WSDL are intended for human consumption. Typically, a programmer offering a service would create an interface definition in some native language, such as Java, and use a tool such as *Java2WSDL* or *wsdl.exe* to create a WSDL document. A programmer that wants to use a service would take the WSDL document and run it through a program such as *WSDL2Java* (an apache Exclipse plug-in) or *wsdllib* (python) to generate template code.

10.3 Microsoft .NET Remoting

Microsoft's COM+ (DCOM) mechanisms were originally designed for embedding objects dynamically in a program and were always a somewhat low-level implementation. Programs often had to provide reference counting explicitly and support across languages and libraries was uneven. For example, Visual Basic provided great support for the framework while Visual C++ required considerably more programming. Microsoft overhauled and ehnaced these mechanisms with **.NET**.

Microsoft's .NET consists of a development framework and runtime environment to create platform independent applications. .NET API provides a set of classes for network-aware object-oriented programming. The Common Intermediate Language (CIL) is a stack-based, low-level, object-oriented assembly language that is machine independent and ensures interoperability between different languages and systems that support the .NET platform. The Common Language Runtime (CLR) is an ECMA standard. It describes instruction set, base classes, and execution environment of the .NET framework. It provides a set of common features that include object lifetime management, garbage collection, security, and versioning. Compilers that support .NET (such as Visual Studio C++, C#, and Visual Basic; approximately 40 languages have now some level of .NET support) generate CIL code, which is then compiled for deployment in DLL and EXE files that conform with the portable executable (PE) format. When these files are run by the CLR, it first translates the CIL into native machine code as execution occurs.

.NET Remoting is the component of .NET that deals with remote objects. It allows one to invoke methods on remote objects. To invoke a method on a remote object, the remote object must be derived from *MarshalByRefObject*. This is similar to Java's *remote* class and ensures that tha remote rererence for the object can be created. A proxy (stub) is created When the object is activated and the CLR intercepts all calls to the object. The CLR is aware of which classes are remote so that it can handle the case of a process requesting a new remote object versus a local object. Any objects that are passed as parameters have to implement the *iSerializable* interfaces, which is analagous to Java's *serializable* class. This ensures that the object's data can be converted to a flat network-friendly format: an array of bytes that can be sent in a network message.

Object creation falls into two categories: server activated objects and client activated objects. A **server activated object** does not rely on the client for controlling the lifetime of an object. Unlike with client activated objects, the client does not create a new object, use it, and then delete it. There are two forms of server activated objects. A **single call** object is completely stateless. A new instance of the object is created per call. This operation is triggered the client invokes a method on the object. A **singleton** object persists continuously. The same instance of the object is used for all client requests. The server creates the object only if the object does not yet exist. From that point onward, all client requests share the object. A **client activated object** is created on the server when the client requests a new object (e.g., via a *new* operation). It is similar to the COM+ model and supports multiple references to the same object and multiple objects created by different clients from the same class. It also supports distributed garbage collection.

Leasing Distributed Garbage Collector

.NET Remoting uses a **Leasing Distributed Garbage Collector** (**LDGC**) for client activated objects. Garbage collection is not needed for server activated objects because the object either lasts for the duration of a method call (single call) or forever (singleton). A **Lease Manager** manages object leases at a server. The server object is considered to be in use as long as its lease has not expired. If a client wants to be contacted when a lease expires, it must provide a **sponsor** object. The client's sponsor object can then choose to extend the lease if necessary.

The garbage collector uses the following parameters for an object:

- InitialLeaseTime: initial lifetime of the remote object (5 min default)
- LeaseManagePollTime: interval at which lease manager polls leases10 sec default)

- sponsorshipTimeOut: amount of time the framework waits for a sponsor to become available (2 min default)

Each time a method is called, the lease time is set to the *maxium(lease time - expired time, RenewOnCallTime)*. The *renewOnCallTime* is set to the amount of time to renew lease after each method call. The requestor (client) has to renew its lease when *leaseTime* elapses. This lease-based approach means that the server does not have to do reference counting as it did with COM+.

The .NET Remoting component was designed for a homogenous environment: to communicate among .NET processes. While .NET has support for Web services, .NET remoting was not designed specifically for it. An extension, called **Web Services Enhancement** (**WSE**), was designed to handle SOAP-based web services. WSE supports the following alphabet soup of protocols and formats:

- **HTTP**: *HyperText Transfer Protocol*, the communications protocol running over TCP/IP
- **XML**: *eXtended Markup Language*, the data representation format
- **SOAP**: *Simple Object Access Protocol*, the protocol for sending and receiving messages and invoking functions
- **WSDL**: *Web Services Definition Language*, the XML document that defines the availabile services and their interfaces
- **UDDI**: *Universal Description, Discovery, & Integration*, the protocol for discovering services

Compared with .NET, SOAP is just the lower-level messaging protocol. To create a .NET web service, one Writes a .NET object as if it were accessed by local clients and marks it with an attribute that it should be available to Web clients. ASP.NET, the extension to Microsoft's web server (IIS) provides the framework that accepts HTTP requests and maps them to object calls. The WSDL document describing the available service is generated automatically by examining metadata in .NET object.

The evolution of .NET facilities is the **Windows Communication Framework** (**WCF**), which was yet another overhaul to provide a unifying communication framework but also to support interoperability with other platforms. It still uses the .NET CLR environment and supporting libraries.

With WCF, the **service** is implemented in a **Service Class**, which implements one or more methods and contains a *Service Contract*, which defines the methods that a client can use, and a *Data Contract*, which defines the data structures. The service class is compiled to a CLR-based language. The service class is typically compiled into a library and runs in a **host process**. The host process can be one of:

- IIS (web server) host process (the communication protocol must use SOAP over HTTP)
- Windows Activation Service
- An arbitrary process

The **endpoint definitions** for a service specify:

- An **address**: the URL for the service
- The **binding**: the description of the protocols and security mechanisms
- The contract**: the name indicating which service contract this endpoint exposes

The Windows Communication Framework offers a set of possible bindings (communication options):

Binding	Description
BasicHttpBinding	SOAP over HTTP (optional HTTPS)
WsHttpBinding	Same but with support for reliable message transport, security, and transactions
NetTcpBinding	Binary-encoded SOAP over TCP, with support for reliable message transport, security, and transactions
WebHttpBinding	HTTP or HTTPS with no SOAP – ideal for RESTful communication; data in XML, JSON, or binary
NetNamedPipesBinding	Binary-encoded SOAP over named pipes (not multi-platform only for WCF-WCF communication)
NetMsmqBinding	Binary-encoded SOAP over MSMQ (also not multi-platform; WCF-WCF only)

10.4 Java API for XML Web Services

Java RMI was designed for interacting with remote objects but was built primarily with a Java model in mind. Moreover, it was not built with web services and HTTP-based messaging in mind. A lot of software has emerged to support Java-based web services. We will look at just one of them in this overview. **JAX-WS** (**Java API for XML Web Services**) is designed as a Java API for web service messaging and remote procedure calls. It allows one to invoke a Java-based web service using Java RMI (i.e., relatively transparently for the programmer). A goal of JAX-WS is platform interoperability. The API uses SOAP and WSDL. A Java environment is not required on both sides and the service definition is available to clients as a WSDL document.

Creating an RPC endpoint

On the server side, the following steps take place to create an RPC endpoint (that is, an RPC service that is available to clients):

1. Define an interface (Java interface)
2. Implement the service
3. Create a publisher, which ceates an instance of the service and publishes it with a name

On the client:

1. Create a proxy (client-side stub). The **wsimport** command takes a WSDL document and creates a client stub

2. Write a client that creates an instance of the remote service via the proxy (stub) and invokes methods on it

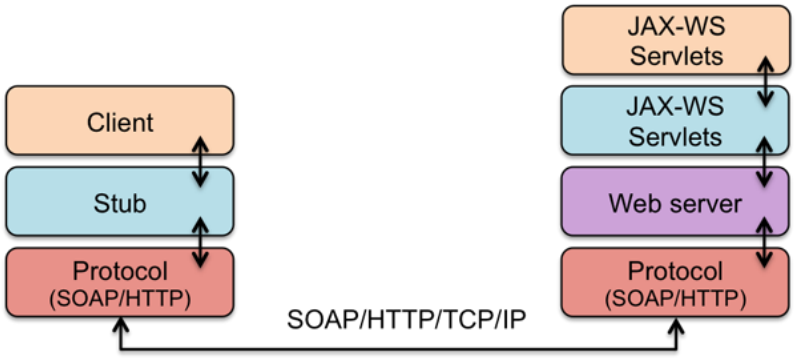

Figure 7: Figure 7. JAX-WS call flow

The execution flow used in JAX-RPC follows the flow of just about every other RPC system with the exception that the framework is usually hosted by the server's web server. It is shown in Figure 7:

1. The Java client calls a method on a stub (proxy)

2. The stub calls the appropriate web service

3. The web server gets the call and directs it to the JAX-WS framework

4. The framework calls the implementation

5. The implementation returns results to the framework

6. The framework returns the results to the web server

7. The server sends the results to the client stub

8. The client stub returns the information to the caller

10.5 Beyond SOAP

While SOAP was, and continues to be, widely deployed, many environments dropped it in favor of mechanisms that are either lighter weight, easier to understand, or fit the web interaction model more cleanly. For example, Google dropped support of SOAP interfaces to their APIs in 2006, offering alternatives of AJAX, XML-RPC, and REST. An anonymous Microsoft employee criticized SOAP for being overly complex because "we want our tools to read it, not people". Whether the quote is accurate or not does not really matter since SOAP is clearly a complex and highly-verbose format.

AJAX

One restriction of the original design of the web browser was the non-dynamic interaction model offered by web pages. The web browser was built with a synchronous equest-response interaction model. A request is sent to a server and the server returns an entire page. There was simply no good way of updating parts of a page. The only approach that was viable was to use frames (once they became available) and load a different page into each frame. Even that was clunky and restrictive. What changed this was the emergence of the Document Object Model and JavaScript, which allowed one to programmatically change various parts of a web page. The other key ingredient that was needed was to interact with the server in a non-blocking manner, allowing users to still interact with the page even if the underlying JavaScript is still waiting for results from the server. This component is called AJAX.
AJAX stands for **Asynchronous JavaScript And XML**. Let's examine each of those three items:

- It is *asynchronous* because the client not blocked while waiting for results from the server.
- AJAX is integrated into JavaScript and designed to be invoked by browsers as part of interpreting a web page. AJAX requests are invoked from JavaScript using *HTTPRequest*. The JavaScript may also modify the Document Object Model that defines how the page looks
- Data is sent and received as XML documents.

AJAX ushered in what became known as Web 2.0: highly interactive services such as Google Maps, Writely, and many others. Basically, it allows JavaScript to issue HTTP requests, get and process results, and elements on the page without refreshing the entire page. In most browsers[10], the request is of the format:

```
new XMLHttpRequest()
xmlhttp.open(‘‘HEAD’’, ‘‘index.html’’, true)Tell object:
```

The JavaScript code tells the XMLHttpRequest object the type of request you're issuing, the URL you are requesting, the function to call back when request is made, and information to send along in the body of request

Note that AJAX is far removed from RPC. It does not provide a functional interface to server functions. It is also designed specifically for web browsers.

10.6 REST

While SOAP created its own messaging protocol that just happened to be transported over HTTP, the approach of **REST** (**REpresentational State Transfer**) is to stay with the principles of the web and use HTTP as a core part of the protocol.

The original HTTP protocol was already defined with four commands that cleanly map onto various operations that can be performed on data (a "resource"):

- PUT (insert)

[10] In Internet Explorer, the request is: new ActiveXObject("msxml3.XMLHTTP").

- GET (select)
- POST (update)
- DELETE (delete)

The idea behind REST is to use these HTTP commands to request data and effect operations on that data. As part of the HTTP protocol REST uses URLs to refer to both objects and operations, since they provide a hierachical naming format and attribute-value lists of parameters (e.g., http://mydomain.com/mydata/getlist?item=123&format=brief).

As an example of interacting with a bloc, consider these operations:

Get a snapshot of a user's blogroll: HTTP GET //rpc.bloglines.com/listsubs

HTTP authentication handles user identification and authentication

To get info about a specific subscription: HTTP GET http://rpc.bloglines.com/getitems?s={subid}

A REST model makes a lot of sense for resource-oriented services, such as Bloglines, Amazon, flikr, del.icio.us, etc.

As a further example, consider this HTTP operation to get a list of parts:

```
HTTP GET //www.parts-depot.com/parts
```

This command will return an XML document containing a list of parts. Note that what is being returned is not a web page, just an XML data structure containing the requested data.

```
<?xml version="1.0"?>
<p:Parts xmlns:p="http://www.parts-depot.com"
         xmlns:xlink="http://www.w3.org/1999/xlink">
      <Part id="00345" xlink:href="http://www.parts-depot.com/parts/00345"/>
      <Part id="00346" xlink:href="http://www.parts-depot.com/parts/00346"/>
      <Part id="00347" xlink:href="http://www.parts-depot.com/parts/00347"/>
      <Part id="00348" xlink:href="http://www.parts-depot.com/parts/00348"/>
</p:Parts>
```

To get detailed information for a specific part, one would send an HTTP GET command querying that specific part:

```
HTTP GET //www.parts-depot.com/parts/00345
```

This would then return information about a specific part:

```
<?xml version="1.0"?>
<p:Part xmlns:p="http://www.parts-depot.com"
        xmlns:xlink="http://www.w3.org/1999/xlink">
      <Part-ID>00345</Part-ID>
      <Name>Widget-A</Name>
      <Description>This part is used within the frap assembly</Description>
```

```
        <Specification xlink:href="http://www.parts-depot.com/parts/00345/specification"/>
        <UnitCost currency="USD">0.10</UnitCost>
        <Quantity>10</Quantity>
    </p:Part>
```

Note that this is an example and the parts query could just as easily have had the part number as parameter to the URL. For example

```
HTTP GET //www.parts-depot.com/parts?partid=00345
```

REST is not RPC but has an analogous request-response model. The transparency of formulating requests, marshaling data, and parsing responses is not part of REST. REST is very widely supported and used in services such as Yahoo! Search APIs, Ruby on Rails, Twiter, and Open Zing Services. The last of these is the programmatic interface to XM-Sirius Radio. A URL to get a channel list is:

```
svc://Radio/ChannelList
```

and to get information about a specific channel at a specific time, one can request

```
svc://Radio/ChannelInfo?sid=001-siriushits1&ts=2012091103205
```

10.7 Google Protocol Buffers: Just Marshaling

There are times when neither RPC nor web services are needed but a programmer just wants to simplify the task of marshaling and unmarshaling data on the network. **Google Protocol Buffers** provide an efficient mechanism for serializing structured data, making it easy to encode the data onto the network and decode received data. Protocol Buffers are a compact binary format that is much simpler, smaller, and faster than XML. They are language independent and serve only to define data types. Each message is a structured set of data names, types, and values. The message structure is defined in a high-level format, similar to many interface definition languages. This file is then compiled to generate conversion routines for your chosen language. Protocol Buffers are used extensively within Google. Currently over 48,000 different message types defined. They are used both for RPC-like messaging as well as for persistent storage, where you need to convert data into a standard serial form for writing it onto a file. An example of a protocol buffer definition, from Google's Protocol Buffers Developer Guide is:

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
```

```
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Note that this defines only data structures, not functions. A sample use of this structure is:

```
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@example.com");
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);
```

Compared to even compact XML versions, protocol buffers are far more efficient both in parsing time as well as in space. Also, from the Developer Guide, if we compare this XML version:

```
<person>
    <name>John Doe</name>
    <email>jdoe@example.com</email>
</person>
```

with this uncompiled protocol buffer:

```
person {
   name: "John Doe"
   email: "jdoe@example.com"
}
```

The binary message generated from the protocol buffer is approximately 28 bytes long and takes around 100–200 ns to parse. The XML version, by comparison, is over 69 bytes long (2.5 times larger) and takes from 5,000 to 10,000 ns to parse (50 times longer).

10.8 JSON

Anther marshaling format that has gained considerable popularity is **JSON**. This is not a binary format such as Google Protocol Buffers and is hence suitable for use as payload in a message transported via HTTP. JSON was is based on JavaScript, is human readable and writable, and easy to parse. It was introduced as the "fat-free alternative to XML".

Currently, convertors exist for 50 languages and remote procedure invocation has been added with JSON-RPC. Keep in mind that this is just a messaging format and JSON makes no attempt to offer RPC libraries and support for service discovery, binding, hosting, and garbage collection.

---

11 References (partial)

- The Component Object Model Specification, Draft version 0.9, October 24, 1995, 1992–1995 Microsoft Corp, http://www.microsoft.com/oledev/olecom/title.htm
- CORBA Architecture, version 2.1, Object Management Group, August 1997, pp 1–1 – 2–18. [introductory CORBA concepts straight from the horse's mouth]
- The OSF Distributed Computing Environment: Building on International Standards – A White Paper, Open Software Foundation, April 1992. [introduction to DCE and DCE's RFS]
- Networking Applications on UNIX System V Release 4, Michael Padovano, 1993 Prentice Hall. [guide to sockets, Sun RPC, and Unix network program-ming]
- UNIX Network Programming, W. Richard Stevens, 1990 Prentice Hall. [guide to sockets, Sun RPC, and Unix network programming]
- JAVA Remote Method Invocation (RMI), 1995–1997 Sun Microsystems, http://java.sun.com/products/jdk/rmi/index.html
- Distributed Operating Systems, Andrew Tanenbaum, 1995 Prentice Hall, pp. 68–98, 520–524,535–540. [introductory information on sockets and RPC]
- Modern Operating Systems, Andrew Tanenbaum, 1992 Prentice Hall, pp. 145- 180, 307–313, 340–346. [introductory information on sockets and RPC]