Report for seminar
Operating Systems:
Remote Procedure Call

Andrey Shadrin

October 5, 2004

Contents

Chapter 1

Introduction

A client-server model provides a convenient way to structure a distributed operating system, it suffers from one incurable flaw: the basic paradigm around which all comunication is built is input/output. The such primitives are engaged in doing I/O: *send* and *recieve*. Since I/O is not one of the key concepts of centralized systems. The goal is to make distribited computing more look centralized computing, i.e. put communication procedures into librares, from them can invoke these procedures. Building everything around I/O is not way to do it.

This problem has long been known, but little was done about it until a paper by Birrell and Nelson (1984) introduced a completely different way of attacking the problem.

In a nutshell, what Birrell and Nelson suggested was allowing programs to call procedures located on other machines. When a process on machine A calls a procedure on machine B, the calling process on A is suspended, and execution of the called procedure takes place on B. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. Neither message passing nor I/O at all is visible to the programmer. This method is known as **remote procedure call**, or often just **RPC**.

The idea of RPC is based on the observation that local procedure calls are a well-known and well-understood mechanism for transfer of control and data within a program running on a single computer. Therefore, it is proposed that this same mechanism be extended to provide for transfer of control and data across a communication network.

Major issues facing the designer of an RPC mechanism include:

- the precise semantics of a call in the presense fo machine and communication failures
- the semantics of address-containing arguments in the (possible) absense of a shared address space
- integration of remote calls into existing (or future) programming systems

- binding (how a caller determines the location and identity of the calle)
- suitable protocols for transfer of data and control between caller and calle
- how to provide data integrity and security (in an open network)

While the basic idea sounds simple and elegant, subtle problems exist. To start with, because the calling and called procedures run on different machines, they execute in different adddress spaces, which causes problems. Parameters and result aslo have to be passed, which can be complicated, especially if the machines are not identical. Finally, both machines can crash, and each of possible failures causes different problems. Still, most of these can be dealt with, and RPC is a widely-used techique that underlies many *distributed operating systems.*

Chapter 2

Basic RPC

To understand how RPC works, it is important first to fully understand how a conventional local procedure call works. Conseder a call like

count = read(fd, buf, nbytes);

where *fd* is an integer, *buf* is an array of characters, and *nbytes* is another integer. If the call is made from the main program, the stack will be as shown in . 2.1(a) before the call. To make the call, the caller pushes the parameters onto the stack in order, last one first, as shown in Fig.2.1(b). After *read* has finished running, it puts the return valur in a register, removes the return address, and transfers control back to the caller. The caller then removes the parameters from the stack, returning it to the original state, as shown in Fig. 2.1(c).

Several things are worth noting. For one, in C, parameters can be **call-by-value** or **call-by-reference**. A value parameter, such as *fd* or *nbytes*, is simply copied to the stack as shown in Fig. 2.1(b). To the called procedure, a value parameter is just an initialized local variable. The called procedure may modify it, but such changes do not affect the original value at the calling side.

A reference parameter in C is a pointer to a variable (i.e. the address of the variable), rather than the value of the variable. In the call to *read*, the second parameter is a reference parameter because arrays are always passed by reference in C. What is actually pushed onto the stack is the address of the character array. If the called procedure uses this parameter to store something into the character array, it does modify the array in the calling procedure. The difference between call-by-value and call-by-reference is quite important for RPC, as we shall see.

One other parameter passing mechanism also exists, although it is not used in C. It is called **call-by-copy/restore**. It consists of having the variable copied to the stack by the caller, as in call-by-value, and then copied back after the call, overwritting the caller's original value. Under most conditions, this achieves the same effect as call-by-reference, but in some situations, such as the same parameter being present multiple times in the parameter list, the semantics are different.

The decision of which parameter passing mechanism to use is normally made

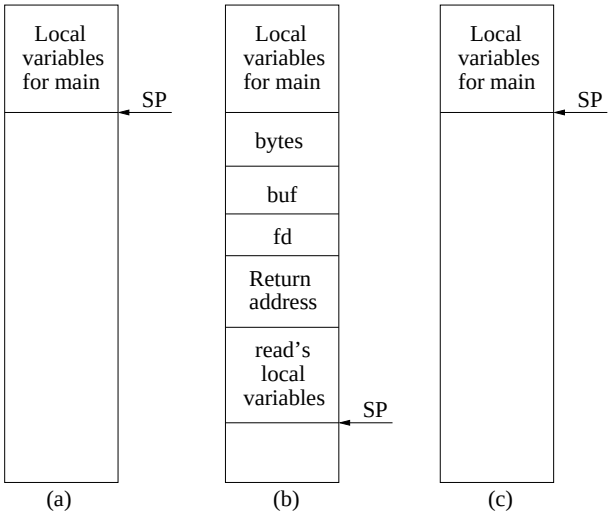

Figure 2.1: (a) The stack before the call to *read*. (b) The stack while the called procedure is active. (c) The stack after the retutn to the caller

by the language designers and is a fixed property of the language.

The idea behind RPC is to make a remote procedure call a much as possible like a local one. In other words, we want RPC to be transparent - the calling procedure should not be aware thar the called procedure is executing on a different machine, or vice versa. Suppose that a program needs to read some data from a file. The programmer puts a call to *read* in the code to get the data. In a traditional (single-processor) system, the *read* routine is extracted from the library by the linker and inserted into the object program. It is a short procedure, usually written in assembly language, that puts the parameters in registers and then issues a READ system call by trapping to the kernel.

Even though *read* issues a kernel trap, it is called in the usual way, by pushing the parameters onto the stack, as shown Fig. 2.1. Thus the programmer does not know that *read* is actually doing something fishy.

RPC achieves its transparency in an analogous way. When *read* is actually a remote procedure (e.g., one that will run on the file server's machine), a different version of *read*, called a **client stub**, is put into the library. The function of the client stub is to take its parameters, pack them into a message, and send it to the server stub. Like the original one, it too, is called using the calling sequence of Fig. 2.1. Also like the original one, it too, traps to the kernel. Only unlike the original one, it does not put the parameters in registers and ask the kernel to give it data. Instead, it packs the parameters into a message and asks the kernel to send the message to the server as illustrated in Fig. 2.2. Following the call to *send*, the client stub calls receive, blocking itself until the reply comes back.

When the message arrives at the server, the kernel passes it up to a **server**

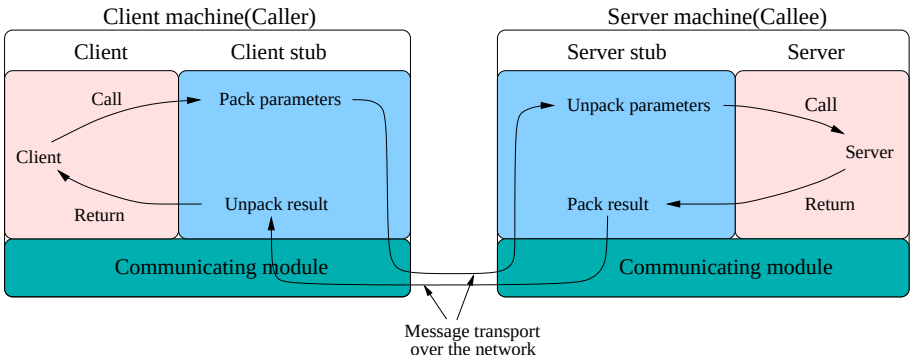

Figure 2.2: The structure of an RPC

stub that is bound with the actual server. Typically the server stub will have called *receive* and be blocked waiting for incoming messages. The server stub unpacks the parameters from the message and then calls the server procedure in the usual way (i.e., as in Fig. 2.1). From the server's point of view, it is as though it is being called dierctly by the client - the parameters and return address are all on the stack where they belong and nothing seems unusual. The server performs its work and the returns the result to the caller in the usual way.

When the server stub gets control back after the call had completed, it packs the result (the buffer) in a message and calls *send* to return it to the client. Then it goes back to the top of its own loop to call *receive*, waiting for the next message.

When the message gets back to the client machine, the kernel sees that it is addressed to the client process (to the stub part of that process, but the kernel does not know that). The message is copied to the waiting buffer and the client process unblocked. The client stub unpacks the result, copies to the caller, and returns in the usual way. When the caller gets control following the call to *read*, all it knows is that its data are available. It has no idea that the work was done remotely instead of by the local kernel.

All the details of message passing are hidden away in the two library procedures, just as the details of actually making system call traps are hidden away in traditional libraries.

To summarize, a remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way
2. The client stub builds a message and traps to the kernel
3. The kernel sends the message to the remote kernel
4. The remote kernel gives the message to the server stub
5. The server stub unpacks the parameters and calls the server

6. The server does the work and returns the result to the stub
7. The server stub packs it in a message and traps to the kernel
8. The remote kernel sends the message to the client's kernel
9. The client's kernel gives the message to the client stub
10. The stub unpacks the result and returns to the client

The net effect of all these steps is to convert the local call by the client procedure to the client stub to a local call to the server procedure without either client or server being aware of the intermediate steps.

Chapter 3

Design issues

3.1 Classes of RPC system

Many RPC systems have been built since appeared the problem about RPC. They fall into two classes:

- In the first class, the RPC mechanism is intergrated with a particular programming language, that includes a notation for defining interfaces.
- In the second class, a special-purpose **Interface Defintion Language** (here and after IDL) is used to describe the interfaces between clients and servers.

The first class includes Cedyr, Argus and Arjuna programming languages. The language integration that it achieves has the advantage that the particular requirements of remote procedures can be dealt with by language constructs such as exceptions.

The second class includes Sun RPC, on which the Sun Network File System is based. The separate interface language approach has the advantage that it is not tied to a particular language environment, although in practice, almost all examples of this approach are used in a C programming environment. That is we write two codes, the first are client and server code, and the second is interface definition file (see below).

3.2 Interface definition language

An RPC interface definition specifies those characteristics of the procedures provided by a server that are visible to the server's clients. The characterstics that must be defined include the names of the procedures and the types of their parameters. Each parameter should also be defined as input, output or in some cases both, to enable the RPC system to identify which values should be packed into the request and reply messages.

An interface contains a list of procedure *signatures* - that is, their names, together with the types of their input and output arguments. Though an interface compilers should be designed to process interfaces for use with different languages enabling clients and servers written in different languages to communicate by using remote procedure calls.

3.3 Exception handling

Any remote procedure call may fail because it may not be able to contact a server, probably because the server has failed or is too busy to reply. Therefore remote procedure calls must be able to report error types such as time-outs that are due to distribution as well as those that relate to problems encounterd in executing the procedure. The same mechanism may be used to report errors discovered by procedures.

The exception handling mechanism consist of two parts, the *raising of exceptions* and their *handling procedures*. When an error occurs in a procedure, an exception is raised and the appropriate handling procedure is automaticaly executed in the caller's environment.

Many RPC systems are designed for use with existing programming languages that have no exception handling mechanisms. In the absence of an exception handling mechanism, RPC systems generally resort to the method used in UNIX and other conventional operating systems, in which the system functions deliver a well-known a value to indicate failure and further information about the type of error is reported in a variable in the environment of the calling program. In the case of an RPC, a return value indicating an error is used both for errors due to failure to communicate with the server and errors reported in the reply message from the server. Further information about the type of error is stored in a global variable in the client program. Though this method has the disadvantage that it requires the caller to test every return value.

3.4 Delivery guarantess

The main choices to provide a deal between client and server of delivery guarantees are:

Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed

Duplicate filtering: when retransmissions are used, whether to filter out duplicates at the server

Retransmission of replies: whether to keep a history of reply messages to enable lost replies to be retransmitted without re-executing the server operations

The combinations of these choices leads to a variety of possible semantics for the reliability of remote procedure calls as seen by the caller. The choices of interest

Delivery guarantess			RPC call semantics
Retry request message	Duplicate filtering	Re−execute procedure or retnasmit reply	
No	Not applicable	Not applicable	Maybe
Yes	No	Re−execute procedure	At−least−once
Yes	Yes	Retransmit reply	At−most−once

Figure 3.1: The RPCs semantics

are shown in the Fig. 3.1, with corresponding names for the call semantics that they produce. The semantics are as follows:

Maybe call semantics: No fault tolerance measures. If the reply message has not been received after a time-out and there are no retries, it is uncertain whether the procedure has been executed. If the request message was lost or the server crashed, then the procedure will not have been executed. On the other hand, it may have been executed and the reply message lost. This known as **maybe** call semantics, because clients cannot tell for sure whether remote procedures have been called or not. It is not generally acceptable.

At-least-once call semantics: Retransmission of request messages without filtering of duplicates. The caller is eventually either given a reply, or else is informed that the server is presumed to have failed. In cases when the request message is retransmitted, the server may receive and execute it more than once. Eventually, when the RPC is completed, the client will not know how many times it has been called. This is called **at-least-once** call semantics. If a server can be designed with idempotent operations in all of its remote procedures, then at-least-once call semantics may be acceptable. This semantic is accetable, iff the remote procedures are idempotent, i.e. the execution of such procedures returns always the same result (state) (e.g. the transfer of money is not idempotent operation). If all procedures are idempotent, then we do not achive side effects after its exucution.

At-most-once call semantics: Filtering of duplicates and retransmission of replies without re-executing operations. Some operations can have the wrong effect if they are performed more than once. For example, an oper-

ation to increase a bank balance by $10 should only be performed once; if it were to be repeated, the balance could grow and grow! To allow the use of such non-idempotent operations, RPCs should be designed to execute their operations exactly once. Birrell and Nelson guarantee in Cedar RPC that if the server does not crash and the client receives the result of a call, then the procedure has been executed exactly once. Otherwise, an exception is reported and the procedure will have been called either once or not at all. This known as **at-most-once** call semantics and is the one usually chosen in RPC implementations.

Chapter 4

Implementation issues

The software that supports remote procedure calling has three main tasks:

Interface processing: integrating the RPC mechanism with client and server programs in conventional programming languages. This includes the packing and unpacking of arguments in the client and the server and the dispatching of request messages to the appropriate procedure in the server:

Communication handling: transmitting and receiving request and reply messages;

Binding: locating an apropriate server for a particular service.

4.1 Interface processing

An interface definition may be used as a basis on which to construct the extra software components of the client and server programs that enable remote procedure calling. These components are illustrated in Fig. 3.1. Both client and server assign the same unique procedure identifier to each procedure in an interface (they are usually numbered 0, 1, 2 ... in order) and the procedure identifier is included in request messages.

Building the client program: An RPC system will provide a means of building a "complete client program" by providing a **stub procedure** to stand in for each remote procedure that is called by the client program. The purpose of a client stub procedure is to convert a local procedure call to a remote procedure call to the server. The types of the arguments and results in the client stub must conform to those expected by the remote procedure. This is achieved by the use of a common interface definition. The task of a client stub procedure is to pack the arguments and to pass them with the procedure identifier into a message, send the message to the server and then await the reply message, unpack it and return the results.

Building the server program: An RPC system will provide a **despatcher** and a set of server stub procedures. The despatcher uses the procedure identifier in the request message to select one of the server stub procedures and pass on the arguments. The task of a server stub procedure is to unpack the arguments, call the appropriate service procedure, and when it returns, to marshal the output arguments (or in the case of failure an error report), into a reply message.

An interface compiler processes interface definitions written in an interface definition language. Interface compilers are designed to produce components that can be combined with client and server programs, without making any changes to the existing compilers. An interface compiler normally performs the following tasks:

1. Generate a client stub procedure to correspond to each procedure signature in the interface. The stub procedures will be compiled and linked with the client program.
2. Generate a server stub procedure to correspond to each procedure signature in the interface. The despatcher and the server stub procedures will be compiled and linked with the server program.
3. Use the signatures of the procedures in the interface - which define the argument and result types - to generate appropriate packing and unpacking operations in each stub procedure.
4. Generate procedure headings for each procedure in the service from the interface definition. The programmer of the service supplies the bodies of these procedures.

The use of a common interface definition when generating the stub procedures for the client program and the headings for the procedures in the server programs ensures that the argument types and results used by clients conform to those defined in a server.

4.2 Communication handling

The task of the communication handling module in both the client and the server programs is to deal with communication between them, generally by using a form of *request-reply* communication. The communication handling module is provided in forms suitable for linking with client and server programs. And it could use some networks protocol (e.g. connection-oriented - TCP, conectionless - UDP), or is based on these protocols with own new implemented details.

4.3 Binding

An interface definition specifies a textual service name for use by clients and servers to refer to a service. However, client request messages must be addressed

Call	Input	Output
Register	name, version, address	
Unregister	name, version, address	
Lookup	name, version	address

Figure 4.1: The binder interface

to a server port(or socket).

Binding means specifying a mapping from a name to a particular object, usually identified by a communication identifier. The binding of a service name to the communication identifier specifying the server port (whether it be a port identifier, a port group identifier, or any other form of destination) - is evaluated each time a client program is run. The form of the communication identifier depends on the environment. For example, in a UNIX environment, it will be a socket address containing the internet address of a computer and a port number.

In a distributed system, a **binder** is a separate service that maintains a table containing mappings from service names to server ports. A binder is an example of a Name Service. The Domain Name service which maps domain names into Internet addresses is another example.

A binder is intended to be used by servers to make their port identifiers known to potential clients. A typical binder would include the procedures shown in Figure 4.1. *Register* and *Unregister* are intended to be used by servers. *LookUp* is intended to be used by clients to obtain the addresses of servers.

When a server process starts executing, it sends a message to the binder requesting it to *Register* its service name and server port. If a server process terminates, it should send a message to the binder requesting it to Unregister its entry from the mappings.

When a client process starts, it sends a message to the binder requesting it to *LookUp* the identifier of the server port of a named service. The client program sends all its request messages to this server port until the server fails to reply, at which point the client may contact the binder and attempt to get a new binding.

The purpose of the version number is to enable client and server programs to check that they are using the same version of the software. If the program of a server is altered in such a way that clients will no longer be able to communicate with it, for example, requiring an extra argument to a procedure, then it must update its version number and client programs will have to be brought up to date.

If location-independent port identifiers are in use, then servers may be relocated without informing the binder and clients are unaffected by the move. However, if port identifiers include host addresses, the binder must be informed

whenever a server is relocated and clients find out when their request messages to the old location are ignored, in which case they will contact the binder and be given the identifier of the new server port.

Locating the binder. All clients and servers need to use a binder, to import and export services. Therefore they need to know the port identifier of a binder before they can do anything useful. The following alternative approaches are commonly used in systems such as UNIX in which a port identifier includes a host address:

- Always run the binder on a computer with a well known host address and compile this host address into all client programs. All client and server programs must be recompiled if the binder ever needs to be relocated.
- Make the client and server operating systems responsible for supplying the current host address of the binder at run time, for example in UNIX it may be supplied via an environment variable. Users running client and server programs need to be informed whenever the binder is relocated. This method allows occasional relocation of the binder.
- When a client or server program starts executing, it uses a broadcast message to locate the binder. For example, in UNIX, the broadcast message will specify the port number of the binder and a binder receiving such a request will reply with its current host address. The binder can be run on any computer and can easily be relocated.

Chapter 5

Case studies: Sun RPC

The best known of the UNIX RPC systems is Sun RPC [Sun 1990] which was originally designed for client-server communication in the Sun NFS network file system. Sun RPC is supplied as a part of the various Sun UNIX operating systems and is also available with other NFS installations. Implementers have the choice of using remote procedure calls over either UDP/IP or TCP/IP.

The Sun RPC system provides an interface language called **XDR** and an interface compiler called *rpcgen.*

Interface definition language. The Sun XDR language which was originally designed for specifying external data representations has been extended to become an interface definition language. It is a standard to represent a data to exchange between different architectures of machine. But it is not a programming language, we can only describe data. It may be used to specify a Sun RPC interface which contains a program number and a version number rather than an interface name, together with procedure definitions and supporting type definitions. A procedure definition specifies a procedure signature and procedure number. As only a single input parameter is allowed, procedures requiring multiple parameters must include them as components of a signle structure. The output parameters of a procedure also are returned via a single result. The procedure signature consists of the result type, the name of the procedure and the type of the input parameter.

Listing 5.1: Files interface in Sun XDR

```
/*
* FileReadWrite service interface definition in file FileReadWrite.x
*/
constMAX= 1000;

typedefint FileIdentifier;
typedefint FilePointer;
typedefint Length;

struct Data {
int length;
char buffer[MAX];
};
```

```
struct writeargs {
Fileldentifier f;
FilePointer position;
Data data;
};

struct readargs {
Fileldentifier f;
FilePointer position;
Length length;
};
```

For example, see the XDR definition of Listings 7.1 of an interface with a pair of procedures for writing and reading files. The program number is 9999 and the version number is 2. The *READ* procedure takes as input parameter a structure with three components specifying: a file identifier, a position in the file and the number of bytes required. Its result is a structure containing the number of bytes returned and the file data. The *WRITE* procedure has no result. The *WRITE* and *READ* procedures are given numbers 1 and 2. The number zero is reserved for a null procedure that is generated automatically and is intended to be used to test whether a server is available.

This IDL have almost the same syntax to define data types like in C. It provides a notation for defining constants, typedefs, structures, enumerated types, unions and programs. Typedefs, structure and enumerated types have the same sytax as in C. The interface compiler *rpcgen* can be used to generate the following from an interface definition:

- client stub procedures
- server main procedure, despatcher and server stub procedures
- XDR packing and unpacking procedures for use by the despatcher and client and server stub procedures
- a header file, for example "FileReadWrite.h", containing definitions of common constants and types that may be used in client and server programs. The service procedure signatures are given as C function prototypes. The developer of the service provides implementations of these procedures that conform to the prototypes

Binding. Sun RPC does not have a network-wide binding service. Instead it provides a local binding service called the *port mapper* which runs on every computer. It uses the fixed port number 111 for providing the service. Each instance of a port mapper records the port in use by each service running locally. Port the same for a different version of program, otherwise a port is unique for each services. Therefore to import an interface, the client must specify the hostname of the server as well as the program number and version number.

Chapter 6

The Sun Network File System

Sun Microsystem's **Network File System**, universally known as NFS. NFS was originally designed and implemented by Sun Microsystems for use on its UNIX-based workstations. Other manufacturers now support it as well, for both UNIX and other operating systems. NFS supports heterogeneous systems, it allows for some operating system clients making use of UNIX servers. It is not even required that all the machines use the same hardware.

Three aspects of NFS are of interest: the architecture, the protocol, and the implementation.

6.1 NFS Architecture

The basic idea behind NFS is to allow an arbitrary collection of clients and servers to share a common file system. In most cases, all the clients and servers are on the same LAN, but this is not required. It is possible to run NFS over a wide-area network. For simplicity we will speak of clients and server as though they were on distinct machines, but in fact, NFS allows every machines to be both a client and a server at the same time.

Each NFS server exports one or more of its directories for access by remote clients. When a directory is made available, so are all of its subdirectories, so in fact, entire directory trees are normally exported as a unit. The list of directories a server exports is maintained in the */etc/exports* file, so these directories can be exported automatically whenever the server is booted.

Clients access exported directories by mounting them. When a client mounts a (remote) directory, it becomes part of its directory hierarchy. Many Sun workstations are diskless. If it so desires, a diskless client can mount a remote file system on its root directory, resulting in a file system that is supported entirely on a remote server. Those workstations that do have local disks can mount remote directories anywhere they wish on top of their local directory

hierarchy, resulting in a file system that is partly local and partly remote. To programs running on the client machine, there is (almost) no difference between a file located on a remote file server and a file located on the local disk.

Thus the basic architectural characteristic of NFS is that servers export directories and clients mount them remotely. If two or more clients mount the same directory at the same time, they can communicate by sharing files in their common directories. A program on one client can create a file, and a program on a different one can read the file. Once the mounts have been done, nothing special has to be done to achieve sharing. The shared files are just there in the directory hierarchy of multiple machines and can be read and written the usual way. This simplicity is one of the great attractions of NFS.

6.2 NFS protocols

Since one of the goals of NFS is to support a heterogeneous system, with clients and servers possibly running on dirrerent operating system and on different hardware, it is essential that the interface between the clients and servers be well defined.

The first NFS protocol handles mounting. A client can send a path name to a server and request permission to mount that directory somewhere in its directory hierarchy. The place where it is to be mounted is not contained in the message, as the server does not care where it is to be mounted. If the path name is legal and the directory specified has been exported, the server returns a **file handle**. The file handle contrains fields uniquely identifying the file system type, the disk, the i-node number of the directory, and security information. Subsequent calls to read and write files in the mounted directory use the file handle.

Alternatively, Sun's version of UNIX also supports **automounting**. This feature allows a set of remote directories to be associated with a local directory. None of these remote directories are mounted when the client is booted. Instead, the first time a remote file is opened, the operating system sends a message to each of the servers. The first one to reply wins, and its directory is mounted.

The second NFS protocol is for directory and file access. Clients can send messages to servers to manipulate directories and to read and write files. In addition, they can also access file attributes, such as file mode, size, and time of last modification. Most UNIX system calls are supported by NFS, do not use OPEN and CLOSE. The omission of OPEN and close is not an accident. As it is not necessary to open a file before reading it, nor close it when done. Each message is self-contained, i.e. it contains enough information to operate with a remote file/directory on the server side. The advantage of this scheme is that the server does not have to remember anything about open connections in between calls to it. Thus if server crashes and then recovers, no information about open files is lost. A server like this that does not maintain state information about open files is called **stateless**.

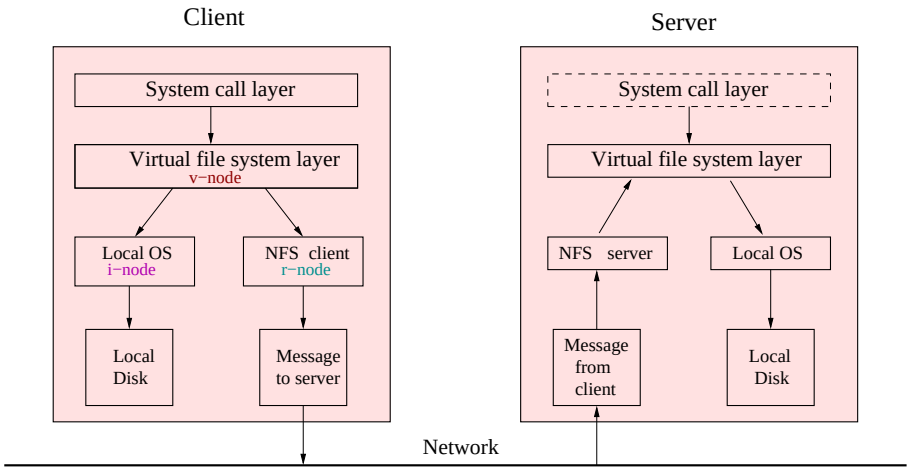

Figure 6.1: NFS layer structure

6.3 NFS Implementation

It consist of three layers, as shown in Fig. 6.1. The top layer is **system call** layer. This level handles calls like OPEN, READ, and CLOSE. After parsing the call and checking the parameters, it invokes the second layer, the **virtual file system** (VFS) layer.

The task of the VFS layer is to maintain a table with one entry for each open file, analogous to the table of i-nodes for open files in UNIX. In ordinary UNIX, an i-node is indicated uniquely by a (device, i-node number) pair. Instead, the VFS layer has an entry, called a **v-node** (**virtual i-node**), for every open file. V-nodes are used to tell whether the file is local or remote. For remote files, enough information is provided to be able to access them.

To see how v-nodes are used, let us trace a sequence of MOUNT, OPEN, and READ system calls. To mount a remote file system, the system administrator calls the *mount* program specifying the remote directory, the local directory on which it is to be mounted, and other information. The *mount* program parses the name of the remote directory to be mounted and discovers the name of the machine on which the remote directory is located. It then contacts that machine asking for a file handle for the remote directory. If the directory exists and is available for remote mounting, the server returns a file handle for the director. Finally, it makes a MOUNT system call, passing the handle to the kernel.

The kernel then constructs a v-node for the remote directory and asks the NFS client code in Fig. 6.1 to create an **r-node** (**remote i-node**) in its internal tables to hold the file handle. The v-nodes points to the r-node. Each v-node in the VFS layer will ultimately contain either a pointer to an r-node in the NFS client code, or a pointer to an i-node in the local operating system (see Fig. 6.1. Thus from the v-node it is possible to see if a file or directory is local or remote, and if it is remote, to find its file handle.

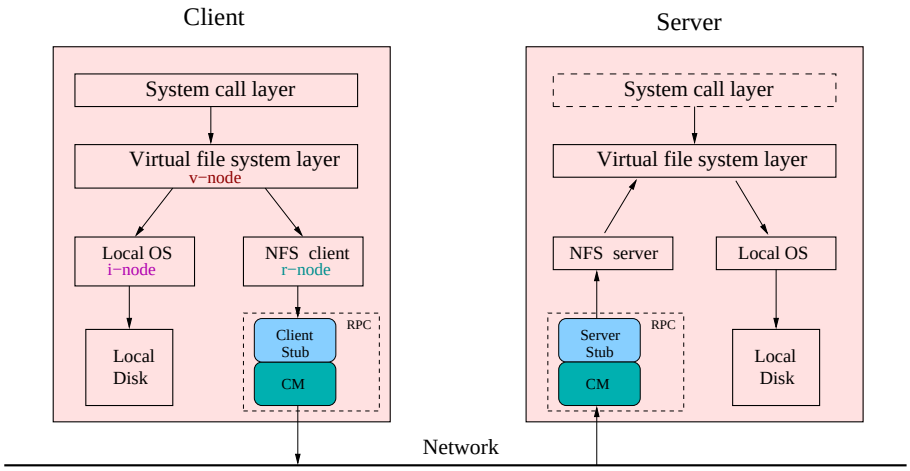

Figure 6.2: NFS layer structure

When a remote file is opened, at some point during the parsing of the path name, the kernel hits the directory on which the remote file system is mounted. It sees that this directory is remote and in the directory's v-node finds the pointer to the r-node. It then asks the NFS client code to open the file. The NFS client code looks up the remaining portion of the path name on the remote server associated with the mounted directory and gets back a file handle for it. It makes an r-node for the remote file in its tables and reports back to the VFS layer, which puts in its tables a v-node for the file that points to the r-node. Again here we see that every open file or directory has a v-node that points to either an r-node or an i-node. Note that no table entries are made on the server side. Although the server is prepared to provide file handles upon request, it does not keep track of which files happen to have file handles outstanding and which do not. When a file handle is sent to it for file access, it checks the handle, and if it is valid, uses it.

Where is a place for an RPC in the NFS structure? See the Fig. 6.2. When a NFS client pass control to the unit which sends a message to a server. And also it provides the recieving of messages. This unit is an RPC. Furthermore the NFS client is a client program, and also the NFS server is a server program.

Bibliography

[1] Andrew D. Birrell and Brace Jay Nelson. Implementing remote procedure calls. *Xerox Palp Alto Research Center*, July 1984.

[2] J.Dollimore G.Coulouris and T.Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, 2nd edition, 1994.

[3] R. Srinivasan. *RFC 1831. RPC: Remote Procedure Call Protocol Specification Version 2.* Sun Microsystems, August 1995.

[4] R. Srinivasan. *RFC 1832. XDR: External Data Representation Standard.* Sun Microsystems, August 1995.

[5] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 2nd edition, 1995.