GOVERNMENT COLLEGE OF ENGINEERING, JALGAON
(An Autonomous Institute of Government of Maharashtra)
Department of Computer Engineering

Name: Vaibhav Bharat Sontakke	PRN No: 1942207
Class: T. Y. B. Tech.	Batch: L4
Date of Performance: ___________	Date of Completion: _____________
Subject: DSL	Sign. of Teacher with Date: __________

Experiment No.: 08

Aim: RPC mechanism for a file transfer across a network.

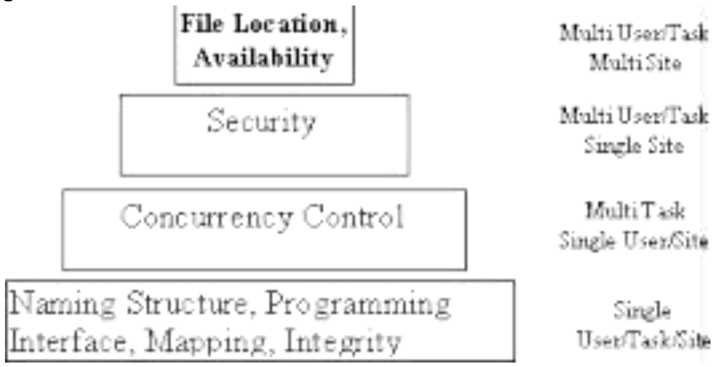
Title: Implement RPC mechanism for a file transfer across a network.

Software Requirements: Ubuntu OS, Eclipse IDE 4.11 **Theory:**

Distributed File Systems

A file system defines the naming structure, characteristics of the files and the set of operations associated with them.

The classification of computer systems and the corresponding file system requirements are given below. Each level subsumes the functionality of the layers below in addition to the new functionality required by that layer, shown in figure 1.


File Location, Availability
Multi User/Task Multi Site
Security
Multi User/Task Single Site
Concurrency Control
Multi Task Single User/Site
Naming Structure, Programming Interface, Mapping, Integrity
Single User/Task/Site

Distributed File System Issues 1. **Naming**
How are files named? Access independent? Is the name location independent?

- FTP. location and access dependent.
- NFS. location dependent through client mount points. Largely transparent for ordinary users, but the same remote file system could be mounted differently on different machines. Access independent. See

Fig 9-3. Has automount feature for file systems to be mounted on demand. All clients could be configured to have same naming structure.

- AFS. location independent. Each client has the same look within a *cell*. Have a cell at each site. See Fig 13-15.

2. Migration

Can files be migrated between file server machines? What must clients be aware of?

- FTP. Sure, but end-user must be aware.
- NFS. Must change mount points on the client machines.
- AFS. On a per-volume (collection of files managed as a single unit) basis.

3. Directories

Are directories and files handled with the same or a different mechanism?

- FTP. Directory listing handled as remote command.
- NFS. Unix-like. 🎬 AFS. Unix-like.

Amoeba has separate mechanism for directories and files.

4. Sharing Semantics

What type of file sharing semantics are supported if two processes accessing the same

file? Possibilities:

- Unix semantics - every operation on a file is instantly visible to all processes.
- session semantics - no changes are visible to other processes until the file is closed.
🎬 immutable files - files cannot be changed (new versions must be created)
- FTP. User-level copies. No support.
- NFS. Mostly Unix semantics. 🎬 AFS. Session semantics.

Immutable files in Amoeba.

5. Caching

What, if any, file caching is supported?

Possibilities:

- write-through - all changes made on client are immediately written through to server
- write-back - changes made on client are cached for some amount of time before being written back to server. 🎬 write-on-close - one type of write-back where changes are written on close (matches session semantics).

- FTP. None. User maintains own copy (whole file)
- NFS. File attributes (inodes) and file data blocks are cached separately. Cached attributes are validated with the server on file open.

Version 3: Uses read-ahead and delayed writes from client cache. Time-based at block level. New/changed files may not visible for 30 seconds. Neither Unix nor session semantics. Non-deterministic semantics as multiple processes can have the same file open for writing.

Version 4: Client must flush modified file contents back to the server on close of file at client. Server can also *delegate* a file to a client so that the client can handle all requests for the file without checking with the

server. However, server must now maintain state about open delegations and recall (with a callback) a delegation if the file is needed on another machine.

- AFS. File-level caching with callbacks (explain). Session semantics. Concurrent sharing is not possible.

6. Locking

Does the system support locking of files?

- FTP. N/A.
- NFS. Has mechanism, but external to NFS in v3. Internal to file system in version 4. 🎬 AFS. Does support.

7. Replication/Reliability

Is file replication/reliability supported and how?

- FTP. No.
- NFS. minimal support in version 4.
- AFS. For read-only volumes within a cell. For example binaries and system libraries.

8. Scalability

Is the system scalable?

- FTP. Yes. Millions of users.
- NFS. Not so much. 10-100s
- AFS. Better than NFS, keep traffic away from file servers. 1000s.

9. Homogeneity

Is hardware/software homogeneity required?

- FTP. No.
- NFS. No. 🎬 AFS. No.

10. File System Interface

Is the application interface compatible to Unix or is another interface used?

- FTP. Separate.
- NFS. The same. 🎬 AFS. The same.

11. Security

What security and protection features are available to control access?

- FTP. Account/password authorization.
- NFS. RPC Unix authentication. Version 4 uses RPCSEC_GSS, a general security framework that can use proven security mechanisms such as Kerberos.
- AFS. Unix permissions for files, access control lists for directories. CODA has secure RPC implementation.

12. **State/Stateless**

Do file system servers maintain state about clients?

- FTP. No.
- NFS. No. In Version 4 servers maintains state about delegations and file locking.
- AFS. Yes.

File Models

1. Unstructured and Structured files

- In the unstructured model, a file is an unstructured sequence of bytes. The interpretation of the meaning and structure of the data stored in the files is up to the application (e.g. UNIX and MS-DOS). Most modern operating systems use the unstructured file model.
- In structured files (rarely used now) a file appears to the file server as an ordered sequence of records. Records of different files of the same file system can be of different sizes.

2. Mutable and immutable files

- Based on the modifiability criteria, files are of two types, mutable and immutable. Most existing operating systems use the mutable file model. An update performed on a file overwrites its old contents to produce the new contents

Mechanisms for Building Distributed File Systems

1. Mounting

Mount mechanisms allow the binding together of different file namespaces to form a single hierarchical namespace. The Unix operating system uses this mechanism. A special entry, known as a mount point, is created at some position in the local file namespace that is bound to the root of another file name space. From the users point of view a mount point is indistinguishable from a local directory entry and may be traversed using standard path names once mounted. File access is therefore location transparent for the user but not for the system administrator. The kernel maintains a structure called a mount table which maps mount points to appropriate file systems. Whenever a file access path crosses a mount point, this is intercepted by the kernel, which then obtains the required service from the remote server.

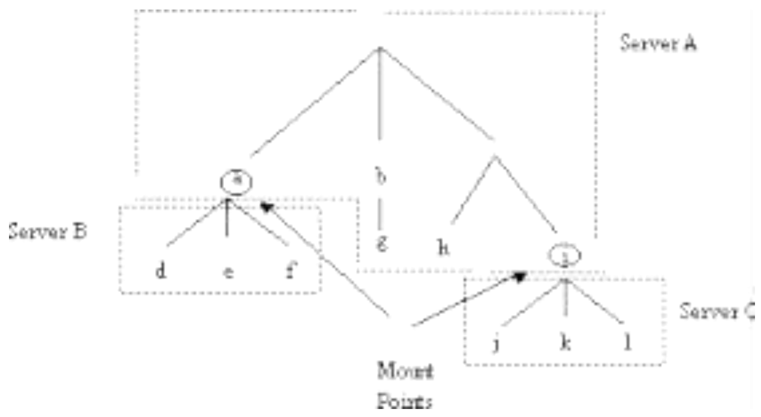


Figure 2- Mounting in file systems

Client machines can maintain mount information individually as is done in Sun's Network File System. Every client individually mounts all the required file systems at specific points in its local file space. Note that each client need not necessarily mount the file systems at the same points. This makes it difficult to write portable code which can locate files in the distributed file system irrespective of where it is executed. Movement of files between servers requires each client to unmount and remount the affected subtree. Note that to simplify the implementation, Unix does not allow hard links to be placed across mount points, i.e. between two separate file systems. Also, servers are unaware of where the subtrees exported by them have been mounted.

Mount information can be maintained at servers in which case it is possible that every client sees an identical namespace. If files are moved to a different server then mount information need only be updated at the servers. Servers each manage domains or volumes which are subtrees of the overall name space. Clients may initially direct file requests to any server which will direct the client to the server which manages the domain containing the required file. This information may be used to guide the client more efficiently for future accesses.

2. Client Caching

Caching is the architectural feature which contributes the most to performance in a distributed file system. Caching exploits temporal locality of reference. A copy of data stored at a remote server is brought to the client. Other metadata such as directories, protection and file status or location information also exhibit locality of reference and are good candidates for caching.

Data can be cached in main memory or on the local disk. A key issue is the size of cached units, whether entire files or individual file blocks are cached. Caching entire files is simpler and most files are in fact read sequentially in their entirety, but files which are larger than the client cache cannot be fetched. Cache validation can be done in two ways. The client can contact the server before accessing the cache or the server can notify clients when data is rendered stale. This can reduce client-server traffic.

3. Bulk Data Transfer

All data transfer in a network requires the execution of various layers of communication protocols. Data is assembled and disassembled into packets, it is copied between the buffers of various layers in the communication protocols and transmitted in individually acknowledged packets over the network. For small amounts of data, the transit time across the network is low, but there are relatively high latency costs involved with the communication protocols establishing peer connections, packaging the small data packets for individual transmission and acknowledging receipt of each packet at each layer.

Transferring data in bulk reduces the relative cost of this overhead at the source and destination. With this scheme, multiple consecutive data packets are transferred from servers to clients (or vice versa) in a burst. At the source, multiple packets are formatted and transmitted with one context switch from client to kernel. At the destination, a single acknowledgement is used for the entire sequence of packets received.

Caching amortizes the cost of accessing remote servers over several local references to the same information, *bulk transfer* amortizes the cost of the fixed communication protocol overheads and possibly disk seek time over many consecutive blocks of a file. Bulk transfer protocols depend on the spatial locality of reference within files for effectiveness. Remember there is substantial empirical evidence that files are read in their entirety. File server performance may be enhanced by transmitting a number of consecutive file blocks in response to a client block request. ***4. Encryption***

Encryption is used for enforcing security in distributed systems. A number of possible threats exist such as unauthorized release of information, unauthorized modification of information, or unauthorized denial of resources. Encryption is primarily of value in preventing unauthorized release and modification of information.

ALGORITHM:
Steps:

1. Defining a remote interface
2. Implementing the remote interface
3. Creating Stub and Skeleton objects from the implementation class using rmic (rmicomplier) 4. Start the rmiregistry
5. Create and execute the server application program
6. Create the client application program.
7. Read over the file content until either EOF is reached or maximum characters is read and stored incharacter array.
8. Execute the client application program.

Conclusion:

In this Practical, we learnt about the implementation of RPC mechanism for a file transfer across

the network. **DOC: 27/04/2021**

…………………………………

Miss. Archana Chitte

Name & Sign of Course Teacher