© 2014 IJIRT | Volume 1 Issue 6 | ISSN : 2349-6002

RPC
SECURE MODE OF CALLING

Rakesh Sondal, Sheena Batra
Dronacharya College Of Engineering, Gurgaon, India

Abstract- **Remote procedure call (RPC) is a powerful primitive used for communication and synchronization between distributed processes. RPC mechanism is extended to provide transfer of control and data within a communication network. RPC poses a problem that it reduce the amount of parallelism, because of its synchronous nature. This paper shows how simple process can be used to find a way of avoiding a difficulty in this problem . The primary purpose of our paper is to make distributed computation easier and secure. The combination of blocking RPC calls and light-weight processes provides both simple semantics and efficient exploitation of parallelism.**

I. 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 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 is 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.

II. CONCEPTUAL PROBLEMS WITH RPC

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

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

RPC is not appropriate to all computations. A 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 *serve* into a file *client* , certainly an abnormal. 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. 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 highpriority (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._□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.

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 in 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 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 multiprogramming. 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.

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!).

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.

III. TECHNICAL ISSUES

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

1. Parameter Marshalling

In order to marshall 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.

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 intro introduced may be long enough to cause the controller to time out and the operation to fail.

IV. 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.

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.

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.

3. 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.

CONCLUSION

- Transparency is imperative, and leads to effectiveness
- Maintain local procedure calling semantics
- Binding strategies influences efficiency
- Emulate shared address space
- Timeout implementation

REFRENCES

[1] Birrell, A. D. and Nelson, B. J.; **Implementing remote procedure calls**; ACM Trans. Comput. Syst. 2.1
[2] Garry Nutt; **Operating systems**; 2nd Edition, Addison Wesley
[3] Andrew Tanenbaum and Maarten van Steen; **Distributed Systems: Principles and Paradigms**, Prentice Hall, 2002