ISSN No. 0976-5697

Volume 2, No. 1, Jan-Feb 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

# The Design of Server and the role of Threads in Parallel Processor Applications

Dr. GSVP Raju
Associate Professor
CS&ST Dept
Andhra University, India
ards2003@rediffmail.com

***Abstract:*** The availability of low cost and high performance workstations connected by a high speed network has made distributed computing an attractive mechanism to exploit parallelism at functional level present in user or application programs. A distributed system cab be used efficiently by its end users only if its software presents a single system image to the users. It is observed that in an attempt to exploit any functional level parallelism, a programmer writing user level application programs would be ease while using the threads rather than the processes. Spreading execution of processes or threads over several processors can exploit parallelism and thus achieve improved performance. As compared to a process, a level application programs would be ease while using threads rather than processes. Spreading execution of processes or threads over several processors can exploit parallelism and thus achieve improved performance. As compared to a process, a thread is lighter in terms of overhead associated with creation, context switching, inter process communication and other routing function. This paper describes a prototype for design of server and role of threads in parallel processor applications

***Keywords:*** Serve, thread, parallel processing, RPC

## I. INTRODUCTION

Distributed computing is the method of computer processing in which different parts of a program run simultaneously on two or more computers that are communicating with each other over a network. distributed computing is a type of segmented or parallel computing But the latter term is most commonly used to refer to processing in which different parts of a program run simultaneously on two or more processors that are part of the same computer. While both types of processing require that a program be segmented divided into sections that can run simultaneously, distributed computing also require s that the division of the program take into account the different environments on which the different sections of the program will be running. For example, two computers are likely to have different file systems and different hardware components. An example of distributed component is BOINC, a framework in which large problems can be divided into many small problems which are distributed to many computers. Later the small results are assembled into a larger solution .Distributed computing is a natural result of the use of networks to allow computers to efficiently communicate. But distributed computing is distinct from computer networking or fragmented computing. The latter refers to two or more computers interacting with each other, but not typically sharing the processing of a single program. he Worldwide Web is an example of a network, but not an example of distributed computing. There are numerous technologies and standards used to construct distributed Computations, including some in which are specially designed and optimized for that purpose, such as Remote Procedure Call (RPC). The widest possible range and types of computers, the protocol or communication channel should not contain or use any information that may not be understood by certain organization. Organizing the interaction between each computer is of prime importance. In order to be able to use machines. Special care must also be taken that messages are indeed delivered correctly and that invalid messages are rejected which would otherwise bring down the system and perhaps the rest of the network. Another important factor is the ability to send software to another computer in a portable way so that it may execute and interact with the existing network. This may not always be possible or practical when using differing hardware and resources, in which case other methods must be used such as cross-compiling or manually porting these software.

### A. *Goals and advantages*

The Goals and advantages can be listed as Openness, Monotonicity, Pluralism, Unbounded non determinism and different architectures are

[a] Client-server
[b] 3-tier architecture
[c] N-tier architecture
[d] Tightly coupled (clustered)
[e] Peer-to-peer
[f] Space based

## II. SYSTEM ANALYSIS

Remote procedure call defines a powerful technology for creating distributed client/server program. The RPC run-time stubs and libraries manage of the processes relating to network protocols and communication. It also be used create client and server programs for heterogeneous network environments that include such as systems as Unix and Apple.

### *A. Connecting the Client and the Server*

To communicate client and server programs must establish a communication session across the network or networks that connect them. Once they establish the connection, the client can call remote procedures in the server programs as if they were local to the client program.

#### *[a] Protocol Sequence*:

When network operating systems communicate with each other, they must listen and speak the same language. These languages are called ***protocol sequences.*** Client can server programs must use protocol sequences that the network connecting them.

#### *[i] Server Program:*

The server program runs on the server host computer. However, much literature on client/ server computing refers to both the server program and the server host computer as the "server".

#### *[ii] End Point:*

Server programs listen to a port or a group of ports on the server host computer for client requests. Server host systems maintain a database of these ports, which are called end points in RPC. The database is called the endpoint map.

#### *[iii] Binding:*

Client programs create a to the server to establish a communication session. A binding contains all of the information the client applications needs.

#### *[b] Selecting a Protocol Sequence*

A protocol sequence is the language that a network operating system uses to talk over the network to other computers. In more specific terms, RPC applications must specify a string that represents a combination of an RPC Protocol, a transport protocol and a network protocol. RPC applications can use the rpc protocol to invoke procedures offered by server programs running on the computer that the client program uns on. This is by far, the most efficient method for calling functionality in a different process on the same computer.

End point attribute: The endpoint attribute specifies a well- known port or ports (communication endpoints) on which servers of the interface listen for calls.

Protocol – sequence: It specifies a character string that represents valid combination of an RPC protocol (such as "ip").

End-point: It specifies a string that represents the endpoint designation for the specified protocol family. The syntax of the port string is specific to each protocol sequence.

#### *[c] HOW THE Server Prepares for a Connection*:

When a server program begins execution, it must first register the interface it contains with the RPC run-time library. It then creates the necessary binding information. The server program must also register the end point or end points it listen to. It can then begin listening for client calls.

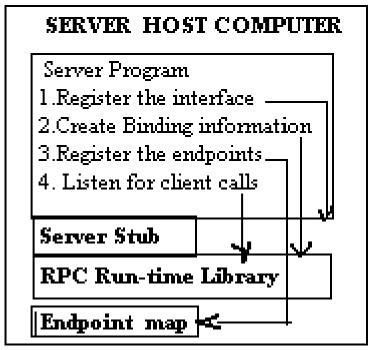

Figure 1: Client Establishes a Connection

#### *[d] How the client establishes a connection:*

To establish a connection client/server communication session with a server program, client applications with explicit handles need to create a binding handle. After they do, the RPC run-time library finds the computer that hosts the server program. It then finds the endpoint that the server program is listening to and directs the call to it. The following diagram illustrates this process.

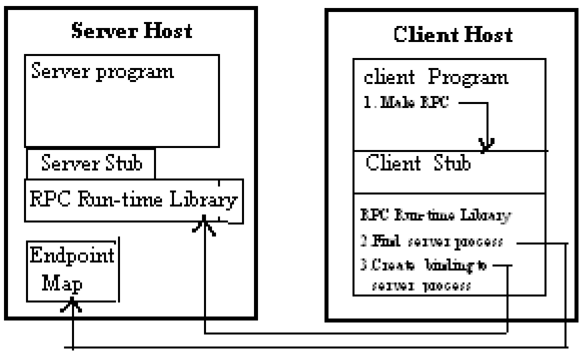

Figure 2: Client Connects to a Server Program

## III. INTERFACE-RPC

A separate, novel RPC system is built on top of the Tube itself. In general, distributed applications built using RPC consist of a number of clients and servers that communicate with each other using predefined and fixed interfaces. Clients are linked in with compiled communication stubs generated from the interface definitions .When clients' use stubs that are compiled once from a definition to communicate directly with a server, some constraints are imposed on the server. If clients are not to fail (or perform marshalling incorrectly), the server must neither change the signatures not remove any of the procedures it offers. Whilst the server can impose versioning and force clients to rebind, neither approach gives the client information about exactly what the server is doing.

Moving (some part of ) the server into the client allows it to give information about any changes made in service provision through up-calls in the client's environment, Facilitating server-driven per-client adaptation that takes into account client state. This also allows for per-client optimization of communication with a server .Supporting plain-text reference by clients of server procedures at run-time allows a server to change its interface without having to worry about the underlying RPC mechanism breaking .When this is combined with a system of negotiation between client and server over the (declarative) naming of services on offer then changes in service provision could provoke a client to re-negotiate its interface with the server.

This re-negotiation introduces a dynamic interface between the parties and represents an advertisement published by a server for clients to take hints from. By relaxing the requirement that a client contracts a service with a fixed interface, more flexibility can be achieved. However, it does break the notion of fixed classes of communication between clients and servers.

Tube RPC supports the mode of operation by allowing an RPC server to send clients a closure which, when run, installs itself there and acts as the server's proxy in the client. The client is freed from having to know how to contact and communicate directly with the server-it conducts all dialogue with the proxy. This allows the server to define at run-time what it offers to clients: it can update the implementation of its proxies at any time, by downloading code to each client, to add or change facilities.

There is work to done into the interface between clients and the server proxies. Clients must be written so that they re able to adapt aw the proxy informs them of new circumstances. The Tube RPC mechanism is now in place, and we will experiment with changing interfaces and shifting the balance between client and server. As well as coping with typing problems, we will ensure that clients which want to use a simple mode of operation that does not require downloading of proxies can do so. We will implement some real applications using this technique and cope with auditing interface change through coins of clients and servers. We are proposing Tube RPC as one possible abstraction over mobile code; our experiments are at a very early stage.

A further extension is to replace the publishing of server addresses in a trader with the publication of small pieces of code; instead of client looking up the address of a server and then explicitly binding to it, it downloads the code from the trader and executes it. The code is responsible for contacting the server and retrieving the proxy that the client can use in further communication. This abstracts particular methods of contact away from clients. Servers publish methods of contact and then give clients tailored proxies for communication.

This allows not easily changing the initial arrangement of clients and servers. At one extreme, the complete server functionality could be published in the trader so that clients would actually download and install servers locally. At the other extreme, the code published in the trader would be a simple network connection to the real server executing somewhere else. Subsequently, by using downloadable proxies, applications can dynamically be rearranged – for instance, a server could download (replicate) itself to all its clients and the terminate.

### *A. Uses and Future Applications*

The Tube was first used for experimenting with REPs that traverse a number of nodes in order to deliver and collect events. The system was at an early stage of development and this first application involved the transfer of drawing events for the update of shared drawing spaces. Whilst no further use of REPs to deliver events is planned (because of the relatively high latency involved in marshalling and interpretation), this approach has evolved into the use of REPs to configure different sites for client-specific event generation and notification. The Tube has extended our existing distributed system with support for a scripting language. These allow us to prototype object implementations, object interfaces and their calling semantics.

In the future, we wish further to integrate the Tube with a traditional RPC system, in particular to allow Tube RPC objects to be given interface definitions so that other, third party components can access their methods. We would then be able to snapshoot at any time the interfaces of Tube RPC objects into a series of IDL files. A system could be prototyped until the balance between client and server elements of each component was correct and the snapshots of their interfaces taken in order to gain a high-level view. We believe it to be a general and powerful mechanism that we intend to exploit in any application involving the Tube in order to evaluate its mode of operation. We have also started to investigate the use of mobile code in Computer Supported Collaborative Working (CSCW) applications. Mobile management entity objects are used to group a user's objects into those participating into particular instances of an application. When interfaced with an Active Badge event system, the objects are able to follow a user from room to room.

## IV. THREADS ROLE IN DESIGN OF SERVER

Knowing how to properly use threads should be part of every computer science and engineering student repertoire. This is an attempt to help you become familiar with multi-threaded programming with the POSIX (Portable OS Interface) threads, or pthreads. This partl explains the different tools defined by the pthread library, shows how to use them, and gives examples of using them to solve real life programming problems.

### *A. Threads*

Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. A thread is a semi-process that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same ser of file descriptors, etc. All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).

### *B. PThreads*

Historically, hardware vendors have implemented their own proprietary versions of threads, These implementations differed substantially from each other, making it difficult for programmers to develop portable threaded applications. In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required, For UNIX systems, this interface have been specified by the IEEE POSIX 1003. 1c standard (1996). Implementations which adhere to this standard are referred to as POSIX threads, or P threads. Most hard vendors now offer P threads in addition to their proprietary threads.

### *C. Efficiency of Threads*

If implemented correctly, threads have some advantages over processes, Compared to the standard fork (), threads carry a lot less overhead. Remember that fork () produces a second copy of the calling process. The parent and the child are completely independent, each with its own address space, with its own copies of its variables, which are completely independent of the same variables in the other process.
Threads share a common address space, thereby avoiding a lot of the inefficiencies of multiple processes.

[a] The kernel does not need to make a new independent copy of the process memory space, file descriptors, etc. This saves a lots of CPU time, making thread creation ten to a hundred times faster than a new process creation. Because of this, you can use a whole bunch of threads and not worry about the CPU and memory overhead incurred. This means you can generally create threads whenever it makes sense in your program.
[b] Less time to terminate a thread than a process.
[c] Context switching between threads is much faster then context switching between processes (context switching means that the system switches from running one thread or process, to running another thread or process)
[d] Less communication overheads – communicating between the threads of one process is simple because the threads share the addresses space the address space. Data produced by one thread is immediately available to all the other threads.

On the other hand, because threads in a group all use the same memory space, if one of them corrupts the contents of its memory, other threads might suffer as well. With processes, the operating system normally protects processes from one another, and thus if one corrupts its own memory space, other processes won't suffer.

### *D. Thread applications:*

The thread applications can include
[a] A responsive user interface
[b] A graphical interface
[c] A server thread

## V. A REMOTE PROCEDURE CALL LIBRARY

A remote procedure call (RPC) library provides client applications with transparent access to a server Services that the server provides to the client could include computational services access to a file system, access to Part of an operating system, or most other functions used in typical client / server framework.

The crucial requirement of an RPC system is that it provides a client with reliable, transparent access to a server. A remote procedure call to a server looks exactly the same to client application as a local procedure call. This is a very power full concept. The application programmer does not need to be aware pf the fact that he is accessing a remote computer when executing an RPC function. The client application makes a procedure call exactly as it would make any local procedure call. The procedure invoked by the client is called a stub. It flattens the arguments passed to it, packs them with additional information needed by the server, and passes this entire packet to the client network interface. It is important to note that the client application need note that the client application need not know that this stub exists. The stub is generated by the RPC library and provides the illusion of a local procedure call. As far as the client is concerned the stub is actually executing the procedure call.

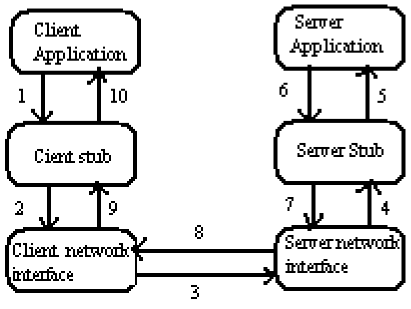

Figure 3: RPC Call

The network interface then executes a protocol to reliably transfer the stub-generated packet to the appropriate server. In step four the server's network interface calls the server stub.

The server stub unpacks the received request and executes the remote procedure on the server. The returned data is then passed back to the server stub, which packages it and has the server network interface transmit it back to the client network interface. The client network interface then passes the packet to the client stub, which is now responsible for unpacking the received packet and returning the desired data to the client application. The design of the network interfaces between the client and the server plays a major role. How an RPC library generates Stubs is a very interesting and complex research area but does not fall within the scope of this project. When generating the stubs one needs to consider such issues as authentication ( this could also occur at the network layer), how to pass parameters (this is especially difficult when pointer are used), which server the RPC call should be made as well as Many more issues.

### *A. RPC network interfaces*

The most important requirement of the RPC network interfaces is that they provide reliable operation. This is especially challenging in this project because UDP is the network protocol that will be used. This is an unreliable protocol that is it can re-order, drop, duplicate and corrupt packets (corrupted packets are dropped). In addition, it, is only able to transmit packets of limited size. Because our RPC library should be capable of handling arbitrarily large packets, the network interfaces will have to be capable of breaking large packets into smaller sub-packets and recombining them at the receiving end.

Once might why one would ever want to use UDP for an implementation since reliable protocols such as TCP are available. The reason for this choice of network Protocol is that a carefully designed implementation on top of UDP can avoid much of the overhead associated with network operations that use TCP. Some people even argue that RPC should not be built on top of UDP or even IP. Instead, an entirely new protocol should be used to avoid the time spent on name-resolution (across network boundaries) which is accrued when IP or UDP are used. This latency can be eliminated since RPC is usually performed within a small subnet, not across boundaries. All in all, UDP Provides convenient and sufficiently efficient protocol for our library and was chosen for that reason. One important aspect of our RPC library is that it provides at most once semantics. This means that any RPC call will be executed no more than once, but possibly not all. It would be desirable to have a RPC protocol that provides exactly –once semantics. However, it is extremely difficult to achieve this because of the need to continuously check-point the work that has been done to ensure that execution can be resumed in case of a server crash.

At-most once semantics will ensure that an RPC call is not executed multiple times(which could easily occur in a trivial RPC implementation because of the message duplication in the network).Only in rate circumstances, such as a server crash will the RPC request not be executed at all. From the client's perspective the most desirable property of the RPC library is that it executes quickly Small RPC requests have to be handled very efficiently to hide as much latency as possible and large RPC calls have to be handled efficiently so that they execute quickly, while not using too many resources. In particular, the RPC library should try to use the network resources efficiently. In many applications the network is a bottleneck and so network bandwidth should not be wasted.

The network interface of the RPC would b used in a typical stub. On the left side of the figure are the procedure calls made by the client and the right are made by the server. The arrows between client and server indicate message being sent over the network.(note: more detailed descriptions of the protocols and data structures used will be given in the design and implementation section)

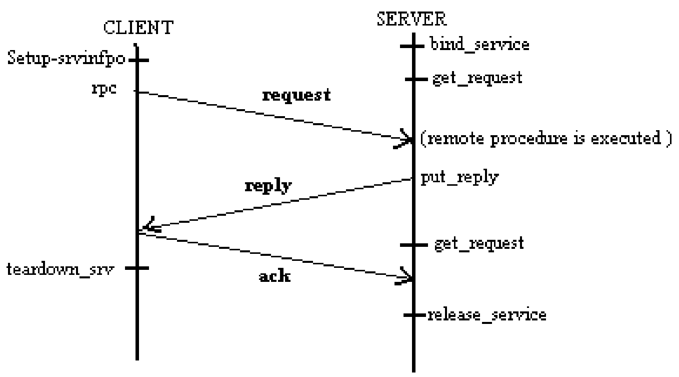

Figure 4: Network interface Usage

The first step the client and server need to perform isinitialization of their sockets and associated parameters. The server calls the RPC library bind-service to its appropriate socket (the socket is identified by one of the parameters). Bind_service is also responsible for initialization of several internal data structures. The client calls the function setup_srvinfo to initialize its socket. At this point, an internal structure to uniquely identify a client RPC request is also initialized. Once initialization of the server has been completed , the server will execute the get_request command. This command will block until a request from a client arrives. The server is now ready to receive RPC requests.

After the client has been initialized , it makes a call to rpc. This sends the RPC request (along with any parameter values) over the network to the server. After get_request has received the client request, it returns the request as well as client identifier( this is used to reply to the client) to the stub. The stub decode the request ,executes the remote procedure and then makes a call to put_ reply. This takes the value returned by the remote procedure call and sends it to the client. The server then runs get_request again to receive the next RPC request. Once the client , that is the call to rpc, receives the server's response, an acknowledgement of receipt is sent back to the server and the received value is passed on to the client stub which in turn decodes it and passes it on to the client application. The client can now make additional calls to rpc or close the rpc service by calling teardown_srvinfo.

### *B. Multi–Threaded(VS)Single0–threaded Implementation:*

The first major decision that had to be made when designing the RPC network interface library was whether it should be multi-threaded or not. On the client side , a single thread clearly sufficient since the client should block until the rpc call returns. Things are not as clear on the server side. Here it would be nice to have one server thread that reads all the data from the network. Another thread could examine the incoming data and spawn a new thread depending on the action that needs to be performed. Either a thread to perform the RPC call on the server should be spawned or a thread to send reply to the client should be spawned. Such a multi-threaded server would clearly by much more efficient than a single-threaded implementation. This is because the server needs to perform are mostly blocking instructions or system calls with much idle time(because of I/O delays).A uni-processor and especially a multi-processor server , would thus benefit from a multi-threaded implementation and most commercial RPC systems are almost certainly multi-threaded. However, we chose to make the server a single threaded process because it makes the design significantly easier. It is not clear to me , if or how , one can share a single socket to communicate with clients among several threads. Many also need to be studied. In order to make the design simpler we choose a single threaded design.

## VI. CONCLUSIONS

This is an efficient and reliable network interface for an RPC library. It provides at-most once semantics, does not waste excessive amounts of network bandwidth , and tries to hide some of the latency at the Server side by multiplexing operations while it is waiting for acknowledgements. Small refinements to the protocol would lead to a more efficient implementation ,but most changes seem to involve considerable design complexity. It is good balance between complexity and performance. This remote procedure mechanism is basis for mark up language like xml soap etc. This is similar to doors concept in UNIX programming so this can be used for Unix systems. The remote procedure call model can be used on same machine to provide security by using large grained protection model provided by them. This protocol is used for designing real time systems.

## VII. REFERENCES

[1] Andrew D.Birrell and Bruce Jay Nelson- Implementing Remote Procedure call.ACM Transaction on computer systems.

[2] Java;the complete reference,7th edition,Herbert schildt,TMH.[3]Understanding OOP with Java,updated edition,T.Budd,pearson education.

[4] An Introduction to OOP,second edition,T.Budd,pearson education.

[5] Introduction to Java programming 6th edition,Y.Daniel Liang,pearson education.

[6] Operating System Concepts-Abraham Silberchatz,Peter B.Galvin,Greg Gagne 7th Edition,John Wiley.

[7] Operating systems-A concept based Approach-D.M.Dhamdhere,2nd Edition,TMH.

[8] Operating Systems-Internal and Design Principles Stallings,Fifth Edition-2005,Pearson education/PHI.

[9] Operating System A Design Approach-Crowley,TMH.