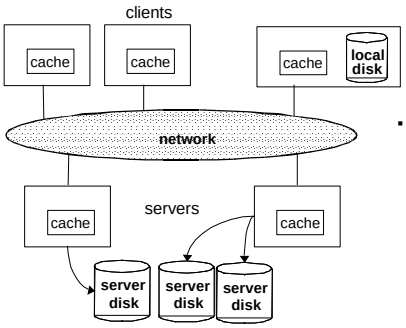
On previous lecture

- Distributed system - is a collection of independent machines (loosely coupled hardware) that appears to the user to be a single system (tightly coupled software)
- distributed system is provides - a cheap, scalable, powerful and reliable alternative to single and multiprocessor computers (tightly coupled hardware)
- cluster is a type of distributed system - it is a small scale homogeneous collection of machine that performs a small set of well-defined tasks as a single entity

Lecture 21: Distributed File Systems

- Network File System (NFS)
  - v2
  - v3
- Andrew File System (AFS)
  - the original
  - DCE Distributed File System (DCE DFS)

What is distributed file system



- *Distributed file system* is a part of distributed system that provides a user with a unified view of the files on the network.

A machine that holds the shared files is called a *server*, a machine that accesses the files is called a *client.*

Design goals for a distributed file system

- network transparency – clients should be able to access remote files using the same operations that apply to local files.
- location transparency – the name of the file should not reveal its location in the network.
- user mobility – users should be able to access shared files from any node in the network.
- fault tolerance – the system should continue to function after failure of a single component (a server or a network segment). It may, however, degrade in performance or make part of the file system unavailable.
- scalability – the system should scale well as its load increases. Also, it should be possible to grow the system incrementally by adding components.
- file mobility – it should be possible to move the files from one physical location to another in a running system.

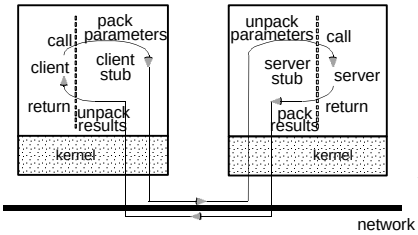
Distributed file system design considerations

- stateful or stateless operation
  - stateful server retains information about the client operations between requests and uses this state information to service subsequent requests. Requests such as *open* or *seek* are inherently stateful - ether client or server has to remember the state information. Stateless server requests - are self-contained - no information needs to be kept in between.
  - stateful server is faster (it can take advantage of the knowledge of client's state to eliminate network traffic); stateless servers can recover from failures easier.
- semantics of sharing
  - *Unix semantics* - the changes made by one client to a file are immediately visible to other clients accessing this file
  - *session semantics* - changes are propagated to other clients when the file is either opened or closed.

Client-server model

- A client-server model is a programming paradigm where the computing task is split between two entities:
  - server - a passive entity that responds only when client requests a certain service to be done
  - client - an active entity, requests server to perform certain cervice
- Client-server computing model is simple, scalable, and reliable
- In case of distributed file systems:
  - a machine that holds the shared files is called a *server*
  - a machine that accesses the files is called a *client*.

Remote Procedure Calls (from previous lectures)



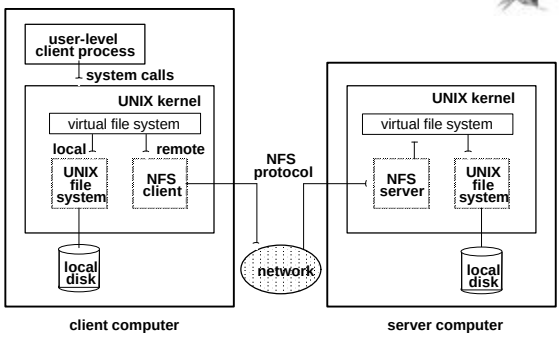
Each RPC invocation by a client process calls a *client stub*, which builds a message and sends it to a *server stub*

- The server stub uses the message to generate a local procedure call to the server
- If the local procedure call returns a value, the server stub builds a message and sends it to the client stub, which receives it and returns the result(s) to the client

Network File System (NFS) v2

- developed by Sun Microsystems in 1985, now supported by most Unix and some non-Unix systems.
- NFS is a *protocol* by which machines share files
- NFS is stateless - no requests to open and close a file. **`Read/write`** operations (unlike Unix **`read`** and **`write`**) pass the offset of the file as a parameter
- simple crash recovery - no persistent information about the client is maintained on server - in client crashes the server is not affected; if server crashes - the client keeps repeating the request until it is satisfied (the server is back online)
- problem with statlessness - the **`write`**s have to be committed to stable storage before reporting successful completion of the **`write`**. Why? This includes inodes as well as data blocks of a file
- client and server communicate via synchronous RPCs

Network File System (NFS) v2 (cont.)

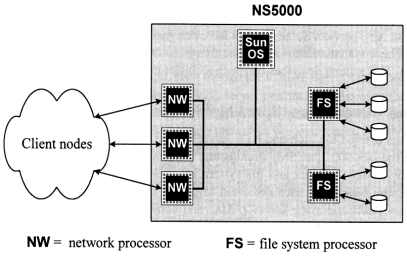


- Virtual file system - abstracts the functions of a file system from it's implementation, can be implemented over various FS

NFS v2 performance improvements, v3

- Client-side caching:
  - every NFS operation requires network access - slow. client can cache the data it currently works on to speed up access to it.
  - problem - multiple clients access the data - multiple copies of cached data may become inconsistent.
  - solutions - cache only read-only files; cache files where inconsistency is not vital (inodes) and check consistently frequently
- deferral of **`write`**s
  - client side - the clients bunch **`write`**s together (if possible) before sending them to server (if the client crushes - it knows where to restart)
  - a sever must commit the **`write`**s to stable storage before reporting them to a client. Battery backed non-volatile memory (NVRAM) is used (rather than the disk). NVRAM -> disk transfers are optimized for disk head movements and written to disk later.
- v3 allows delayed writes by introducing commit operation - all writes are "volatile" until the server processes commit operation.
- Requires changes in semantics - applications programmer cannot assume that all the writes are done and should explicitly issue commit .

NFS network appliances (dedicated NFS servers)



- Auspex NS500
- NW and FS run functional multi-processing kernel (FMK) - a "stripped down" version of Unix (no shared memory, processes never terminate) which allows NW and FS to context switch fast and service requests quickly. NW receives a request - if its NFS it passes it to FS if not a processor running regular Unix (SunOS) services it.

Andrew File System (AFS)

- developed in Carnegie-Mellon University in 1982
- AFS was made to span large campuses and scale well therefore the emphasis was placed on offloading the work to the clients
- as much as possible data is cached on clients, uses session semantics - cache consistency operations are done when file is opened or closed
- AFS is stateful, when a client reads a file from a server it holds a *callback*, the server keeps track of callbacks and when one of the clients closes the file (and synchronizes it's cached copy) and updates it, the server notifies all the callback holders of the change *breaking* the callback, callbacks can be also broken to conserve storage at server
- problems with AFS:
  - even if the data is in local cache – if the client performs a write a complex protocol of local callback verification with the server must be used; cache consistency preservation leads to deadlocks
  - in a stateful model, it is harder to deal with crushes.

Caching in Andrew

- When a remote file is accessed, the server sends the entire file to the client
  - The entire file is then stored in a disk cache on the client computer
    - Cache is big enough to store several hundred files
- Implements session semantics
  - Files are cached when opened
  - Modified files are flushed to the server when they are closed
  - Writes may not be immediately visible to other processes
- When client caches a file, server records that fact — it has a *callback* on the file
  - When a client modifies and closes a file, other clients lose their callback, and are notified by server that their copy is invalid

How can Andrew perform well?

- Most file accesses are to files that are infrequently updated, or are accessed by only a single user, so the cached copy will remain valid for a long time
- Local cache can be big — maybe 100 MB — which is probably sufficient for one user's working set of files
- Typical UNIX workloads:
  - Files are small, most are less than 10kB
  - Read operations are 6 times more common than write operations
  - Sequential access is common, while random access is rare
  - Most files are read and written by only one user; if a file shared, usually only one user modifies it
  - Files are referenced in bursts

DCE Distributed File System (DCE DFS)

- Modification of AFS by Open Software Foundation for its Distributed Computing Environment (DCE)
- one of the major modifications - cache consistency mechanism is streamlined by using tokens. Each file contains a tokens for,
  - data, status, lock, etc.
- a client can "check out" a token from a server for a particular file. Only if a client holds a lock token the client is allowed to modify the file; if a client holds status token it is allowed to modify status of the file
- tokens provide finer degree of control than callbacks of AFS

AFS vs. NFS

- NFS is simpler to implement, AFS is cumbersome due to cache consistency synchronization mechanisms, etc
- NFS does not scale well - it is usually limited to a LAN, AFS scales well - it may span the Internet
- NFS performs better than AFS under light to medium load. AFS does better under heavy load
- NFS does writes faster.