UNIT 3

3.1. DISTRIBUTED PROCESS SCHEDULING

The primary objective of scheduling is to enhance overall system performance metrics such as process completion time and processor utilization. The existence of multiple processing nodes in distributed systems present a challenging problem for scheduling processes onto processors and vice versa.

3.1.1. A System Performance Model

Partitioning a task into multiple processes for execution can result in a speedup of the total task completion time. The speedup factor S is a function

S = *F (Algorithm, System, Schedule)*

S can be written as:

$$S = \frac{OSPT}{CPT} = \frac{OSPT}{OSPT_{IDEAL}} \times \frac{OCPT_{IDEAL}}{CPT} = S_i \times S_d$$

where

• OSPT = optimal sequential processing time

• CPT = concurrent processing time

• $OSPT_{IDEAL}$ = optimal concurrent processing time

• S_i = the ideal speedup

• S_d = the degradation of the system due to actual implementation compared to an ideal system.

S_i can be rewritten as:

$$S_i = \frac{RC}{RP} \times n$$

Where,

$$RP = \frac{\sum_{i=1}^{m} P_i}{OSPT}$$

And

$$RC = \frac{\sum_{i=1}^{m} P_i}{OCPT_{ideal} \times n}$$

and n is the number of processors. The term $\sum_{i=1}^{m} P_i$ is the total computation of the concurrent algorithm where m is the number of tasks in the algorithm. S_d can be rewritten as:

$$S_d = \frac{1}{1+\rho}$$

Where

$$\rho = \frac{CPT - OCPT_{ideal}}{OCPT_{ideal}}$$

RP is Relative Processing: how much loss of of speedup is due to the substitution of the best sequential algorithm by an algorithm better adapted for concurrent implementation. RC is the Relative Concurrency which measures how far from optimal the usage of the n-processor is. It reflects how well adapted the given problem and its algorithm are to the ideal n-processor system. The final expression for speedup S is

$$S = \frac{RC}{RP} \times \frac{1}{1+\rho} \times n$$

The term ρ is called efficiency loss. It is a function of scheduling and the system architecture. It would be decomposed into two independent terms: $\rho = \rho_{sched} + \rho_{syst}$, but this is not easy to do since scheduling and the architecture are interdependent. The best possible schedule on a given system hides the communication overhead (overlapping with other computations).

The unified speedup model integrates three major components

• algorithm development

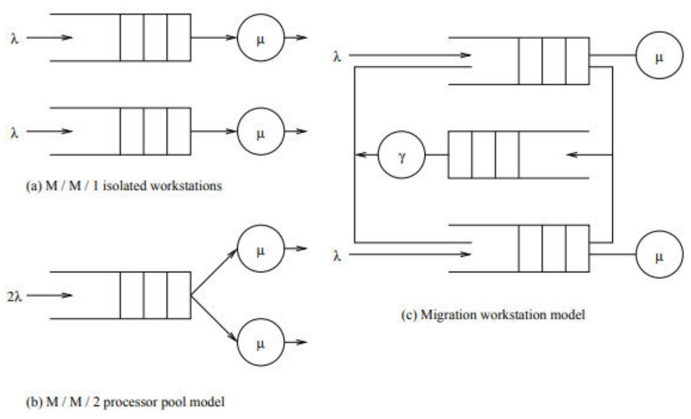
• system architecture

• scheduling policy

with the objective of minimizing the total completion time (makespan) of a set of interacting processes. If processes are not constrained by precedence relations and are free to be redistributed or moved around among processors in the system, performance can be further improved by sharing the workload

• statically - load sharing

• dynamically - load balancing


λ
μ
λ
μ
(a) M / M / 1 isolated workstations
2λ
μ
μ
(b) M / M / 2 processor pool model
λ
μ
γ
λ
μ
(c) Migration workstation model

The standard notation for describing the stochastic properties of a queue is Kendall's notation. An X/Y /c is one with an arrival process X, a service time distribution of Y and c servers. The processor pool can be described as an M/M/2, where M stands for a Markovian distribution.

In the migration workstation model, the migration rate γ is a function of the channel bandwidth, process migration protocol, and context and state information of the process being transferred.

3.1.2. Static Process Scheduling with Communication

- Scheduling a set of partially ordered tasks on a nonpreemtive multiprocessor system of identical processors to minimize the overall finishing time (makespan)
- Except for some very restricted cases scheduling to optimize makespan is NP-complete
- Most research is oriented toward using approximate or heuristic methods to obtain a near optimal solution to the problem
- A good heuristic distributed scheduling algorithm is one that can best balance and overlap computation and communication

In static scheduling, the mapping of processes to processors is determined before the execution of the processes. Once a process is started, it stays at the processor until completion

3.2.2. Dynamic Load Sharing and Balancing

The assumption of prior knowledge of processes is not realistic for most distributed applications. The disjoint process model, which ignores the effect of the interdependency among processes, is used. Objective of scheduling: utilization of the system (has direct bearing on throughput and completion time) and fairness to the user processes (difficult to define).

If we can designate a controller process that maintains the information about the queue size of each processor:

• Fairness in terms of equal workload on each processor (join the shortest queue) - migration workstation model (use of **load sharing** and **load balancing**, perhaps **load redistribution** i.e. **process migration**)

• Fairness in terms of user's share of computation resources (allocate processor to a waiting process at a user site that has the least share of the processor pool) - processor pool model

Solutions without a centralized controller: **sender- and receiver-initiated algorithms.**

Sender-initiated algorithms:

- push model

• includes probing strategy for finding a node with the smallest queue length (perhaps multicast)
• performs well on a lightly loaded system

Receiver-initiated algorithms:

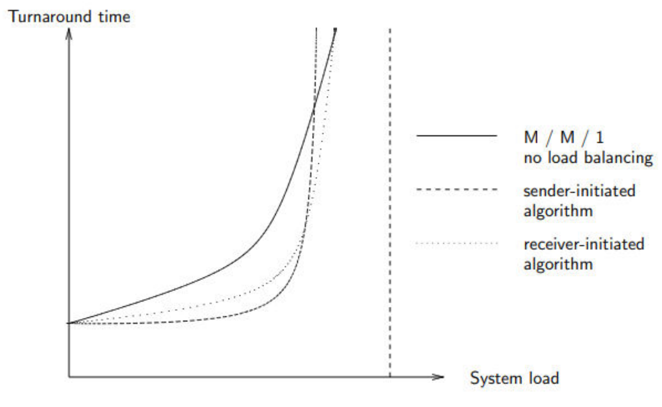
• pull model

• probing strategy can also be used

• more stable

• perform on average better

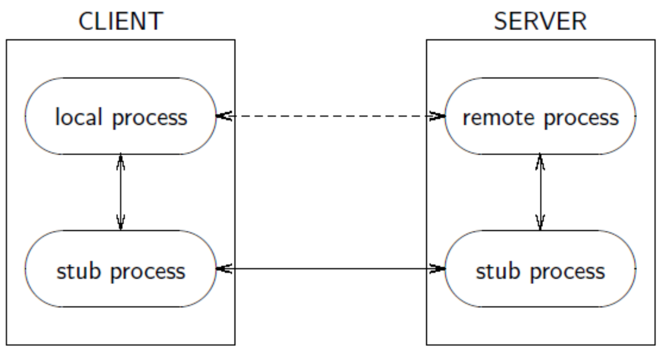
Combinations of both algorithms are possible: choice based on the estimated system load information or reaching threshold values of the processing node's queue.



Performance comparison of dynamic load-sharing algorithms

Performance comparison of dynamic load-sharing algorithms

3.1.4. Distributed Process Implementation



Logical model of local and remote processes

Three significant application scenarios:

Remote service: The message is interpreted as a request for a known service at the remote site (constrained only to services that are supported at the remote host) {

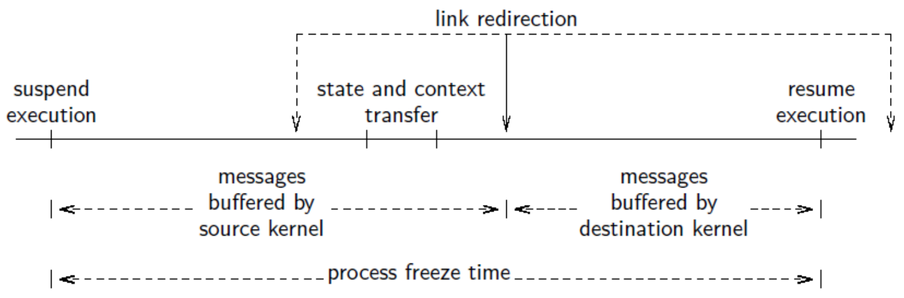
- remote procedure calls at the language level
- remote commands at the operating system level
- interpretive messages at the application level

Remote execution: The messages contain a program to be executed at the remote site; implementation issues:

- load sharing algorithms (sender-initiated, registered hosts, broker...)
- location independence of all IPC mechanisms including signals
- system heterogeneity (object code, data representation)
- protection and security

Process migration: The messages represent a process being migrated to the remote site for continuing execution (extension of load-sharing by allowing a remote execution to be preeemted).

State information of a process in a distributed systems consists of two parts: computation state (similar to conventional context switching) and communication state (status of the process communication links and messages in transit). The transfer of the communication state is performed by *link redirection* and *message forwarding*.


link redirection
suspend execution
state and context transfer
resume execution
messages buffered by source kernel
messages buffered by destination kernel
process freeze time

Link redirection and message forwarding

Reduction of *freeze time* can be achieved with the transfer of minimal state and leaving *residual computation dependency* on the source host: this concept ¯ts well with distributed shared memory.

3.2. DISTRIBUTED FILE SYSTEM

A file system is responsible for the organization, storage, retrieval, naming, sharing, and protection of files. File systems provide directory services, which convert a file name (possibly a hierarchical one) into an internal identifier (e.g. inode, FAT index). They contain a representation of the file data itself and methods for accessing it (read/write). The file system is responsible for controlling access to the data and for performing low-level operations such as buffering frequently used data and issuing disk I/O requests.

3.2.1. Transparencies and Characteristics of DFS

A distributed file system is to present certain degrees of transparency to the user and the system:

Access transparency: Clients are unaware that files are distributed and can access them in the same way as local files are accessed.

Location transparency: A consistent name space exists encompassing local as well as remote files. The name of a file does not give it location.

Concurrency transparency: All clients have the same view of the state of the file system. This means that if one process is modifying a file, any other processes on the same system or remote systems that are accessing the files will see the modifications in a coherent manner.

Failure transparency: The client and client programs should operate correctly after a server failure.

Heterogeneity: File service should be provided across different hardware and operating system platforms.

Scalability: The file system should work well in small environments (1 machine, a dozen machines) and also scale gracefully to huge ones (hundreds through tens of thousands of systems).

Replication transparency: To support scalability, we may wish to replicate files across multiple servers. Clients should be unaware of this.

Migration transparency: Files should be able to move around without the client's knowledge. Support fine-grained distribution of data: To optimize performance, we may wish to locate individual objects near the processes that use them.

Tolerance for network partitioning: The entire network or certain segments of it may be unavailable to a client during certain periods (e.g. disconnected operation of a laptop). The file system should be tolerant of this.

3.2.2. Characteristics of DFS

A good distributed file system should have the following features:

1. **Transparency**
Transparency refers to hiding details from a user. The following types of transparency are desirable.

i. Structure transparency: Multiple file servers are used to provide better performance, scalability, and reliability. The multiplicity of file servers should be transparent to the client of a distributed file system. Clients should not know the number or locations of file servers or the storage devices instead it should look like a conventional file system offered by a centralized, time sharing operating system.

ii. Access transparency: Local and remote files should be accessible in the same way. The file system should automatically locate an accessed file and transport it to the client's site.

iii. Naming transparency: The name of the file should not reveal the location of the file. The name of the file must not be changed while moving from one node to another.

iv. Replication transparency: The existence of multiple copies and their locations should be hidden from the clients where files are replicated on multiple nodes.

2. User mobility
The user should not be forced to work on a specific node but should have the flexibility to work on different nodes at different times. This can be achieved by automatically bringing the users environment to the node where the user logs in.

3. Performance

Performance is measured as the average amount of time needed to satisfy client requests, which includes CPU time plus the time for accessing secondary storage along with network access time. Explicit file placement decisions should not be needed to increase the performance of a distributed file system.

4. Simplicity and ease of use

5. User interface to the file system be simple and number of commands should be as small as possible. A DFS should be able to support the whole range of applications.

6. Scalability
A good DFS should cope with an increase of nodes and not cause any disruption of service. Scalability also includes the system to withstand high service load, accommodate growth of users and integration of resources.

7. High availability
A distributed file system should continue to function even in partial failures such as a link failure, a node failure, or a storage device crash. Replicating files at multiple servers can help achieve availability.

8. High reliability

Probability of loss of stored data should be minimized. System should automatically generate backup copies of critical files in event of loss.

9. Data integrity

Concurrent access requests from multiple users who are competing to access the file must be properly synchronized by the use of some form of concurrency control mechanism. Atomic transactions can also be provided to users by a file system for data integrity.

10. Security

A distributed file system must secure data so that its users are confident of their privacy. File system should implement mechanisms to protect data that is stored within.

11. Heterogeneity

Distributed file system should allow various types of workstations to participate in sharing files via distributed file system. Integration of a new type of workstation or storage media should be designed by a DFS.

3.2.2. DFS Design

The principal motivation for the development of a distributed file system is the need to provide information sharing. This motivates the following system goal which will be defined, along with a brief explanation on how the DFS will realize them.

* Location transparency

* Consistent naming

* Data consistency

* Reliability

Location transparency allows a user or administrator to move file system objects from one network node to another without having to find or alter all name and program references to that object. Therefore the human readable name for an object must not bind the specified object to a particular node. Location transparency results in being able to open a file in precisely the same manner independently of whether the file is local or remote i.e. issue the same system call, with the same parameters in the same order etc. If open 'filename' is used to access local files, it is also used to access remote files. The network thus becomes invisible. Location transparency is achieved in the DFS by having the user view the entire distributed file system as one logical hierarchically structured tree. Users and application programs are thus unaware of the underlying network and that their files are physically distributed over several sites.

Consistent naming means that every program and user in the network uses exactly the same name text for a file system object, regardless of where it exists. The system must thus guarantee that a given name will access a particular file from anywhere in the network. This is achieved by site independent naming of files. Every file has a unique global logical name and the system maintains a mapping between these names and physical file names.

To increase *availability* and *reliability* of files, provision is made in the DFS to replicate files at several sites. In this way, if a site fails it is likely that other sites will continue to operate and provide access to the file. Availability and reliability of a file can be made arbitrarily high by increasing the order of replication. Since data is replicated at one or more sites, the system ensures data consistency by keeping these file images consistent. Consistency is maintained by means of two version numbers, an original version number and a current version number associated with each file. The original version number keeps track of partitioned updates while the current version number is incremented every time the file is updated. The original version number is unused in the current implementation as partition of the network into independently operating subnetworks is assumed not to occur. Thus the problems related to partitioned update are not considered. This version number can be used to accommodate partitioned update if it is to be implemented at a later date and section 5.5 on file consistency describes a possible protocol that may be used.

The DFS protocols assume that the underlying network is fully connected. If site A can communicate with site B and site B with site C, then site A is assumed to be able to communicate with site C. The converse of this assumption of transitivity states that if site B is not accessible to site A then no other site can communicate with it either. This assumption is crucial to recovery protocols wherein if a site is detected as having crashed by any other site, then it is assumed that none of the remaining sites in the network can communicate with the crashed site. Section 5.9 on error recovery discusses this protocol in more detail.

Replication at the file and directory level is provided to increase file availability and reliability in the face of node failures. Reliability is augmented by a simple atomic transaction mechanism which allows all operations bracketed within the 'begin trans' and 'end trans' calls to execute atomically. This transaction mechanism supports failure atomicity, wherein if a transaction completes successfully, the results of its operations will never subsequently be lost and if it is aborted, no action is taken. Thus, either all or none of transactions operations are performed. The transaction mechanism is built on top of a locking mechanism and the system automatically locks data base files in the appropriate mode when they are opened. Deadlock problems are avoided by ensuring that processes never wait for files to be unlocked, and an error message is returned if are not available.

Protection has not been considered an issue in this thesis. All files are created readable and writeable by everyone. Authentication can be implemented at a later date by making a few changes to the name server data base format and associating each file's record in it with a protection code. This code can then be checked on file access to see whether the user has valid permission to read or write the file. In the current implementation as long as the named file and its parent exist in the name server, file access can take place and no other directory searching is done.

3.2.3. DFS Implementation

Major issues common to most distributed file systems are:

* User interface

* Naming

* Inter process communication

* File access mechanism
* File consistency
* Synchronization
* Locking
* Atomic transactions
* Error recovery

This section discusses these issues and describes the DFS implementation with respect to these issues.

1. User interface: The DFS presents users with two command interfaces: the shell interface and the data base interface.

The shell interface: The shell interface is a Unix like command interface that allows manipulation of files (unrestricted streams of bytes) within the hierarchical logical file structure. In general terms a command is expressed as

```
command argl [arg2, arg3 ... argn]
```

Two modes of operations are supported in this interface: default mode and explicit mode. The default mode is the normal mode of operation wherein the system makes use of a version number associated with each file to access the most up to date file image. This mode allows complete location transparency thus relieving the user of having to remember where files are located.

To afford more flexibility over file manipulation, the explicit mode of operation allows users to override this version number mechanism and specify the site at which a file image is located in the command line. Explicit mode is specified by :

```
command arg1@site1 [arg2@site2, arg3@site3 ...]
```

The system can be queried at any time to find out where the files images are located. This mode is useful if the file replication mechanism is to be avoided. Files created in default mode are created at more than one randomly selected site whereas in explicit mode the file is only created at the location specified by the user. The explicit mode can also be used to copy images from one location to another. This is useful in the event that file images become inconsistent due to one or more sites crashing while the file is being updated Under normal circumstances, the system attempts to make the images consistent during the next file access, but this can also be done manually using the explicit mode of operation.

The data base interface: The data base interface is also a command interface much like the shell interface and allows access to special files called data base files. Data base files created in this interface are typed DFS files with all access taking place one record at a time. Commands to create, read, write, remove and rewind a data base file are supported.

This interface has no concept of current working directories or different modes of operation. Default mode is the only mode of operation and all data base files are created in the root directory. In addition an atomic transaction mechanism has been implemented that allows a series of reads and writes to be considered as one atomic action. Transactions are bracketed by 'begin trans' and 'end trans' or 'abort' system calls. All writes to the database are made to a temporary file and written to the data base only in the event of a 'end trans' call and discarded otherwise.

The transaction mechanism is enforced in this interface and it is considered an error if the 'begin trans' system calls ('bt' command) does not precede each database session. The transaction has to be restarted if more operations on the data base are to follow.

2. **File name mapping**: An identifier or a name is a string ofsymbols, usually bits or characters, used to designate or refer to an object. These names are used for a wide variety of purposes like referencing, locating, allocating, error controlling, synchronizing and sharing of objects or resources and exist in different forms at all levels of any system.

If an identified object is to be manipulated or accessed, the identifier must be 'mapped' using an appropriate mapping function and context into another identifier and ultimately into the object. A name service provides the convenience of a runtime mapping from string names to data. An important use of such a service is to determine address information, for example mapping a host name to an IP address. Performing this kind of address look up at run time enables the name server to provide a level of indirection that is crucial to the efficient management of distributed systems. Without such a run time mapping, changes to the system topology would require programs with hardwired addresses to be recompiled, thus severely restricting the scope for systems to evolve.

These name to data mappings may consist of a logically centralized service eg. Clearing House [OPEN 83] or a replicated data base e.g. Yellowpages [WALS 85]. The DFS follows the second approach for file name mapping. To attain location transparency, one must avoid having resource location to be part of the resource name. This is done by dynamically mapping global logical user defined file names to multiple physical file names.

Mapping of logical to physical file names is maintained in a DFS regular replicated file called the name server. Since this file contains system information, it is typed and contains only fixed length records (one record for each file created within the system). Each record in the name server maintains information about a DFS file such as the file type (whether it is a file or a directory) , the number of images, the files logical name, the locations of the file images and the version numbers of the file images. Since all access to any file must go through the CSS, after opening a file, the file descriptors obtained at the US and the CSS uniquely specify the file's global low level name and is used for most of the low level communication about open files.

MAXCOPY images of the name server are created at system boot time at potential CSS locations. Since access to the name server is only through the CSS, each potential CSS will know the locations of the name server images. Reads from the name server may take place from any image with the highest version number. However, since the name server is heavily used, writes are made to the local image when possible thus avoiding network traffic and speeding up access. An error during the write results in failure of the system call and no attempt is made to contact any other image. This enables the CSS to always store the most recently updated image and enables it to keep it consistent. Other operations on the name server include deleting records.

In addition to the name server, the CSS also maintains a cache of the most recently read record in main memory for fast access and in the event the name server becomes inaccessible after a file has been opened.

This cache is in the form of a shared memory segment, since it will be referenced by future calls. On an open, once the name server has been read, and the record cached, no further services of the name server are required until the file is closed. Calls that refer to this cache are the ones associated with locking and transactions. On a close, the cache is deallocated, the version number incremented (if the file was opened for writing) and the record written back to the data base.

Since the name server is a regular replicated file, it may fall out of consistency due to site failure. Consistency of the name server is maintained by a 'watch dog' program running in the background, by checking the version vector associated with the name server at regular intervals. The first record in the name server data base stores version number information about the remaining name server images and this record is read when it is required to be made consistent. If the versions are found to be inconsistent, a copy of the local file image is sent out to the other lower version numbered image sites and consistency is restored.

3. Inter process communication: Machines converse in the DFS by sending messages using an underlying protocol for reliable transmission. The DFS makes use of the Transport Level Interface (TLI) [AT & T 87]mechanism of the Transmission Control Protocol / Internet Protocol (TCP/IP) suite of protocols [POST 81a] and [POST 81b] implemented on the AT&T 3B1 machines, to allow processes on different machines to communicate with each other. The TLI layer provides the basic service of reliable end to end data transfer needed by applications. The kernel structure consists of three parts: the transport interface layer, the protocol layer and the device layer as shown in Figure

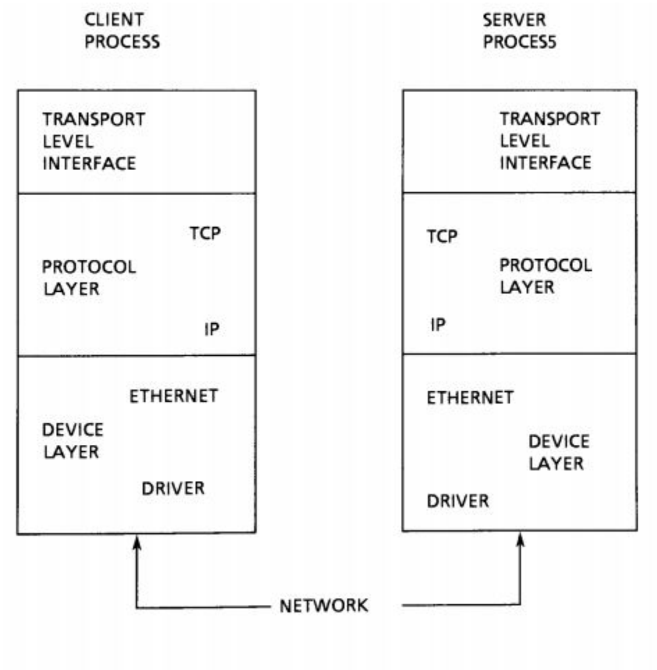


Fig. 5.3 Transport interface model

The transport level interface provides the system call interface between application programs and the lower layers. The protocol layer contains the protocol modules used for communications and the device layer contains the device drivers that control the network devices.

All network messages in the system require an acknowledgement response message from the serving site. The response message, in addition to telling the requesting site that the call was successful, also returns additional information that is produced as a result of the call. Moreover extensive file transfer is carried out by the system in keeping the name server and user files consistent. The connection mode of communication, being circuit oriented, seemed particularly attractive for this kind of data stream interaction and would enable data to be transmitted over the connection in a reliable and sequenced manner. Hence the connection mode service of the transport level interface was chosen for use in the DFS so that higher level system routines are assured of reliable communication.

Every host that wishes to offer service to users (remote or local) has a process server (PS) through which all services must be requested [TANE 81]. Whenever the process server is idle, it listens on a well known address. Potential users of any service must begin by establishing a connection with the process server. Once the connection has been established, the user sends the PS a message telling it which program it wishes to run (either the CSS or the SS). The process server then chooses an idle address and spawns a new process, passing it the parameters of the call, terminates the connection and goes back to listening on its well known address. The new process then executes either the CSS or SS programs, executes the appropriate system call and sends its reply back to the calling process.

Addresses in the Internet domain are composed of two parts, the host network address (identifying the machine) and a port number which is simply a sixteen bit integer that allows multiple simultaneous conversations to occur on the same machine to machine link. These two parts are commonly referred to as a socket and uniquely identifies an end point for communication.

In the current prototype implementation, all process servers are assigned one unique port number which is hardcoded in the program. In an operational implementation, the requesting site or US would look up a (Unix) system data base usually found in /etc/services for the port number of the process server. All requesting sites can also read a (DFS) system file to find out the current location of the CSS. This string name is used to query the Unix system and the appropriate network address of the process server is thus obtained. The two parts of the Internet address are now known and this enables the requesting site to construct the complete address of the process server and thus establish a connection.

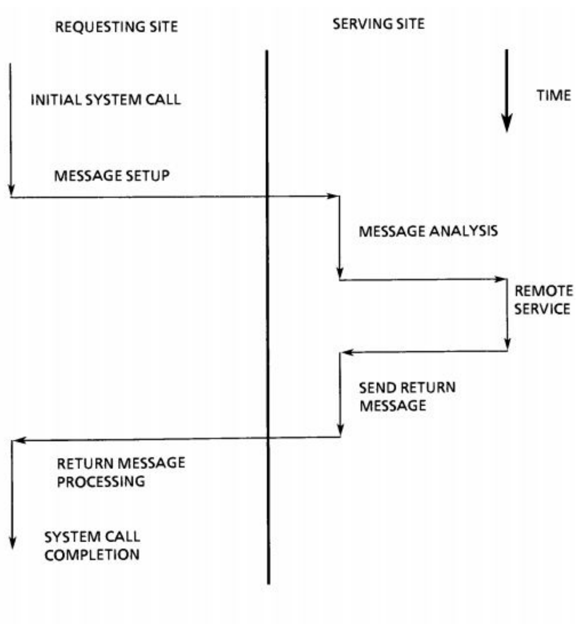


Fig. 5.4 Communication sequence

CASE STUDIES

3.7.1. Sun Network File System

Network File System (NFS) is a distributed file system (DFS) developed by Sun Microsystems. This allows directory structures to be spread over the net- worked computing systems.

A DFS is a file system whose clients, servers and storage devices are dis- persed among the machines of distributed system. A file system provides a set of file operations like read, write, open, close, delete etc. which forms the file services. The clients are provided with these file services. The basic features of DFS are multiplicity and autonomy of clients and servers.

NFS follows the directory structure almost same as that in non-NFS system but there are some differences between them with respect to:

- Naming
- Path Names
- Semantics

Naming

Naming is a mapping between logical and physical objects. For example, users refers to a file by a textual name, but it is mapped to disk blocks. There are two notions regarding name mapping used in DFS.

- **Location Transparency:** The name of a file does not give any hint of file's physical storage location.
- **Location Independence:** The name of a file does not need to be changed when file's physical storage location changes.

A location independent naming scheme is basically a dynamic mapping. NFS does not support location independency.

There are three major naming schemes used in DFS. In the simplest approach, files are named by some combination of machine or host name and the path name. This naming scheme is neither location independent nor location transparent. This may be used in server side. Second approach is to attach or mount the remote directories to the local directories. This gives an appearance of a coherent directory. This scheme is used by NFS. Early NFS allowed only previously mounted remote directories. But with the advent of automount , remote directories are mounted on demand based on the table of mount points and file structure names. This has other advantages like the file-mount table size is much smaller and for each mount point, we can specify many servers. The third approach of naming is to use name space which is identical to all machines. In practice, there are many special files that make this approach difficult to implement.

Mounting

The mount protocol is used to establish the initial logical connection between a server and a client. A mount operation includes the name of the remote directory to be mounted and the name of the server machine storing it. The server maintains an export list which specifies local file system that it exports for mounting along with the permitted machine names. Unix uses /etc/exports for this purpose. Since, the list has a maximum length, NFS is limited in scalabilty. Any directory within an exported file system can be mounted remotely on a machine. When the server receives a mount request, it returns a file handle to the client. File handle is basically a data-structure of length 32 bytes. It serves as the key for further access to files within the mounted system. In Unix term, the file handle consists of a file system identifier that is stored in super block and an inode number to identify the exact mounted directory within the exported file system. In NFS, one new field is added in inode that is called the generic number.

Mount can be is of three types -

1. **Soft mount:** A time bound is there.

2. **Hard mount:** No time bound.
3. **Auto mount:** Mount operation done on demand.

NFS Protocol and Remote Operations

The NFS protocol provides a set of RPCs for remote operations like lookup, create, rename, getattr, setattr, read, write, remove, mkdir etc. The procedures can be invoked only after a file handle for the remotely mounted directory has been esta- blished. NFS servers are stateless servers. A stateless file server avoids to keep state informations by making each request self-contained. That is, each request iden- tifies the file and the position of the file in full. So, the server needs not to store file pointer. Moreover, it needs not to establish or terminate a connection by opening a file or closing a file, repectively. For reading a directory, NFS does not use any file pointer, it uses a magic cookie.

Except the opening and closing a file, there is almost one-to-one mapping between Unix system calls for file operations and the NFS protocol RPCs. A remote file operation can be translated directly to the corresponding RPC. Though conceptu- ally, NFS adheres to the remote service paradigm, in practice, it uses buffering and caching. File blocks and attributes are fetched by RPCs and cached locally. Future remote operations use the cached data, subject to consistency constraints.

Since, NFS runs on RPC and RPC runs on UDP/IP which is unreliable, operations should be idempotent.

Cache Update Policy

The policy used to write modified data blocks to the server's master copy has critical effect on the system performance and reliability. The simplest policy is to write through the disk as soon as they are placed on any cache. It's advantageous because it ensures the reliability but it gives poor performance. In server site this policy is often followed. Another policy is delayed write. It does not ensure reliability. Client sites can use this policy. Another policy is write-on-close. It is a variation of delayed write. This is used by Andrews File System (AFS).

In NFS, clients use delayed write. But they don't free delayed written block until the server confirms that the data have been written on disk. So, here, Unix semantics are not preserved. NFS does not handle client crash recovery like Unix. Since, servers in NFS are stateless, there is no need to handle server crash recovery also.

Time Skew

Because of differences of time at server and client, this problem occures. This may lead to problems in performing some operations like " make ".

Performance Issues

To increase the reliability and system performance, the following things are generally done.

1. Cache, file blocks and directory informations are maintained.
2. All attributes of file / directory are cached. These stay 3 sec. for files and 30 sec. for directory.
3. For large caches, bigger block size (8K) is benificial.

This is a brief description of NFS version 2. NFS version 3 has already been come out and this new version is an enhancement of the previous version. It removes many of the difficulties and drawbacks of NFS 2.

3.7.3. Andrews File System (AFS)

AFS is a distributed file system, with scalability as a major goal. Its efficiency can be attributed to the following practical assumptions (as also seen in UNIX file system):

- Files are small (i.e. entire file can be cached)
- Frequency of reads much more than those of writes
- Sequential access common
- Files are not shared (i.e. read and written by only one user)
- Shared files are usually not written
- Disk space is plentiful

AFS distinguishes between client machines (workstations) and dedicated server machines. Caching files in the client side cache reduces computation at the server side, thus enhancing performance. However, the problem of sharing files arises. To solve this, all clients with copies of a file being modified by another client are not informed the moment the client makes changes. That client thus updates its copy, and the changes are reflected in the distributed file system only after the client closes the file. Various terms related to this concept in AFS are:

- **Whole File Serving:** The entire file is transferred in one go, limited only by the maximum size UDP/IP supports

- **Whole File Caching:** The entire file is cached in the local machine cache, reducing file-open latency, and frequent read/write requests to the server
- **Write On Close:** Writes are propagated to the server side copy only when the client closes the local copy of the file

In AFS, the server keeps track of which files are opened by which clients (as was not in the case of NFS). In other words, AFS has **stateful servers**, whereas NFS has **stateless servers.** Another difference between the two file systems is that AFS provides **location independence** (the physical storage location of the file can be changed, without having to change the path of the file, etc.) as well as location transparency (the file name does not hint at its physical storage location). But as was seen in the last lecture, NFS provides only **location transparency**. Stateful servers in AFS allow the server to inform all clients with open files about any updates made to that file by another client, through what is known as a **callback**. Callbacks to all clients with a copy of that file is ensured as a **callback promise** is issued by the server to a client when it requests for a copy of a file.

The key software components in AFS are:

- **Vice:** The server side process that resides on top of the unix kernel, providing shared file services to each client
- **Venus:** The client side cache manager which acts as an interface between the application program and the Vice

All the files in AFS are distributed among the servers. The set of files in one server is referred to as a volume. In case a request can not be satisfied from this set of files, the vice server informs the client where it can find the required file.

The basic file operations can be described more completely as:

- Open a file: Venus traps application generated file open system calls, and checks whether it can be serviced locally (i.e. a copy of the file already exists in the cache) before requesting Vice for it. It then returns a file descriptor to the calling application. Vice, along with a copy of the file, transfers a callback promise, when Venus requests for a file.
- Read and Write: Reads/Writes are done from/to the cached copy.
- Close a file: Venus traps file close system calls and closes the cached copy of the file. If the file had been updated, it informs the Vice server which then replaces its copy with the

updated one, as well as issues callbacks to all clients holding callback promises on this file. On receiving a callback, the client discards its copy, and works on this fresh copy.

The server wishes to maintain its states at all times, so that no information is lost due to crashes. This is ensured by the Vice which writes the states to the disk. When the server comes up again, it also informs all the servers about its crash, so that information about updates may be passed to it.

A client may issue an open immediately after it issued a close (this may happen if it has recovered from a crash very quickly). It will wish to work on the same copy. For this reason, Venus waits a while (depending on the cache capacity) before discarding copies of closed files. In case the application had not updated the copy before it closed it, it may continue to work on the same copy. However, if the copy had been updated, and the client issued a file open after a certain time interval (say 30 seconds), it will have to ask the server the last modification time, and accordingly, request for a new copy. For this, the clocks will have to be synchronized.