Bapatla Engineering College :: Bapatla

Department of Information Technology

Assignments

Sub: Distributed Systems (IT421) Max.Marks:10M.

Class: 4/4 IT, Section-B Time: 50Min.

**DS Assignment-I**

1 a) Define a distributed system and explain the goals of a distributed systems.

*A distributed system is a collection of independent computers that appears to its users as a single coherent system.*

This definition has several important aspects. The first one is that a distributed system consists of components (i.e., computers) that are autonomous. A second aspect is that users (be they people or programs) think they are dealing with a single system. This means that one way or the other the autonomous components need to collaborate. How to establish this collaboration lies at the heart of developing distributed systems. Note that no assumptions are made concerning the type of computers. In principle, even within a single system, they could range from high-performance mainframe computers to small nodes in sensor networks. Likewise, no assumptions are made on the way that computers are interconnected.

**Goals**

There are four important goals that should be met to make building a distributed system worth the effort. A distributed system should make resources easily accessible; it should reasonably hide the fact that resources are distributed across a network; it should be open; and it should be scalable.

1. ***Making Resources Accessible***

The main goal of a distributed system is to make it easy for the users (and applications) to access remote resources, and to share them in a controlled and efficient way. Resources can be just about anything, but typical examples include things like printers, computers, storage facilities, data, files, Web pages, and networks, to name just a few. There are many reasons for wanting to share resources. One obvious reason is that of economics. For example, it is cheaper to let a printer be shared by several users in a small office than having to buy and maintain a separate printer for each user. Likewise, it makes economic sense to share costly resources such as supercomputers, high-performance storage systems, imagesetters, and other expensive peripherals.

1. ***Distribution Transparency***

An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers. A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent. Let us first take a look at what kinds of transparency exist in distributed systems. After that we will address the more general question whether transparency is always required.

*Types of Transparency*

The concept of transparency can be applied to several aspects of a distributed system, the most important ones shown in figure below.

|  |  |
| --- | --- |
| Transparency | Description |
| Access | Hide differences in data representation and how a resource is accessed |
| Location | Hide where a resource is located |
| Migration | Hide that a resource may move to another location |
| Relocation | Hide that a resource may be moved to another location while in use |
| Replication | Hide that a resource is replicated |
| Concurrency | Hide that a resource may be shared by several competitive users |
| Failure | Hide the failure and recovery of a resource |

***Access transparency*** deals with hiding differences in data representation and the way that resources can be accessed by users. At a basic level, we wish to hide differences in machine architectures, but more important is that we reach agreement on how data is to be represented by different machines and operating systems. For example, a distributed system may have computer systems that run different operating systems, each having their own file-naming conventions. Differences in naming conventions, as well as how files can be manipulated, should all be hidden from users and applications.

***Location transparency*** refers to the fact that users cannot tell where a resource is physically located in the system. Naming plays an important role in achieving location transparency. In particular, location transparency can be achieved by assigning only logical names to resources, that is, names in which the location of a resource is not secretly encoded. An example of a such a name is the URL http://www.prenhall.com/index.html, which gives no clue about the location of Prentice Hall's main Web server. The URL also gives no clue as to whether index.html has always been at its current location or was recently moved there.

***Migration Transparency:*** Distributed systems in which resources can be moved without affecting how those resources can be accessed are said to provide migration transparency.

***Relocation Transparency:***Even stronger is the situation in which resources can be relocated while they are being accessed without the user or application noticing anything. In such cases, the system is said to support relocation transparency. An example of relocation transparency is when mobile users can continue to use their wireless laptops while moving from place to place without ever being (temporarily) disconnected.

***Replication Transparency:*** As we shall see, replication plays a very important role in distributed systems. For example, resources may be replicated to increase availability or to improve performance by placing a copy close to the place where it is accessed. Replication transparency deals with hiding the fact that several copies of a resource exist. To hide replication from users, it is necessary that all replicas have the same name. Consequently, a system that supports replication transparency should generally support location transparency as well, because it would otherwise be impossible to refer to replicas at different locations.

***Concurrency transparency:***

We already mentioned that an important goal of distributed systems is to allow sharing of resources. In many cases, sharing resources is done in a cooperative way, as in the case of communication. However, there are also many examples of competitive sharing of resources. For example, two independent users may each have stored their files on the same file server or may be accessing the same tables in a shared database. In such cases, it is important that each user does not notice that the other is making use of the same resource. This phenomenon is called *concurrency transparency*. An important issue is that concurrent access to a shared resource leaves that resource in a consistent state. Consistency can be achieved through locking mechanisms, by which users are, in turn, given exclusive access to the desired resource.

***Failure Transparency***

A popular alternative definition of a distributed system, due to Leslie Lam-port, is "You know you have one when the crash of a computer you've never heard of stops you from getting any work done." This description puts the finger on another important issue of distributed systems design: dealing with failures. Making a distributed system failure transparent means that a user does not notice that a resource (he has possibly never heard of) fails to work properly, and that the system subsequently recovers from that failure. Masking failures is one of the hardest issues in distributed systems and is even impossible when certain apparently realistic assumptions are made. The main difficulty in masking failures lies in the inability to distinguish between a dead resource and a painfully slow resource. For example, when contacting a busy Web server, a browser will eventually time out and report that the Web page is unavailable. At that point, the user cannot conclude that the server is really down.

1. **Openness**

Another important goal of distributed systems is openness. An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services. For example, in computer networks, standard rules govern the format, contents, and meaning of messages sent and received. Such rules are formalized in protocols. In distributed systems, services are generally specified through interfaces, which are often described in an Interface Definition Language (IDL). Interface definitions written in an IDL nearly always capture only the syntax of services. In other words, they specify precisely the names of the functions that are available together with types of the parameters, return values, possible exceptions that can be raised, and so on. The hard part is specifying precisely what those services do, that is, the semantics of interfaces. In practice, such specifications are always given in an informal way by means of natural language.

If properly specified, an interface definition allows an arbitrary process that needs a certain interface to talk to another process that provides that interface. It also allows two independent parties to build completely different implementations of those interfaces, leading to two separate distributed systems that operate in exactly the same way. Proper specifications are complete and neutral. Complete means that everything that is necessary to make an implementation has indeed been specified. However, many interface definitions are not at all complete, so that it is necessary for a developer to add implementation-specific details. Just as important is the fact that specifications do not prescribe what an implementation should look like; they should be neutral. Completeness and neutrality are important for interoperability and portability (Blair and Stefani, 1998). Interoperability characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard. Portability characterizes to what extent an application developed for a distributed system A can be executed, without modification, on a different distributed system B that implements the same interfaces as A.

Another important goal for an open distributed system is that it should be easy to configure the system out of different components (possibly from different developers). Also, it should be easy to add new components or replace existing ones without affecting those components that stay in place. In other words, an open distributed system should also be extensible. For example, in an extensible system, it should be relatively easy to add parts that run on a different operating system, or even to replace an entire file system. As many of us know from daily practice, attaining such flexibility is easier said than done.

1. ***Scalability***

Worldwide connectivity through the Internet is rapidly becoming as common as being able to send a postcard to anyone anywhere around the world. With this in mind, scalability is one of the most important design goals for developers of distributed systems.

Scalability of a system can be measured along at least three different dimensions (Neuman, 1994). First, a system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system. Second, a geographically scalable system is one in which the users and resources may lie far apart. Third, a system can be administratively scalable, meaning that it can still be easy to manage even if it spans many independent administrative organizations. Unfortunately, a system that is scalable in one or more of these dimensions often exhibits some loss of performance as the system scales up.

2. Explain in detail about a) Hardware Concepts

b) Software Concepts

3)Explain in detail about Remote Procedure Call.

***Remote Procedure Call***

Many distributed systems have been based on explicit message exchange between processes. However, the procedures send and receive do not conceal communication at all, which is important to achieve access transparency in distributed systems. 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 handling communication.

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. No message passing at all is visible to the programmer. This method is known as Remote Procedure Call, or often just RPC.

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 address spaces, which causes complications. Parameters and results also have to be passed, which can be complicated, especially if the machines are not identical. Finally, either or both machines can crash and each of the possible failures causes different problems. Still, most of these can be dealt with, and RPC is a widely-used technique that underlies many distributed systems.

***Basic RPC Operation***

We first start with discussing conventional procedure calls, and then explain how the call itself can be split into a client and server part that are each executed on different machines.

Conventional Procedure Call

To understand how RPC works, it is important first to fully understand how a conventional (i.e., single machine) procedure call works. Consider a call in C like

count = read(fd, buf, nbytes);

where fd is an integer indicating a file, buf is an array of characters into which data are read, and nbytes is another integer telling how many bytes to read. If the call is made from the main program, the stack will be as shown in Fig. 4-5(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. 4-5(b). (The reason that C compilers push the parameters in reverse order has to do with printf—by doing so, printf can always locate its first parameter, the format string.) After the read procedure has finished running, it puts the return value in a register, removes the return address, and transfers control back to the caller. The caller then removes the parameters from the stack, returning the stack to the original state it had before the call.

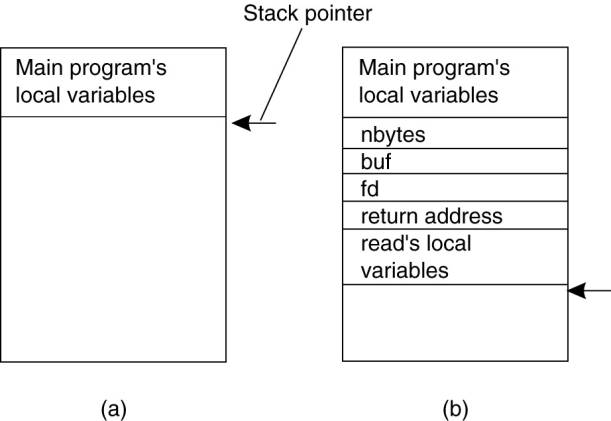


Figure 4-5. (a) Parameter passing in a local procedure call: the stack before the call to read. (b) The stack while the called procedure is active.

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. 4-5(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, overwriting the caller's original value. Under most conditions, this achieves exactly 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 call-by-copy/restore mechanism is not used in many languages.

The decision of which parameter passing mechanism to use is normally made by the language designers and is a fixed property of the language.

***Client and Server Stubs***

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In other words, we want RPC to be transparent—the calling procedure should not be aware that 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, which is generally implemented by calling an equivalent read system call. In other words, the read procedure is a kind of interface between the user code and the local operating system.

Even though read does a system call, it is called in the usual way, by pushing the parameters onto the stack, as shown in Fig. 4-5(b). 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. Like the original one, it, too, is called using the calling sequence of Fig. 4-5(b). Also like the original one, it too, does a call to the local operating system. Only unlike the original one, it does not ask the operating system to give it data. Instead, it packs the parameters into a message and requests that message to be sent to the server as illustrated in Fig. 4-6. Following the call to send, the client stub calls receive, blocking itself until the reply comes back.

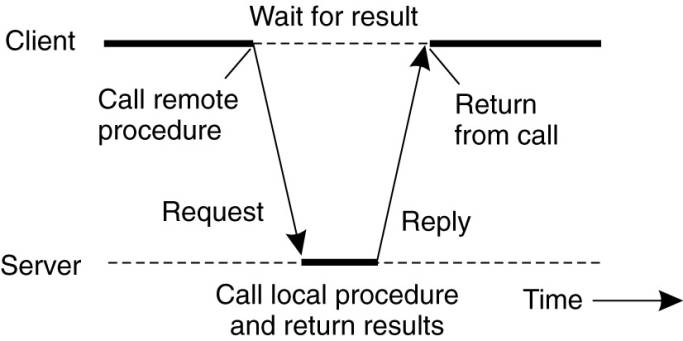


Figure 4-6. Principle of RPC between a client and server program.

When the message arrives at the server, the server's operating system passes it up to a server stub. A server stub is the server-side equivalent of a client stub: it is a piece of code that transforms requests coming in over the network into local procedure calls. 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. 4-5). From the server's point of view, it is as though it is being called directly 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 then returns the result to the caller in the usual way. For example, in the case of read, the server will fill the buffer, pointed to by the second parameter, with the data. This buffer will be internal to the server stub.

When the server stub gets control back after the call has completed, it packs the result (the buffer) in a message and calls send to return it to the client. After that, the server stub usually does a call to receive again, to wait for the next incoming request.

When the message gets back to the client machine, the client's operating system sees that it is addressed to the client process (or actually the client stub, but the operating system cannot see the difference). The message is copied to the waiting buffer and the client process unblocked. The client stub inspects the message, unpacks the result, copies it to its 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 operating system.

This blissful ignorance on the part of the client is the beauty of the whole scheme. As far as it is concerned, remote services are accessed by making ordinary (i.e., local) procedure calls, not by calling send and receive. All the details of the message passing are hidden away in the two library procedures, just as the details of actually making system calls 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 calls the local operating system.

3. The client's OS sends the message to the remote OS.

4. The remote OS 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 calls its local OS.

8. The server's OS sends the message to the client's OS.

9. The client's OS 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 or the existence of the network.

4) Explain in detail about Remote Object Invocation.

5) Explain in detail about Message Oriented Communication.