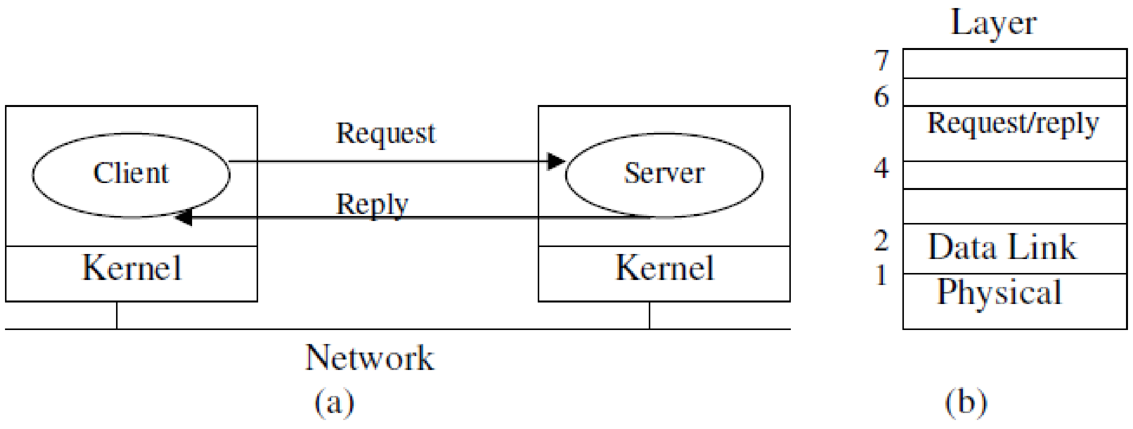
UNIT- 03

REMOTE PROCEDURE CALL

UNIT-03/LECTURE-01

Client Server Model ([RGPV/ June 2011 (7)]

In the basic client-server model, processes in a distributed system are divided into two groups. A Server is a process implementing a specific services. For example, a file system service or a database service. A client is a process that request a service from a server by sending it request and subsequently waiting for the server's reply. The client-server interaction also known as request-reply behaviour. To avoid the considerable overhead of the connection-oriented protocol such as TCP/IP or OSI, the client-server model is usually based on a single connection less request/reply protocol. The client sends a request message to the server asking for same services. The server does the work and returns the data requested or an error code indicating why the work could not be performed.



Client-Server Model

The client sends a request and gets an answer. No connection has to be established before use or turn down afterwards. The reply message serves as the acknowledgement to the request.

The simplicity comes advantage efficiency. The protocol stack is shorter and thus more efficient. Assuming all the machines are identical, only three levels of protocol are needed. The
physical and data link protocol take care of getting the packets from client to server and back. These are always handled by the hardware, for example, an Ethernet or token ring chip. No routing is needed and no connections are established so layer 3 and 4 are not needed. Layer 5 is the request/reply protocol. It defines the set of legal requests and the set of legal replies to these requests. There is no session management because there are no sessions. The upper layers are not needed either.

The communication services provided by the microkernel can, for example, be reduced to two system calls, one for sending message and one for receiving them. These system calls can be invoked through library procedures, say send (dest, &mpti) and receive (addr, &mptr). The former sends the message pointed to by mptr to a process identified by dest and causes the caller to be blocked until the message has been sent. The latter causes the caller to be blocked until a message arrives. When one does, the message is copied to the buffer pointed to by mptr and the caller is unblocked. The addr parameter specifies the address to which the receiver is listening. Many variants of these two procedures and their parameters are possible.

Remote Procedure Call ([RGPV/ June 2011 (7), ([RGPV/ Dec 2013 (7)]

Although the message-passing model provides a convenient way to structure a multicomputer operating system it suffers from one incurable flow- the basic paradigm around which all communication is built is input/output. The procedures send and receive are fundamentally engaged in doing I/O and many people believe that I/O is the wrong program model. This problem has not been solved for a long time unit a paper by Birrell and Nelson introduced a completely different way of attacking the problem. Birrell and Nelson suggested was allowing 2 programs to call procedures located on other CPUs. when a process a machine 1 calls a procedure on machine 2, the calling process on 1 is suspended and execution of the called procedure takes place on 2. information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing or I/O at all is visible to the programmer. This technique is known as **RPC (Remote Procedure Call)** and has become the basis of a large amount of multicomputer software. Traditionally, the calling procedure is known as the client and the called procedure is known as the server.

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In the simplest form, to call a remote procedure, the client program must be bound with a small library procedure called the client stub that represents the server procedure in the client's address space. Similarly, the server is bound with a procedure called the server stub. These procedures hide the fact that the procedure call from client to the server is not local.

The figure shows the steps in making a RPC. These steps are given below –

(i) The client calling the client stub. The call is a local procedure call, with the parameter pushed onto the stack in the normal way.

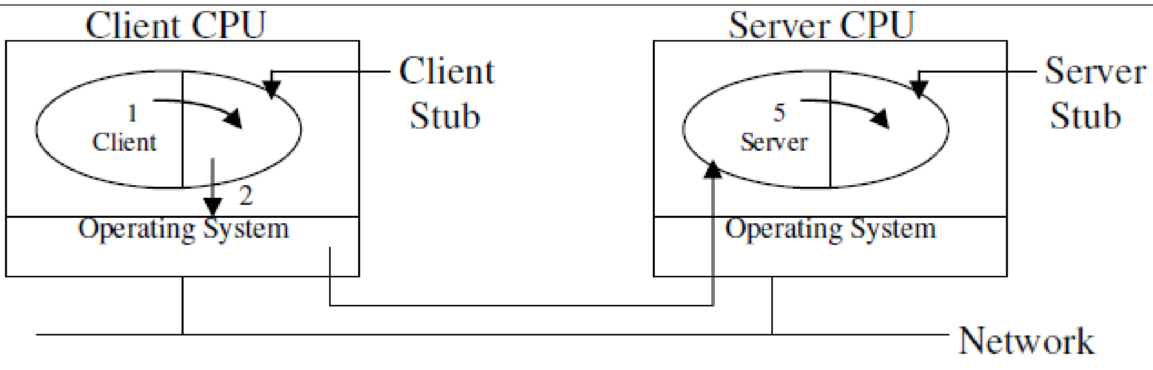
(ii) The client stub packing the parameter into a message and making a system call to send the message.

Packing the parameters is called marshaling.

(iii) The kernel sending the message from the client machine to the server machine.

(iv) The kernel passing the incoming packet to the server stub.

(v) Finally, the server stub calling the server procedure. The reply traces the same in the other direction.



Steps in Making a RPC

The key item to note here is that the client procedure written by the user, must makes a normal procedure call to the client stub, which has the same name the server procedure. Since the client procedure and client stub are in the same address space the parameters are passed in the usual way. Similarly, the server procedure is called by a procedure in its address space with the parameters it expects. To the server procedure, nothing is unusual. In this way, instead of doing I/O using send and receive, remote communication is done by taking a normal procedure call.

The choice of parameter passing semantics is crucial to the design of an RPC mechanism. The two choices are call-by-value and call-by-reference.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	What are the difference between RPC and client server architecture?	June 2011	7
Q.1	Discuss about various remote procedure call semantics.	Dec2013	7

UNIT -03/LECTURE -02

(i) Call-by-value ([RGPV/ June 2011 (7)]

In the call-by-value method, all parameters are copied into a message that is transmitted from the client to the server through the intervening network. This posses no problems for simple compact types such as integers, counters, small arrays and so on. However, passing larger data types, such as multidimensional arrays, trees and so on, can consume much time for transmission of data that may not be used. Therefore this method is not suitable for passing parameters involving voluminous data.

An argument in favour of the high cost incurred in passing large parameters by value is that it forces the users to be aware of the expense of remote procedure calls for large parameters lists. In turn, the users are forced to carefully consider their design of the interface needed client and server to minimize the passing of unnecessary data. Therefore, before choosing RPC parameter passing semantics, it is important to carefully review and properly design the client-server interfaces so that parameters become more specific with minimal data being transmitted.

(ii) Call-by-reference ([RGPV/ June 2011 (7)]

Most RPC mechanisms use the call-by-value semantics for parameter passing because the client and server exist in different address space, possibly even on different types of machines, so that passing pointers or passing parameters by reference is meaningless. However, a few RPC mechanisms do allow passing of parameters by reference in which pointer to the parameters are passed from the client to the server. These are usually closed systems, where a single address space is shared by all processes in the system. For example, distributed system having distributed shared memory mechanisms can allow passing of parameters by reference. In an object-based system that uses the RPC mechanism for object invocation, the call-by-reference semantics is known as call-by-object-reference. This is because in an object based system, the value of a variable is a referenced to an object, so it is this reference that is passed in an invocation. The use of a call-by-object-reference mechanism in distributed systems presents a potentially serious performance problem because on a remote invocation access by the remote operation to an argument is likely to cause an additional remote invocation. Therefore to avoid many remote references, “Emerald” supports a new parameter passing mode that is known as call-by-move. In call-by-move, a parameter is pass by reference, as in the method of call-by-object-reference, but at the time of the call, the parameter object is moved to the destination, but at the time of the cal, the parameter object is moved to the destination node. Following the call, the argument object may either return to the caller’s node or remain at the callee’s node.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Differentiate between (i) Call-by-value (ii) Call-by-reference	June 2011	7

UNIT -03/LECTURE -03

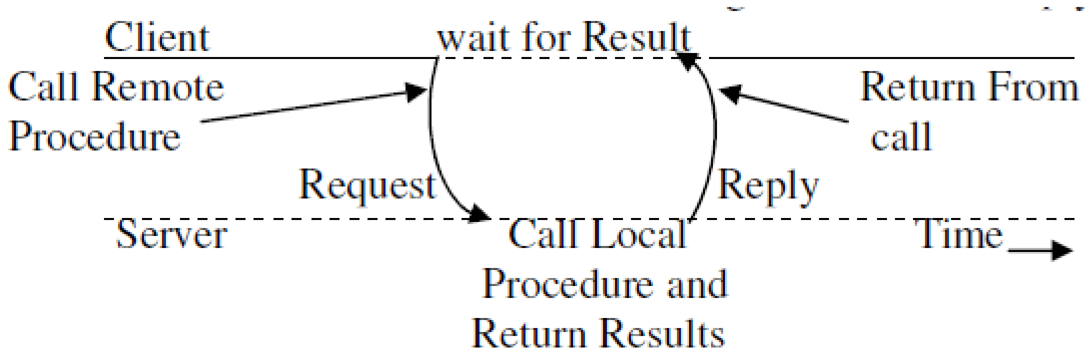
Stub ([RGPV/ June 2014 (7)]

To achieve the goal of semantic transparency, the implementation of RPC mechanism is based on the concept of stubs, which provide a perfectly normal (local) procedure call abstraction by concealing from programs interface to the underlying RPC systems. The RPC involves a client process and a server process, therefore to conceal the interface of the underlying RPC system from both the client and server processes, a separate stub procedure is associated with each of the two processes. Stubs can be generated in one of the following two ways -

(i) Manually :- The RPC implementer provides a set of translation functions from which a user can construct his or her on stubs. This method is simple to implement and can handle very complex parameter types.

(ii) Automatically :- It uses **Interface definition Language** (IDL), that is used to define the interface between names supported by the interface, together with the types of list of procedure names supported by the interface, together with the types of their arguments and results. This is sufficient information for the client and server to independently perform compile-time checking and to generate appropriate calling sequences. Furthermore, an interface definition also contains other information that helps RPC reduce data storage and the amount of data transferred over the network. For example, an interface definition has information to indicate whether each argument is input, or both-only input arguments need by copied from client to server and only output arguments need by copied from server to client. An interface definition also has information about type definitions, enumerated types, and defined constants that each side uses to manipulate data from RPC calls, making it unnecessary for both the client and the server to store this information separately. 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 the traditional 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 and equivalent read system call. Even though read does a system call, it is called in the usual way be pushing the parameters on to the stack. Thus the programmer does not know that read is actually doing something fishy.

When read is implemented in RPC, a different version of read, called a client stub, is put into the library. Like the local call, it to is called using the calling sequence. Also like the local call, it 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 request that message to be sent to the server. Following the call to send, the client stub calls receive, blocking itself until the reply comes back.



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 is the server-side equivalent of a client stubs- it is a piece of code that transforms requests coming in over the network int local procedure calls. Typically the server stub will have called receive and be blockade waiting for incoming messages. The server stub unpacks the parameters from the message and then calls the server procedure in the usual way. From the server's point of view, it is as though it is being called directly by the client-the parameters and return addresses are all on the stack where they belong and nothing seems unusual.

The server performs its working and then returns the result to the caller in the usual way. 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. 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 available. It has no idea that the work done remotely instead of by the local operating system.

Remote services are actually accessed by making ordinary procedure calls, not by calling send and receive. A remote procedure call occurs in the following steps –

(i) The client procedure calls the client stub in the normal way.

(ii) The client stub builds a message and calls the local operating system.

(iii) The client's OS sends the message to the remote OS.

(iv) The remote OS gives the message to the server stub.

(v) The server stub unpacks the parameters and calls the server.

(vi) The server does the work and returns the result to the stub.

(vii) The server stub packs it in a message and calls its local OS.

(viii) The server's OS sends the message to the client's.

(ix) The client's OS gives the message to the client stub.

(x) The stub unpacks the result and return to the client.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	What is the role of stubs and skeleton in distributed object communication? Explain it with example	June 2014	7

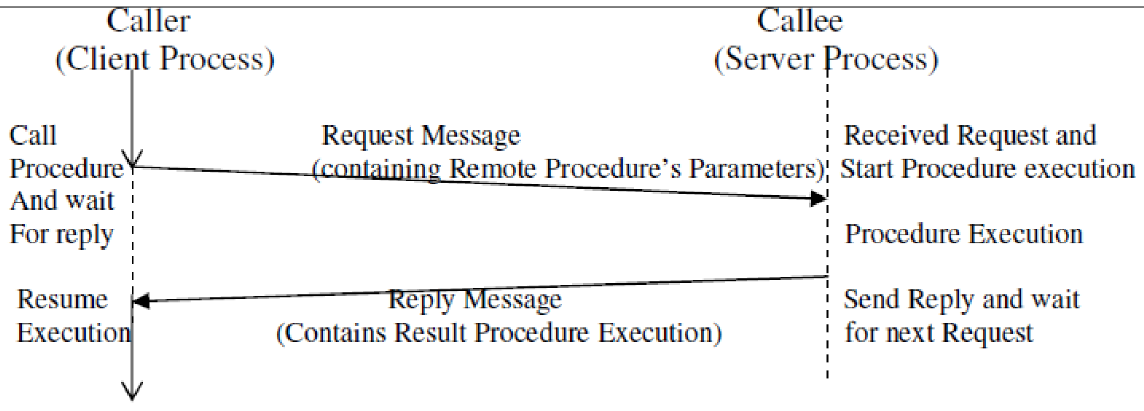
UNIT -03/LECTURE -04

Similarities & differences between RPC model and the ordinary procedure call model ([RGPV/ Dec 2012 (7)]

Similarities

The RPC model is similar to the well known procedure call model used for the transfer of control and data within a program in the following manner –
(i) For making a procedure call the caller places arguments to the procedure in some well specified location.
(ii) Then control is transferred to the sequence of instructions that constitutes the body of the procedure.
(iii) The procedure body is executed in a newly created environment that includes copies of the arguments given in the calling instruction.
(iv) After the procedure's execution is over, control returns to the calling point, possibly returning a result.

In case of RPC, as the caller and the callee processes have disjoint address spaces, the remote procedure has no access to data and variables of the caller's environment. Hence the RPC facility uses a message-passing scheme for information exchange between the caller and the caller processes.


Caller
(Client Process)
Callee
(Server Process)
Call Procedure And wait For reply
Request Message
(containing Remote Procedure's Parameters)
Received Request and Start Procedure execution
Procedure Execution
Resume Execution
Reply Message
(Contains Result Procedure Execution)
Send Reply and wait for next Request

A Typical Model of Remote Procedure Call

(i) The caller (client process) sends a call message to the callee (server process) and waits for a reply message. The request message contains the remote procedure's parameters, among other things.
(ii) The Server process executes the procedure and then returns the result of procedure execution in a reply message to the client process.
(iii) Only the reply message is received, the result of procedure execution is extracted, and the caller's execution is resumed.

Normally, the server process is dormant and awaiting for the arrival of a request message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message and then awaits the next call message. At any given time only one of the two processes is active. In general, the RPC protocol makes no restrictions on the concurrency model implemented, and other models of RPC are possible depending on the details of the parallelism of the caller's and callee's environments and the RPC implementation.

Differences

(i) Unlike local procedure calls, with remote procedure calls, the called procedure is executed in an address space that is disjoint from the calling program's address space. This is the reason, why the called procedure cannot have access to any variables or data values in the calling program's environment. Hence in the absence of shared memory, it is meaningless to pass addresses in arguments. Making call-by-reference pointers highly unattractive. Similarly, it is meaningless to pass argument values containing structures, as pointers are normally represented by memory addresses.

(ii) Remote procedure calls are more vulnerable to failure than local procedure calls, as they involve two different processes and possibly a network and two different computers. Hence programs that make use of RPC must have the capability of handling even those errors that cannot occur in local procedure calls. The need for the ability to take care of the possibility of processor crashes and communication problems of a network makes it even more difficult to obtain the same semantics for remote procedure calls as for local procedure calls.

(iii) Remote procedure calls consume much more time (100-1000 times more) than local procedure calls. This happens due to the involvement of a communication network in RPCs. Therefore application using RPCs must also have the capability to handle the long delays that may possibly occur due to network congestion.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	What are the main similarities and difference between the RPC model and the ordinary procedure call model.	Dec 2012	7

UNIT -03/LECTURE- 05

RPC Protocol Implementation ([RGPV/ June 2014 (7),([RGPV/ Dec 2013 (7)]

There are several major decisions to be made regarding the protocol. The first decision in between a connection-oriented protocol and a connectionless protocol. With a connection-oriented, at the time the client is bound to the server, a connection is established between them. All traffic, in both directions, uses this connection. The advantage of having a connection is that communication becomes much easier. When a kernel sends a message, it does not have to worry about it getting lost, nor does it have to deal with acknowledgements. This is handled at a lower level, by the software that support the connection. This advantage is often too strong to resist over a wide area network.

The disadvantage, especially over a LAN, is the performance loss. All that extra software gets in the way. Moreover, the main advantage (no lost packets) is hardly needed on a LAN, as LANs are so reliable. As a result, most distributed operating systems that are intended for use in a single building or campus use connectionless protocols.

The second design issue is whether to use a standard general-purpose protocol or one specially designed for RPC. As there are no standards in this area, using a custom RPC protocol often means designing your own (or borrowing a friend's). System designers are split about evenly on this one. Some distributed systems use IP as the basic protocol. There are several factors responsible for this choice. They are –

(i) The protocol is already designed, saving considerable work.
(ii) These packets can be sent and received by nearly all UNIX systems.
(iii) Many implementations are available, again saving work.
(iv) IP and UDP packets are supported by many existing networks.

IP and UDP are easy to use and fit in well with existing UNIX systems and networks like internet. This makes it straight forward to write clients and servers that run on UNIX systems, which certainly aids in getting code running quickly and in testing it. However, IP was not designed as an end-user protocol. It was designed as a base upon which reliable TCP connections could be established over recalcitrant internetworks. For example, it can deal with gateways that fragment packets into little pieces so they can through networks with a tiny maximum packet size. However, this feature is not required in a LAN-based distributed system, the IP packet header fields dealing with fragmentation have to be filled in by the sender and verified by the receiver to make them legal IP packets. IP packets have in total 13 fields, of three are useful-the source and destination addresses and the packet length.

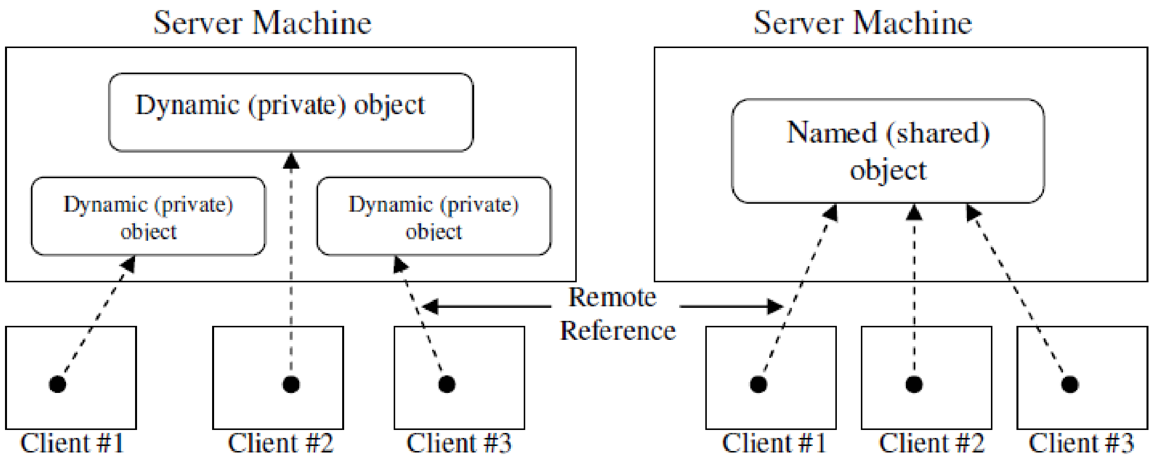
The remaining 10 fields just come along for the ride, and one of them, the header checksum, is time consuming to continue. To make matters worse, UDP has another checksum, covering the data as well. The solution to this problem, is to use a specialized RPC protocol that, unlike IP does not attempt to deal with packets that have been bouncing around the network for a few minutes and then quickly materialize out of him air at an inconvenient moment. However, the protocol has to be invented implemented, tested and embedded in existing systems, so it is considerably more work. Moreover, the rest of the world tends not to jump with joy at the birth of yet another new protocol. In the long run, the development and widespread acceptance of a high performance

RPC protocol is definitely the way to go, but we are not there yet. The last issue regarding the protocol is packet and message length. Doing an RPC has a large, fixed overhead, independent of the amount of data sent. Hence reading a 64 K file in a single 64 K RPC is vastly more efficient than reading it in 64 1K RPCs. Thus it is very important that the protocol and network allow large transmission. Some RPC systems are limited to small size (eg. Sun microsystem’s limit is 8 K). Moreover, many networks cannot handle large packets (Ethernet’s limit is 1536 bytes), So a single RPC will have to be split over multiple packets, causing extra overhead.

Remote Object Invocation ([RGPV/ Dec2012(4)]

Two types of distributed objects are supported. A distributed dynamic object is an object that a server creates locally on behalf of a client, and which in principle, is accessible only to that client. To create an object, a client will have to issue a request at the server. Therefore, each class of dynamic objects has an associated create procedure that can be called using a standard RPC. After creating a dynamic object, the DCE runtime system administrates the new object and associates it with the client on whose behalf it was created.

In contrast to dynamic objects, distributed named objects are not intended to be associated with only a single client but are created by a server to have it shared by several clients. Named objects are registered with a directory service so that a client can look up the object and subsequently bind to it. Registration yields that a unique identifier for that object is stored, along with information on how to contact the object's server. The difference between dynamic and named objects is as follows.



(a) Distributed dynamic objects in DCE **(b) Distributed named**

Each remote object invocation in DCE is done by means of an RPC. When a client invokes a method of an object, it passes the object identifier, the identifier of the interface that contains the method, an identification of the method itself, and parameters to the server. The server maintains an object table from which it can derive which object is to be invoked if given the object identifier and interface identifier. It can then properly dispatch the requested method with its parameters. Because a server may have thousands of objects to serve, DCE offers the possibility to place objects in secondary storage instead of keeping all objects active in main memory. When an invocation request comes in for which no object can be found in the server's object table, the runtime system can alternatively invoke a server-specific lookup function to first retrieve the object from secondary storage and place it into the server's address space. After the object is placed into main memory, the invocation can take place.

Distributed objects in DCE have one problem that is inherent to their strong RPC background : there is no mechanism for transparent object references. At best, a client can use a binding handle associated with a named object. A binding handle contains an identification of an interface of the object the transport protocol used for communicating with the object's server, and the server's host address and endpoint. A binding handle can be turned into a string and as such passed between different processes. Lacking a proper system wide object reference mechanism makes parameter passing in DCE harder than in many other object-based systems. An application developer now has to devise a proprietary solution for passing objects in RPCs. In practice, this means that objects need to be explicitly marshalled to be passed by value, for which object-specific marshalling routines need to be developed.

A developer can use delegation by which a special stub is generated from an object's interface specification. The stub acts as a wrapper for the actual object and contains only those methods that need to be called by a remote process. The stub can then be linked into any other process that wants to use the object. The benefit of this approach becomes clear when realizing that DCE does allow remote reference to stubs to be passed as parameters in RPCs. Consequently, it becomes possible to refer to objects through the entire system by means of stub references.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	What are the different protocols for remote procedure calls? Explain it with diagram.	June 2014,	7
Q.2	Describe the various RPC protocols supporting client server communication.	Dec2013	7
Q.3	Write short note on remote object invocation.	Dec 2012	4

UNIT -03/LECTURE -06

RMI Architecture Layers ([RGPV/ June 2014 (7), ([RGPV/ Dec 2013(7)]

1.Stub and Skeleton layer

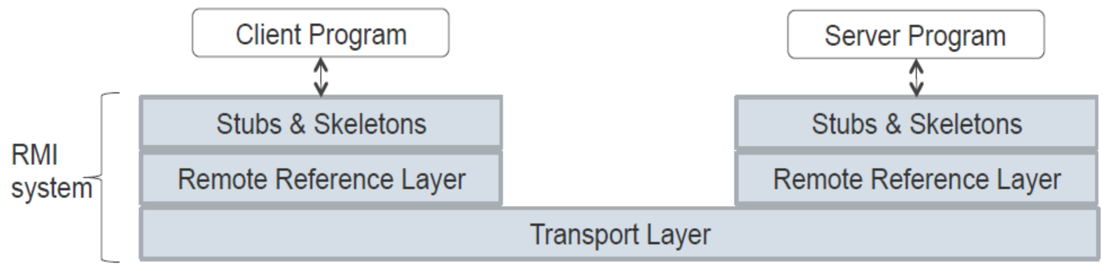
Intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.

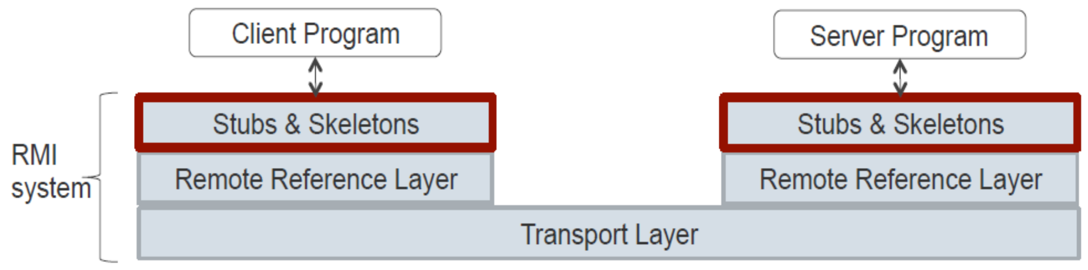
2.Remote Reference Layer

Interpret and manage references made from clients to the remote service objects.

3.Transport layer

Is based on TCP/IP connections between machines in a network .Provides basic connectivity, as well as some firewall penetration strategies





Stub and Skeleton Layer

RMI uses the Proxy design pattern

• Stub class is the proxy

• Remote service implementation class is the RealSubject

Skeleton is a helper class

• Carries on a conversation with the stub

• Reads the parameters for the method call ! makes the call to the remote service implementation object ! accepts the return value ! writes the return value back to the stub.

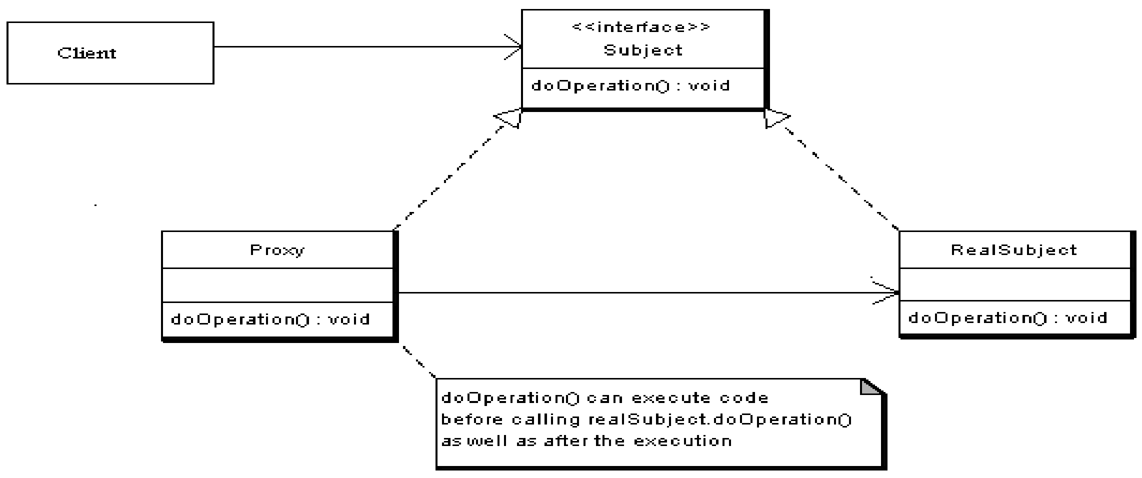
• Please note: In the Java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete. RMI uses reflection to make the connection to the remote service object.

Proxy design pattern

Motivation

Provide a surrogate or placeholder for another object to control access to it.

Implementation


Client
<<interface>>
Subject
doOperation() : void
Proxy
doOperation() : void
RealSubject
doOperation() : void
doOperation() can execute code before calling realSubject.doOperation() as well as after the execution

Proxy design pattern: Applications

Virtual Proxies: delaying the creation and initialization of expensive objects until needed, where the objects are created on demand.

Remote Proxies: providing a local representation for an object that is in a different address space. A common example is Java RMI stub objects. The stub object acts as a proxy where invoking methods on the stub would cause the stub to communicate and invoke methods on a remote object (called skeleton) found on a different machine.

Protection Proxies: where a proxy controls access to RealSubject methods, by giving access to some objects while denying access to others.

Smart References: providing a sophisticated access to certain objects such as tracking the number of references to an object and denying access if a certain number is reached, as well as loading an object from database into memory on demand.

Using Reflection in RMI

• Proxy has to marshal information about a method and its arguments into a request message.

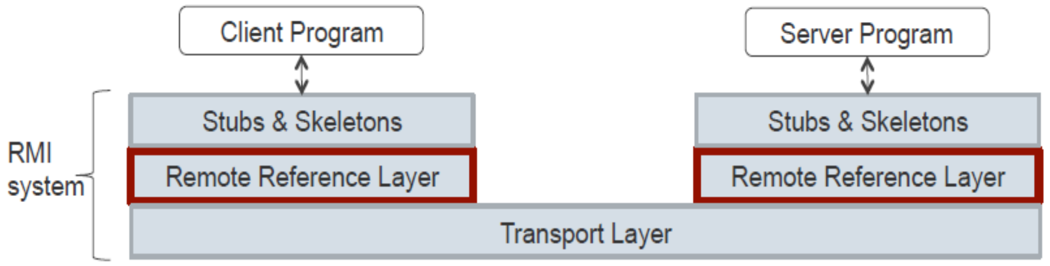
• For a method it marshals an object of class Method into the request. It then adds an array of objects for the methods arguments.

• The dispatcher unmarshals the Method object and its arguments from request message.

• The remote object reference is obtained from remote reference module.

• The dispatcher then calls the Method object's invoke method, supplying the target object reference and the array of argument values.

• After the method execution, the dispatcher marshals the result or any exceptions into the reply message.



Remote Reference Layer

Defines and supports the invocation semantics of the RMI connection. Provides a RemoteRef object that represents the link to the remote service implementation object.

JDK 1.1 implementation of RMI

• Provides a unicast, point-to-point connection

• Before a client can use a remote service, the remote service must be instantiated on the server and exported to the RMI system

Java 2 SDK implementation of RMI

• When a method call is made to the proxy for an activatable object, RMI determines if the remote service implementation object is dormant

• If yes, RMI will instantiate the object and restore its state from a disk file.

Transport Layer

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP.

On top of TCP/IP, RMI uses a wire level protocol called Java Remote Method Protocol (JRMP). JRMP is a proprietary, stream-based protocol that is only partially specified in two versions:

• First version was released with the JDK 1.1 version of RMI and required the use of Skeleton classes on the server.

• Second version was released with the Java 2 SDK. It has been optimized for performance and does not require skeleton classes.

Security

One of the most common problems one encounters with RMI is a failure due to security constraints. For a more complete treatment, one should read the documentation for the Java Security Manager and Policy classes and their related classes.

A Java program may specify a security manager that determines its security policy. A program will not have any security manager unless one is specified. However, many Java installations have instituted security policies that are more restrictive than the default.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain the working of RMI.	June 2014	7
Q.2	Write a short note on RMI.	Dec 2013	7
Q.3	What do you mean by distributed object model.	June 2013	7

UNIT -03/LECTURE -07

Security

Obviously, security plays an important role in any distributed system and object-based ones are no exception. When considering most object-based distributed systems, the fact that distributed objects are remote objects immediately leads to a situation in which security architectures for distributed systems are very similar. In essence, each object is protected through standard authentication and authorization mechanisms.

Security for Remote Objects ([RGPV/Dec 2011(7))]

When using remote objects we often see that the object reference itself is implemented as a complete client-side stub, containing all the information that is needed to access the remote object. In its simplest form, the reference contains the exact contact address for the object and uses a standard marshalling and communication protocol to ship an invocation to the remote object. However, in systems such as Java, the client-side stub (called a proxy) can be virtually anything. The basic idea is that the developer of a remote object also develops the proxy and subsequently registers the proxy with a directory service. When a client is looking for the object, it will eventually contact the directory service, retrieve the proxy, and install it. There are obviously some serious problems with this approach.

First, if the directory service is hijacked, then an attacker may be able to return a bogus proxy to the client. In effect, such a proxy may be able to compromise all communication between the client and the server hosting the remote object, damaging both of them.

Second, the client has no way to authenticate the server: it only has the proxy and all communication with the server necessarily goes through that proxy. This may be an undesirable situation, especially because the client now simply needs to trust the proxy that it will do its work correctly.

Likewise, it may be more difficult for the server to authenticate the client. Authentication may be necessary when sensitive information is sent to the client. Also, because client authentication is now tied to the proxy, we may also have the situation that an attacker is spoofing a client causing damage to the remote object. It describe a general security architecture that can be used to make remote object Invocations safer. In their model, they assume that proxies are indeed provided by the developer of a remote object and registered with a directory service. This approach is followed in Java RMI.

The first problem to solve is to authenticate a remote object. In their solution, Li and Mitchell propose a two-step approach. First, the proxy which is downloaded from a directory service is signed by the remote object allowing the client to verify its origin. The proxy; in tum, will authenticate the object using TLS with server authentication, as we discussed in Chap. 9. Note that it is the object developer's task to make sure that the proxy indeed properly authenticates the object. The client will have to rely on this behaviour, but because it is capable of authenticating the proxy, relying on object authentication is at the same level as trusting the remote object to behave decently.

To authenticate the client, a separate authenticator is used. When a client is looking up the remote object, it will be directed to this authenticator from which it downloads an **authentication** proxy. This is a special proxy that offers an interface by which the client can have itself authenticated by the remote object. If this authentication succeeds. then the remote object (or actually, its object server) will pass on the actual proxy to the client. Note that this approach allows for authentication independent of the protocol used by the actual proxy, which is considered an important advantage.

Another important advantage of separating client authentication is that it is now possible to pass dedicated proxies to clients. For example, certain clients may be allowed to request only execution of read-only methods. In such a case, after authentication has taken place, the client will be handed a proxy that offers only such methods, and no other. More refined access control can easily be envisaged.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Write short note on Security for Remote Objects.	Dec 2011	7

UNIT -03/LECTURE -08

Distributed file system concepts

A file service is a specification of what the file system offers to clients. A file server is the implementation of a file service and runs on one or more machines. A file itself contains a name, data, and attributes (such as owner, size, creation time, access rights). An immutable file is one that, once created, cannot be changed. Immutable files are easy to cache and to replicate across servers since their contents are guaranteed to remain unchanged. Two forms of protection are generally used in distributed file systems, and they are essentially the same techniques that are used in single-processor non-networked systems:

Capabilities

Each user is granted a ticket (capability) from some trusted source for each object to which it has access. The capability specifies what kinds of access are allowed access control lists. Each file has a list of users associated with it and access permissions per user. Multiple users may be organized into an entity known as a group.

File service types

To provide a remote system with file service, we will have to select one of two models of operation. One of these is the upload/download model. In this model, there are two fundamental operations: read file transfers an entire file from the server to the requesting client, and write file copies the file back to the server. It is a simple model and efficient in that it provides local access to the file when it is being used. Three problems are evident. It can be wasteful if the client needs access to only a small amount of the file data. It can be problematic if the client doesn't have enough space to cache the entire file. Finally, what happens if others need to modify the same file?

The second model is a remote access model. The file service provides remote operations such as open, close, read bytes, write bytes, get attributes, etc. The file system itself runs on servers. The drawback in this approach is the servers are accessed for the duration of file access rather than once to download the file and again to upload it. Another important distinction in providing file service is that of understanding the difference between directory service and file service. A directory service, in the context of file systems, maps human-friendly textual names for files to their internal locations, which can be used by the file service. The file service itself provides the file interface (this is mentioned above). Another component of file distributed file systems is the client module. This is the client-side interface for file and directory service. It provides a local file system interface to client software (for example, the vnode file system layer of a UNIX kernel).

Distributed File System (DFS) ([RGPV/Dec2011(7)]

DFS is the file system that is part of the Open Group’s (formerly the Open Software Foundation or OSF) Distributed Computing Environment (DCE) and is a direct descendant of AFS.

Like AFS, it assumes that:

- most file accesses are sequential
- most file lifetimes are short
- the majority of accesses are whole-file transfers
- the majority of accesses are to small files

With these assumptions, the conclusion is that file caching can reduce network traffic and server load. Since the studies on file usage in the early and mid 1980's, it was noticed that file throughput per user has increased dramatically and that typical file sizes became much larger.

DFS implements a strong consistency model (unlike AFS) with Unix semantics supported. This means that a read will return the effects of all writes that precede it. Cache consistency under DFS is maintained by the use of tokens. A token is a guarantee from the server that a client can perform certain operations on the cached file. The server will revoke a token if another client attempts a conflicting operation. A server grants and revokes tokens. It will grant any number of read tokens to clients but as soon as one client requests write access, the server will revoke all outstanding read and write tokens and issue a single write token to the requestor. This token scheme makes long term caching possible (it is not under NFS). Caching is in units of chunk sizes that range from 8K to 256K bytes. Caching is both in client memory and on the disk. DFS also employs read-ahead (similar to NFS) to attempt to bring additional chunks off the file to the client before they are needed.

DFS is integrated with DCE security services. File protection is via access control lists (ACL) and all communication between client and server is via authenticated remote procedure calls.

Sun's Network File System (NFS) ([RGPV/June 2012(7))]

Sun's NFS is one of the most popular and widespread distributed file systems in use today.

The design goals of NFS were:

• Any machine can be a client and/or a server.

• NFS must support diskless workstations (that are booted from the network). Diskless workstations were Sun’s major product line.

• Heterogeneous systems should be supported: clients and servers may have different hardware and/or operating systems. Interfaces for NFS were published to encourage the widespread adoption of NFS.

• high performance: try to make remote access as comparable to local access through caching and read-ahead.

From a transparency point of view NFS offers:

access transparency

Remote (NFS) files are accessed through normal system calls; the protocol is implemented under the VFS (vnode) layer in UNIX.

location transparency

The client adds remote file systems to its local name space via mount. File systems must be exported at the server. The user is unaware of which directories are local and which are remote. The location of the mount point in the local system is up to the client's administrator.

failure transparency

NFS is stateless; UDP is used as a transport. If a server fails, the client retries.

performance transparency

Caching at the client will be used to improve performance

no migration transparency

The client mounts machines from a server. If the resource moves to another server, the client must know about the move.

no support for Unix semantics

NFS is stateless, so stateful operations such as file locking are a problem. All UNIX file system controls may not be available.

NFS protocols

The NFS client and server communicate over remote procedure calls (Sun's RPC) using two protocols: the mounting protocol and the directory and file access protocol. The mounting protocol is used to request a access to an exported directory (and the files and directories within that file system under that directory). The directory and file access protocol is used for accessing the files and directories (e.g. read/write bytes, create files, etc.). The use of RPC's external data representation (XDR) allows NFS to communicate with heterogeneous machines. The initial design of NFS ran only with remote procedure calls over UDP. This was done for two reasons. The first reason is that UDP is somewhat faster than TCP but does not provide error correction (the UDP header provides a checksum of the data and headers). The second reason is that UDP does not require a connection to be present. This means that the server does not need to keep per-client connection state and there is no need to re-establish a connection if a server was rebooted.

The lack of UDP error correction is remedied in the fact that remote procedure calls have built-in retry logic. The client can specify the maximum number of retries (default is 5) and a timeout period. If a valid response is not received within the timeout period the request is re-sent. To avoid server overload, the timeout period is then doubled. The retry continues until the limit has been reached. This same logic keeps NFS clients fault-tolerant in the presence of server failures: a client will keep retrying until the server responds.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	What are the characteristic of distributed file system?	Dec 2011	7
Q.1	Explain the architecture of Network file system.	June 2012	7

UNIT -03/LECTURE -09

Mounting Protocol ([RGPV/June 2012(7))]

The client sends the pathname to the server and requests permission to access the contents of that directory. If the name is valid and exported (listed in /etc/dfs/sharetab on System V release 4 versions of UNIX, and /etc/exports on many other versions) the server returns a file handle to the client. This file handle contains all the information needed to identify the file on the server: {file system type, disk ID, inode number, security info}.

Mounting an NFS file system is accomplished by parsing the path name, contacting the remote machine for a file handle, and creating an in-core vnode at the mount point. A vnode points to an inode for a local UNIX file or, in the case of NFS, an rnode. The rnode contains specific information about the state of the file from the point of view of the client.

Two forms of mounting are supported:

static

In this case, file systems are mounted with the mount command (generally during system boot).

automounting

One problem with static mounting is that if a client has a lot of remote resources mounted, boot-time can be excessive, particularly if any of the remote systems are not responding and the client keeps retrying. Another problem is that each machine has to maintain its own name space. If an administrator wants all machines to have the same name space, this can be an administrative headache. To combat these problems the automounter was introduced.

The automounter allows mounts and unmounts to be performed in response to client requests. A set of remote directories is associated with a local directory. None are mounted initially. the first time any of these is referenced, the operating system sends a message to each of the servers. The first reply wins and that file system gets mounted (it is up to the administrator to ensure that all file systems are the same). To configure this, the automounter relies on mapping files that provide a mapping of client pathname to the server file system. These maps can be shared to facilitate providing a uniform naming space to a number of clients.

Performance

NFS performance was generally found to be slower than accessing local files because of the network overhead. To improve performance, reduce network congestion, and reduce server load, file data is cached at the client. Entire pathnames are also cached at the client to improve performance for directory lookups.

server caching

Server caching is automatic at the server in that the same buffer cache is used as for all other files on the server. The difference for NFS-related writes is that they are all writethrough to avoid unexpected data loss if the server dies.

client caching

The goal of client caching is to reduce the amount of remote operations. Three forms of information are cached at the client: file data, file attribute information, and pathname bindings. We cache the results of read, readlink, getattr, lookup, and readdir operations. The danger with caching is that inconsistencies may arise. NFS tries to avoid inconsistencies (and/or increase performance) with:

- validation - if caching one or more blocks of a file, save a time stamp. When a file is opened or if the server is contacted for a new data block, compare the last modification time. If the remote modification time is more recent, invalidate the cache.

- Validation is performed every three seconds on open files.
- Cached data blocks are assumed to be valid for three seconds.
- Cached directory blocks are assumed to be valid for thirty seconds.
- Whenever a page is modified, it is marked dirty and scheduled to be written(asynchronously). The page is flushed when the file is closed.

Problems

The biggest problem with NFS is file consistency. The caching and validation policies do not guarantee session semantics. NFS assumes that clocks between machines are synchronized and performs no clock synchronization between client and server. One place where this hurts is in distributed software development environments. A program such as make, which compares times of files (such as object and source) to determine whether to regenerate them, can either fail or give confusing results. Because of its stateless design, open with append mode cannot be guaranteed to work. You can open a file, get the attributes (size), and then write at that offset, but you'll have no assurance that somebody else did not write to that location after you received the attributes. In that case your write will overwrite the other once since it will go to the old end-of-file byte offset. Also because of its stateless nature, file locking cannot work. File locking implies that the server keeps track of which processes have locks on the file. Sun's solution to this was to provide a separate process (a lock manager) that does keep state.

One common programming practice under UNIX file systems for manipulating temporary data in files is to open a temporary file and then remove it from the directory. The name is gone, but the data persists because you still have the file open. Under NFS, the server maintains no state about remotely opened files and removing a file will cause the file to disappear. Since legacy applications depended on this, Sun's solution was to create a special hack for UNIX: if the same process that has a file open attempts to delete it, it is instead moved to a temporary name and deleted on close. It's not a perfect solution, but it works well. Permission bits might change on the server and disallow future access to a file. Since

NFS is stateless, it has to check access permissions each time it receives an NFS request. With local file systems, once access is granted initially, a process can continue accessing the file even if permissions change.

By default, no data is encrypted and Unix-style authentication is used (used ID, group ID). NFS supports two additional forms of authentication: Diffie-Hellman and Kerberos. However, data is never encrypted and user-level software should be used to encrypt files if this is necessary.

Since some volume servers may be inaccessible, special treatment is needed to ensure that clients do not read obsolete data. Each file copy has a version stamp. Before fetching a file, a client requests version stamps for that file from all available servers. If some servers are found to have old versions, the client initiates a resolution process which tries to automatically resolve differences (administrative intervention may be required if the process finds problems that it cannot fix). Resolution is only initiated by the client. The process is handled entirely by the servers.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain the architecture of Network file system.	June 2012	7

UNIT -03/LECTURE -10

Andrew File System (AFS) ([RGPV/June 2014(7))]

The goal of the Andrew File System (from Carnegie Mellon University, then a product of Transarc Corp., and now part of the Transarc division of IBM and available via the IBM public license) was to support information sharing on a large scale (thousands to 10000+ users). There were several incarnations of AFS, with the first version being available around 194, AFS-2 in 1986, and AFS-3 in 1989).

The assumptions about file usage were:

• most files are small

• reads are much more common than writes

• most files are read/written by one user

• files are referenced in bursts (locality principle). Once referenced, a file will probably be referenced again.

From these assumptions, the original goal of AFS was to use whole file serving on the server (send an entire file when it is opened) and whole file caching on the client (save the entire file onto a local disk). To enable this mode of operation, the user would have a cache partition on a local disk devoted to AFS. If a file was updated then the file would be written back to the server when the application performs a close. The local copy would remain cached at the client.

Implementation

The client's machine has one disk partition devoted to the AFS cache (for example, 100M bytes, or whatever the client can spare). The client software manages this cache in an LRU (least recently used) manner and the clients communicate with a set of trusted servers.

Each server presents a location-transparent UNIX (hierarchical) file name space to its clients. On the server, each physical disk partition contains files and directories that can be grouped into one or more volumes. A volume is nothing more than an administrative unit of organization (e.g., a user's home directory, a local source tree). Each volume has a directory structure (a rooted hierarchy of files and directories) and is given a name and ID. Servers are grouped into administrative entities called cells. A cell is a collection of servers, administrators, clients, and users. Each cell is autonomous but cells may cooperate and present users with one uniform name space. The goal is that every client will see the same name space (by convention, under a directory /afs). Listing the directory /afs shows the participating cells (e.g., /afs/mit.edu).

Each file and directory is identified by three 32-bit numbers:

volume ID

This identifies the volume to which the object belongs. The client caches the binding between volume ID and server, but the server is responsible for maintaining the bindings.

vnode ID

This is the "handle" (vnode number) that refers to the file on a particular server and disk partition (volume).

Uniquifier

This is a unique number to ensure that the same vnode IDs are not reused. Each server maintains a copy of a database that maps a volume number to its server. If the client request is incorrect (because a volume moved to a different server), the server forwards the request. This provides AFS with migration transparence: volumes may be moved between servers without disrupting access.

Communication in AFS is with RPCs via UDP. Access control lists are used for protection; UNIX file permissions are ignored. The granularity of access control is directory based; the access rights apply to all files in the directory. Users may be members of groups and access rights specified for a group. Kerberos is used for authentication.

Cache coherence

The server copies a file to the client and provides a callback promise: it will notify the client when any other process modifies the file. When a server gets an update from a client, it notifies all the clients by sending a callback (via RPC). Each client that receives the callback then invalidates the cached file. If a client that had a file cached was down, on restart, it contacts the server with the timestamps of each cached file to decide whether to invalidate the file. Note that if an process as a file open, it can continue using it, even if it has been invalidated in the cache. Upon close, the contents will still be propagated to the server. There is no further mechanism for coherency. AFS abides by session semantics.

Under AFS, read-only files may be replicated on multiple servers.

Whole file caching isn't feasible for very large files, so AFS caches files in 64K byte chunks (by default) and directories in their entirety. File modifications are propagated only on close.

Directory modifications are propagated immediately.

AFS does not support byte-range file locking. Advisory file locking (query to see whether a file has a lock on it) is supported.

AFS Summary

AFS demonstrates that whole file (or large chunk) caching offers dramatically reduced loads on servers, creating an environment that scales well. The AFS file system provides a uniform name space from all workstations, unlike NFS, where the client mount each NFS file system at a client specific location (the name space is uniform only under the /afs directory, however). Establishing the same view of the file name space from each client is easier than with NFS. This enables users tomove to different workstations and see the same view of the file system. Access permission is handled through control lists per directory, but there is no per-file access control. Workstation/user authentication is performed via the Kerberos authentication protocol using a trusted third party (more on this in the security section). A limited form of replication is supported. Replicating read-only (and read-mostly at your own risk) files can alleviate some performance bottlenecks for commonly accessed files (e.g. password files, system binaries).

Coda

Coda is a descendent of AFS, also created at CMU. Its goals are:

- Provide better support for replication of file volumes than offered by AFS. AFS' limited form (read-only volumes) of replication will be a limiting factor in scaling the system. We would like to support widely shared read/write files, such as those found in bulletin board systems.
- Provide constant data availability in disconnected environments through hoarding (user-directed caching). This requires logging updates on the client and reintegration when the client is reconnected to the network. Such a scheme will support the mobility of PCs.
- Improve fault tolerance. Failed servers and network problems shouldn't seriously inconvenience users.

To achieve these goals, AFS was modified in two substantial ways:

1. File volumes can be replicated to achieve higher throughput of file access operations and improve fault tolerance.
2. The caching mechanism was extended to enable disconnected clients to operate.

Volumes can be replicated to group of servers. The set of servers that can host a particular volume is the volume storage group (VSG) for that volume. In identifying files and directories, a client no longer uses a volume ID as AFS did, but instead uses a replicated volume ID. The client performs a one-time lookup to map the replicated volume ID to a list of servers and local volume IDs. This list is cached for efficiency. Read operations can take place from any of these servers to distribute the load. A write operation has to be multicast to all available servers. Since some servers may be inaccessible at a particular point in time, a client may be able to access only a subset of the VSG. This subset is known as the Available Volume Storage Group, or AVSG.

Since some volume servers may be inaccessible, special treatment is needed to ensure that clients do not read obsolete data. Each file copy has a version stamp. Before fetching a file, a client requests version stamps for that file from all available servers. If some servers are found to have old versions, the client initiates a resolution process which tries to automatically resolve differences (administrative intervention may be required if the process finds problems that it cannot fix). Resolution is only initiated by the client. The process is handled entirely by the servers.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain the architecture of Andrew file system.	June 2014	7

REFERENCES

BOOK	AUTHOR	PRIORITY
Distributed operating systems; Concepts and design	P K Sinha	1
Distributed systems: Principles and paradigms	Tanenbaum and Steen	2