

A Critique of the Remote Procedure Call Paradigm

Andrew S. Tanenbaum

Robbert van Renesse†

Dept. of Mathematics and Computer Science

Vrije Universiteit

Postbus 7161

1007 MC Amsterdam, The Netherlands

The remote procedure call paradigm is widely used in distributed operating systems. It is conceptually simple to use and straightforward to implement. Nevertheless, experience has shown that it also has some subtle, but less pleasant aspects. In this paper we discuss problems with RPC in the areas of conceptual problems with the model itself, technical problems with implementing it, problems caused by client and server crashes, problems caused by heterogeneous systems, and performance problems. The paper concludes with a discussion and analysis of the problems and proposed solutions.

1. INTRODUCTION

Within the operating system research community, remote procedure call [Birrell and Nelson, 1984; Nelson, 1981] has achieved sacred cow status. It is almost universally assumed to be the appropriate paradigm for building a distributed operating system. Through our use of remote procedure call (RPC) in our own experimental distributed system [references to be provided after blind refereeing is completed], we have discovered that although RPC is an elegant model, it also has a number of unpleasant aspects as well. In this paper we have assembled some of our criticisms, as well as those of other researchers, not because we believe RPC should be abandoned, but as a way to focus attention on the problems and to stimulate others to try and solve them.

Before detailing our criticisms of the RPC model, let us briefly summarize what we mean by RPC. RPC is a communication mechanism between two parties, a *client* and a *server*. For simplicity, let us assume that a computation consist of a main program, running on the client machine, and a procedure to be called, running on the server machine.

When the main program calls the procedure, what actually happens is that a call is made to a special procedure called the *client stub* on the client's machine. The client stub marshalls (collects) the parameters into a message, and then sends the message to the server machine where it is received by the *server stub*.

The server stub unpacks the parameters from the message, and then calls the server procedure using the standard calling sequence. In this way, both the main program and the called procedure see only ordinary, local procedure calls, using the normal calling conventions. Only the stubs, which are typically automatically generated by the compiler, know that the call is remote. In particular, the programmer does not have to be

† This research was supported in part by the Netherlands Foundation for the Advancement of Pure Research (Z.W.O.)

aware of the network at all or the details of how message passing works. The distribution of the program over two machines is said to be *transparent*. Furthermore, between RPCs, there is no connection of any kind established between the client and server.

Our criticism of RPC concerns the advisability of its use as a *general* communication model, for arbitrary applications. In many experimental systems to date, RPC has primarily been used for communication between clients and file servers. In this one restricted application, many of the problems we will point out below do not occur, or can be avoided by careful server design. It is our view that a general paradigm should not require programmers to restrict themselves to a subset of the chosen programming language or force them to adopt a certain programming style (e.g., do not use pointers in their full generality because RPC cannot handle them).

We propose the following test for a general-purpose RPC system. Imagine that two programmers are working on a project. Programmer 1 is writing the main program. Programmer 2 is writing a collection of procedures to be called by the main program. The subject of RPC has never been mentioned and both programmers assume that all their code will be compiled and linked together into a single executable binary program and run on a free-standing computer, not connected to any networks.

At the very last minute, after all the code has been thoroughly tested, debugged, and documented and both programmers have quit their jobs and left the country, the project management is forced by unexpected, external circumstances to run the program on a distributed system. The main program must run on one computer, and each procedure must run on a different computer. We also assume that all the stub procedures are produced mechanically by a stub generating program.

It is our contention that a large number of things may now go wrong due to the fact that RPC tries to make remote procedure calls look exactly like local ones, but is unable to do it perfectly. Many of the problems can be solved by modifying the code in various ways, but then the transparency is lost. Once we admit that true transparency is impossible, and that programmers must know which calls are remote and which ones are local, we are faced with the question of whether a partially transparent mechanism is really better than one that was designed specifically for remote access and makes no attempt to make remote computations look local at all.

An alternative model that does not attempt any transparency in the first place is the virtual circuit model (e.g., the ISO OSI reference model [Zimmermann, 1980]). In this model, a full-duplex virtual circuit using a sliding window protocol is set up between the client and server. If nonblocking SEND and RECEIVE primitives are used, incoming messages can be signalled by interrupts to allow the maximum amount of parallelism between communication and computation. To avoid any misunderstanding, we are certainly NOT advocating that anyone ever use the ISO model for anything. It has more than its share of problems, some of which overlap with RPC's problems and some of which do not. We merely bring this alternative up to provide some perspective and make it clear that some of the problems with RPC are not inherent to distributed computing, but are due to the restrictions imposed specifically by RPC and not present in other paradigms.

In the following sections we will discuss various problems that we and other researchers have encountered with RPC. For convenience we have grouped these into several categories, including conceptual problems with the model, technical problems, problems caused by crashes, problems caused by heterogeneity, and performance problems. These categories frequently overlap, but some structure is better than none.

2. CONCEPTUAL PROBLEMS WITH RPC

In this section we will deal with a variety of problems that are inherent in the RPC model of a client sending a message to a server then blocking until a reply is received.

2.1. Who is the Server and Who is the Client?

RPC is not appropriate to all computations. A simple example of where it is not appropriate, consider a simple UNIX[†] pipeline:

```
sort <infile | uniq | wc -l >outfile
```

that sorts *infile*, an ASCII file with one word per line, strips out the duplicates, and prints the word count on *outfile*.

It is hard to see who is the client and who is the server here. One possible configuration would be to have each of the three programs act as both client and server at times, possibly split up into two processes internally if need be. The client part of *sort* could send read requests to a file server to acquire blocks of the input file. The client part of *uniq* could send requests to the server part of *sort* to provide sorted data as it became available. The client part of *wc* could send requests to the server part of *uniq* to provide duplicateless data as it became available.

So far everything is fine. The problem is what does *wc* do with its output? How does it get it to the file server? If the file server made READ request *wc* could respond with the data, but this would turn the file *server* into a file *client*, certainly an abnormal situation.

This model is read-driven, because the RPC requests are of the form “I want data.” The complementary write-driven model, with *sort* acting as client to *uniq* and saying “Please write this data” solves the problem of producing the output file, since *wc* as client just commands the file server to accept data. Unfortunately it creates a problem for *sort* since the file server refuses to take an active role and pump data at it, as it does to *uniq*.

Having *sort* contain two processes, both clients, one talking to the file server to acquire data and one talking to *uniq* to pump data at it creates an asymmetric situation. The first component of the pipeline then contains two clients and the rest one client and one server. Various ad hoc solutions are possible, such as having the pipes be active processes that pull and push data where needed, but no matter how one looks at it, it is clear that the RPC model just does not fit.

2.2. Unexpected Messages

Various situations exist in which one process has important information for another process, but the intended recipient, typically a client, is not expecting the information. In the RPC model, it is exceedingly difficult for the holder of the information to convey it. In the virtual circuit model with full-duplex connections, either party can send a high-priority (i.e., emergency) message at any moment.

Let us illustrate this problem with two examples. A distributed system has a terminal concentrator to which all the terminals are attached. As characters are typed on the terminals, they are held in the concentrator until some processor in the system, acting as a client, does an RPC to the terminal concentrator, acting as a server, asking for some input. This usually works fine.

[†] UNIX is a Registered Trademark of AT&T Bell Laboratories.

However, a problem arises when a user at a terminal issues a command and then shortly afterwards realizes that the command should be aborted, for example, if the program started up is being debugged and has gotten into an infinite loop. If the user hits the DEL or BREAK key, that character will be held by the terminal server until someone asks for it, which never happens. What is needed is some way for the server, which is normally passive, to initiate action to signal the runaway command or the shell that started it. Here we have an example of a process that is normally a server that in special circumstances has to act like a client and communicate with other processes that are not expecting communication. While various ad hoc solutions can be devised, the RPC model is really at fault here.

A similar problem occurs when workstations sharing a common file server maintain local caches. If one of the workstations modifies part of a file cached by other workstations their caches must be invalidated. In XFDS, modifications to a file are always written through to the file server, which then sends unsolicited messages to the other workstations holding the now invalid file, telling them to purge their caches. As in the terminal server case, we now have a situation in which a server needs to act like a client but the clients are not aware of the fact that they are expected to act like servers.

Of course one can design file systems that do not need unsolicited messages, a dubious concept at best. The point however, is that if the file system designer has good reasons for wanting to do this, it is undesirable that the communication paradigm make such a design impossible. It is as though the ARPANET electronic mail system arbitrarily discarded any message containing the ASCII text "END OF MESSAGE" because such phrases interfered with its internal workings. The communication mechanism should not dictate policy decisions of its users.

2.3. Single Threaded Servers

Another server design decision that RPC virtually forces on the operating system designer is the choice of a multi-threaded over a single threaded file server. Consider a distributed system with a UNIX file server containing a substantial RAM buffer cache, say 64 megabytes, well within the reach of most computer science departments these days.

The file server designer is interested in making the file server as simple as possible to reduce the number of bugs in the code. For this reason, the design chosen is to have a single thread of control within the file server. When a read request arrives at the server stub, it calls the file server as a procedure. The server procedure then carries out the work, usually just fetching a block from the buffer cache, and then returns the requested data to the stub as the value of the procedure.

If the data requested is not in the buffer cache, the file server procedure reads it from the disk, suspending all file server activity while waiting. If the hit rate from the 64M cache is high enough, the designers may consider the occasional disk wait preferable to a complex multi-threaded file server. In any event, for better or for worse, that is their decision to make.

Now consider what happens if a client reads from an empty pipe. In a virtual circuit system, the file server would simply make some table entries noting that the client was trying to read from the pipe, and then go back to the top of its main loop to wait for the next request message. In an RPC system, this is impossible. The server procedure cannot just return control to the stub empty-handed because the stub is programmed to send back the reply to the client immediately. In this case, the reply should not be sent until data arrives in the pipe, possibly hours later.

In a virtual circuit system, the code for the file server looks something like this (in C):

```
do {
    get_message(&mess_buf);
    perform_work(&mess_buf, &reply_buf);
    send_reply(&reply_buf);
}
```

However, if no reply is available (e.g., *reply_buf* contains a special `NO_REPLY_AVAILABLE_YET` code), *send_reply* can just return without sending a reply message. There is no requirement of a strict alternation of getting a message and sending a reply, as there is with RPC.

Because RPC does not allow the same server procedure to be called a second time before it has returned the first time, the server designer is virtually forced to write the server as multithreaded code, with internal multiprocessing. We are not arguing for or against single threaded servers here, but are merely pointing out that using RPC has forced a major design decision on the operating system writers that should be left open to their own judgment.

2.4. The Two Army Problem

Consider what happens if a client requests a server to provide it with some irreplaceable data, for example, by sampling a real-time physics experiment being controlled by the server. After sending its reply, the server cannot just discard the data because the reply may have been lost, in which case the client stub will time out and repeat the request. The question is “How long should the server hold the irreplaceable data?”

One way to handle this problem is to have the client stub send an acknowledgement back to the server stub after receiving the reply. But what happens if the acknowledgement is lost? The server will hold the data forever. To avoid this situation, the server stub should acknowledge the acknowledgement, and the client stub should not terminate the RPC until its acknowledgement has been acknowledged.

However, even this protocol is not adequate. After acknowledging the client stub’s acknowledgement, the server still does not know if the client received the acknowledgement and thus will stop the protocol, or if the acknowledgement got lost, and more messages will be forthcoming from the client side. There is, in fact, no protocol that guarantees that both sides definitely and unambiguously know that the RPC is over in the face of a lossy network.

This problem, known as the two-army problem, also occurs in virtual circuit systems when trying to close a connection gracefully. However there it only occurs once per session, when everything is finished. With RPC it happens on every call. In practice, the problem is not so bad because local networks are highly reliable. Still, one would prefer a mechanism that worked in theory as well as it worked in practice (usually it is the other way around!).

2.5. Multicast

Situations frequently exist in which one process wants to send a message to several other processes. We saw one above—the file server wanting to tell all the processes holding part of a modified file to purge their caches.

Numerous other examples exist. Most local area networks are able to support

broadcast or multicast in hardware. A packet sent in broadcast or multicast mode can be received by multiple machines at once. Thus we have a situation in which processes need to do multicasting and the hardware is able to do it. Only the RPC paradigm is inherently a two-party interaction, so there is no way to utilize the hardware facility.

3. TECHNICAL PROBLEMS

In this section we will look at some problems concerning access to parameters, global variables, and possible timing problems.

3.1. Parameter Marshalling

In order to marshal the parameters, the client stub has to know how many there are and what type they all have. For strongly typed languages, these usually does not cause any trouble, although if union types or variant records are permitted, the stub may not be able to deduce which union member or variant record is being passed.

For languages such as C, which are not type safe, the problems are worse. The procedure *printf*, for example, is called with a variety of different parameters. If *printf* or anything like it is the procedure to be called remotely, the client stub has no easy way of determining how many parameters there are or what there types are.

3.2. Parameter Passing

When the client calls its stub, the call is made using the normal calling sequence. The stub then collects the parameters and puts them into the message to be sent to the server. If all the parameters are value parameters, no problem arises. They are just copied into the message and off they go.

However, if there are reference parameters or pointers, things are more complicated. While it is obviously possible to copy pointers into the message, when the server tries to use them, it will not work correctly because the object pointed to will not be present.

Two possible solutions suggest themselves, each with major drawbacks. The first solution is to have the client stub not only put the pointer itself in the message, but also the thing pointed to. However, if the thing pointed to is the middle of a complex list structure containing pointers in both directions, sublists, etc., copying the entire structure into the message will be expensive. Furthermore, when it arrives, the structure will have to be reassembled at the same memory addresses that it had on the client side, because the server code will just perform indirection operations on the pointers as though it were working on local variables.

Furthermore, if the parameter is a pointer to a union (record variant) of several types, some of which are pointers and some of which are not, it may well be impossible for the client stub to even find the entire data structure because it may not be able to tell which member of the union is the current one.

The other solution is just to pass the pointer itself. Every time the pointer is used, a message is sent back to the client to read or write the relevant word. The problem here is that we violate one of the basic rules: the compiler should not have to know that it is dealing with RPC. Normally the code produced for reading from a pointer is just to indirect from it. If remote pointers work differently from local pointers, the transparency of the RPC is lost.

Forbidden pointers are parameters is equally unattractive since it also violates one of the rules: programmers using RPC systems should not be restricted to only a subset of

the language. If pointers and reference parameters is valid locally, they should be valid remotely as well.

3.3. Global Variables

Most programming languages offer the programmer a way to declare global variables. Procedures may directly access such global variables by just using them. If a procedure that was originally designed to be run locally is suddenly forced to run remote contains references to global variables, these references will fail and the procedure will not work. This problem is similar to that of pointer variables and just as difficult to deal with.

3.4. Timing Problems

For most procedures, the execution speed is not essential for the correct operation of the procedure. However, there is one class of procedure for which the execution speed is critical: I/O device drivers. Some I/O devices have the property that issuing a command to the device requires the driver to write several words into the controller's device registers. Often there are hardware-dependent rules about the allowed interval between the words. For example, it may be required that after the first word has been written, the second word must be written within t microseconds. Failure to observe this limit will cause the controller to time out and the operation to fail.

A problem can occur if the driver calls a small procedure after writing the first word but before writing the second word, for example, to convert the DMA address from virtual to physical. If the small procedure happens to be running remote, the delay introduced may be long enough to cause the controller to time out and the operation to fail.

4. ABNORMAL SITUATIONS

Up until now we have assumed that neither the client nor the server ever crashes. In the real world this assumption is slightly optimistic. Many distributed systems are designed to be fault tolerant, and attempt to reboot crashed processors automatically. For example, if a server fails between RPCs and is brought up again quickly, at first glance it might appear that no one harm is done, but this is not the case. If it crashes during an RPC, even more problems can occur. In this section we will look at some of these problems.

4.1. EXCEPTION HANDLING

The most obvious problem is that normally when the main program calls a procedure, if the code is logically correct the procedure will eventually return to the caller. If the machine crashes, both the main program and the procedure die and the whole program has to be run again. Thus there are basically two modes of operation: the whole program works or the whole program fails.

With RPC another failure mode is introduced: the client works fine but the server crashes. If a main program calls a procedure and there is no response, what should happen? In some systems, the client just hangs forever. While this returns us to the failure modes of the uniprocessor model, one of the goals for making a distributed system is increased fault tolerance, so this approach is not always attractive.

Another possibility is to have the client stub start a timer when it sends the message to the server. If there is no response after a certain interval, it tries again and again. After n retries it concludes that the server is dead and returns an exception or an error code to the calling program.

If the caller expected the exception or error it may be prepared to deal with it. However, in many cases the caller did not expect any exception. For example, a call to the procedure *time* to get the time of day cannot fail when run locally. When run on a remote time server, it can fail if the time server is down. Thus the programmer is forced to check for errors or exceptions where logically none are possible, making the RPC code different from local code and violating transparency.

4.2. Repeated Execution Semantics

A related, but more subtle problem relates to the fact that when a local procedure is called, it is always executed exactly once, no more and no less. With RPC achieving exactly once execution semantics are impossible to achieve in the general case, and complicated and expensive in restricted cases.

For this topic it is important to distinguish two kinds of remote procedures, those that are idempotent and those that are not. Idempotent operations are things like reading block 50 of a file. Performing the operation many times causes no harm. Appending to a file is not idempotent. Neither are most I/O operations. If a remote *write_char* procedure is executed n times, it will write n characters to the output device. Performing idempotent operations remotely cause no problems; it is the nonidempotent ones that cause all the headaches.

Suppose a client does a nonidempotent RPC and the server crashes one machine instruction after finishing the operation, but before the server stub has had a chance to reply. The client stub times out and sends the request again. If the server has rebooted by then, there is a chance that the operation will be performed two or more times and thus fail.

The client stub can prevent the operation from being performed more than once by not repeating the operation upon getting a timeout, but by just reporting failure. The obvious danger here is that the server may have crashed before having performed the operation. For nonidempotent operations there is no way to make the RPC semantics be the same as the local execution semantics. This problem is discussed by Spector [1982].

4.3. Loss of State

Even if a server crashes between RPCs and reboots itself before the next RPC occurs, serious problems can occur. These problems are due to the fact that the server may have long-term state information about the client. A file server, for example, may have information about open files, current file positions, and so on that will be lost when the server reboots. When the client subsequently tries to read file 0, the server will have no idea which file is to be read or how far the program was.

It is obviously possible to try to write servers that have no long term state (e.g., NFS vs. RFS), but in the first place this violates our goal about not restricting programming style on account of RPC, and in the second place it is not always possible. In a data base system, it is frequently necessary for clients to lock files or records before performing operations on them. These locks are long-term state whose loss upon a crash can be disastrous. Trying to devise a data base server that did not use locks might be possible, but it clearly imposes a heavy restriction on the server writer.

4.4. Orphans

So far we have only dealt with server crashes. Client crashes also pose problems. If a client crashes while the server is still busy, the server's computation has become an *orphan*. Obviously having orphans around can be a source of trouble. Virtual circuit systems do not suffer from this problem as much because when a client goes down, its circuits are broken and this can be detected by the server, which then kills off all computations started by the client.

Various methods of orphan elimination have been proposed, but none of them are elegant. Furthermore, most of them have the problem that any method that detects and kills off orphans later runs the risk of killing an orphan in the middle of a critical section, with locks on lots of important resources.

5. HETEROGENEOUS MACHINES

Another class of problems occurs if the client and server run on different kinds of computers (e.g., a VAX and a 68000). The ISO model handles most of these problems with the general mechanism of *option negotiation*. When a virtual circuit is opened, the client can describe the relevant parameters of its machine, and ask the server to describe its parameters. Then the two parties can negotiate and finally choose a set of parameters that both sides understand and can live with. With transparent RPC, the client can hardly be expected to negotiate with its procedures concerning the parameters of the machine they are running on.

5.1. Parameter Representation

One of the problems caused by RPC and which is to some extent soluble by option negotiation is parameter representation. Integers, Booleans, and characters are usually not a problem; the trouble comes with floating point numbers.

5.2. Byte Ordering

Another problem of the same genre is byte ordering. On a VAX, the low-order byte of an integer is the one with the lowest address; on a 68000 it is the one with the highest address. To the extent that the stub routine knows exactly what kind of parameters it has, it can convert them to the format desired by the other party, or to a network standard format.

5.3. Structure Alignment

When a structure is passed as a parameter, the fields may be passed as separate parameters, but when it arrives at the server, the pieces must be reassembled as a structure. Problems can arise if the first field is an 8-bit character and the second field is a 32-bit long. Compilers for 16-bit machines normally skip 1 byte after the character and align the long on a 16-bit boundary, whereas compilers for 32-bit machines would normally skip 3 bytes and align it on a 32-bit boundary.

6. PERFORMANCE PROBLEMS

Our last category of problems has to do with performance rather than correctness. One of the goals of having a distributed system is usually to take advantage of processing power available. In this respect RPC may not be as good as other communication models.

6.1. Lack of Parallelism

With RPC, when the server is active, the client is always idle, waiting for the response. Thus there is never any parallelism possible. The client and the server are effectively coroutines. With other communication models it may be possible to have the client continue computing while the server is working, in order to gain performance.

Furthermore, with a single threaded server and multiple clients, the situation is even worse. While the server is waiting for, say, a disk operation, all the clients have to wait.

6.2. Lack of Streaming

In data base work it is common for a client to request a server to perform an operation to look up tuples in a data base that meet some predicate. With RPC, the server must wait until all the tuples have been found before making the reply. If the operation of finding all the tuples is a time consuming one, the client may be idle for a long time, waiting for the last tuple to be found.

With virtual circuits, the situation is quite different. Here the server can send the first tuple back to the client as soon as it has been located. While the server continues to search for more tuples, the client can be processing the first one. As the server finds more tuples, it just sends them back. There is no need to wait until all have been found.

6.2.1. Bad Assumptions

In many situations, programmers use small procedures instead of inline code because it is more modular and does not affect the performance much. For example, many sort programs have a little routine to exchange element i with element j . If such a procedure ever ran remote, it might slow down the whole computation by the ratio of a remote call to a local call, perhaps a factor of 1000. With nontransparent communication it can never happen that an important little procedure runs remote.