SIENA*college*
Computer Science

Computer Science 330
Operating Systems
Siena College
Spring 2012

Topic Notes: Distributed Systems

Now, we consider distributed systems, where we have a collection of computers connected by a network. These could be connected by a private, fast network, in which case we might call it a *cluster*, or be connected by general-purpose networks, even the Internet.

The interconnection network itself is not our focus – take a Networks course. For now, we'll assume that the computers can communicate with each other.

A networked system allows for

- resource sharing
  - share files
  - remote hardware devices: tape drives, printers
- computational speedup
- reliability – if a node fails, others are available while repairs are made

The operating system for a distributed system could be:

- *Network Operating System* – users are aware of multiple nodes – access resources explicity:
  - remote login to appropriate machine (telnet, rlogin, ssh)
  - transferring files explicitly (ftp, scp)
  - just use standard operating system with network support
- *Distributed Operating System* – users need not know where things are on the network
  - system handles the transparency
  - data migration – automated file transfer (scp), file sharing (NFS, AFS, Samba)
  - computation migration – transfer computation across the system
  - remote procedure call (RPC) – a process on one node makes a function call (essentially) that runs on a different node
  - process migration – execute an entire process, or parts of it, on different nodes

A network operating system is simpler to construct, but puts more burden on the user.

A distributed operating system is more complex, but more automated.

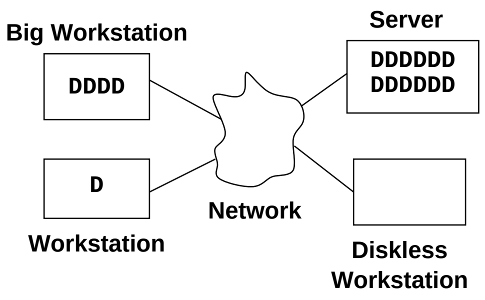
Design Issues for a Distributed Operating System

A key idea is that of *transparency* – where we try to hide the distinction between local and remote resources. This may include:

- *access transparency* – the ability to access local and remote resources in a uniform manner.
- *location transparency* – where users have no awareness of resource locations.
- *migration transparency* – a resource can be moved without changing its name or access method.
- *concurrency transparency* – users can share resources without interference.
- *replication transparency* – maintain consistency of multiple instances (or partitioned) files and data.
- *parallelism transparency* – achieve computational parallelism without need to have users aware of the details.
- *failure transparency* – the ability to have a graceful degradation rather than system disruption when faced with a component failure, minimizing damage to users – *fault tolerance*.
- *performance transparency* – provide consistent and predictable performance when the system structure or load changes.
- *size transparency* – allow the system to grow without users' knowledge – *scalability* – this is a difficult issue and bottlenecks may arise in large systems.

Distributed File Systems

Next, we look in more detail at network file systems.



Files on some of these disks may be shared. Perhaps the user files for all four systems reside on the server. A copy of the operating system resides on the big workstation and the regular workstation, with extra space on the big workstation used for some special software. Then the diskless workstation needs everthing, including its operating system, from the server. A *distributed file system (DFS)* is needed to manage a collection of such storage devices.

Terminology:

- *Service* – software running on one or more machines providing some function to other machines.
- *Server* – service software running on a single machine.
- *Client* – process that can invoke a service.
- *Client interface* – set of client operations provided by a service.

For a DFS, the client interface most likely includes a set of file operations, much like those available to access local disks (create, delete, read, write).

DFS Issues:

- remote and local files should be accesible through the same client interface.
- performance is important (access time, service time, latency).
- naming – file names need to have enough information to find the location.
- replication – are there replicas of some or all files? How are they kept synchronized?
- caching – use local disks to cache remote files, use memory cache on client or server.

Naming Schemes

- *location transparency* – file name does not reveal the file's physical storage location.

  Consider the following directories on `winterstorm.teresco.org`:

  `/home/terescoj`, `/home/terescoj/home`, `/babyred_home/jteresco`, and `/scratch`.

  All are accessible through the normal file system interface, with no host names in their paths, but they exist physically on disks attached to three different computers.

- *location independence/migration transparency* – a file name can remain the same when the physical location changes.

  For example, when the Unix machine in my office was replaced last summer, the physical location of many of my files changed, but the path to access them remained the same.

- Approach 1: file names include a location and a path.

  Examples:

  – `winterstorm:/mnt/terescoj`
  – `\\ntserver\path\to\file`

  Of course, there is no location transparency or independence here.

- Approach 2: remote directories are attached to local directories, in much the same way local filesystems are included.

  Examples:

  – Sun Network File System (NFS) (more soon)
  – Windows "attach network drive"
  – Mac "connect to server"

  File names do not include the server name, but the server name is needed to make the initial connection, or "mount".

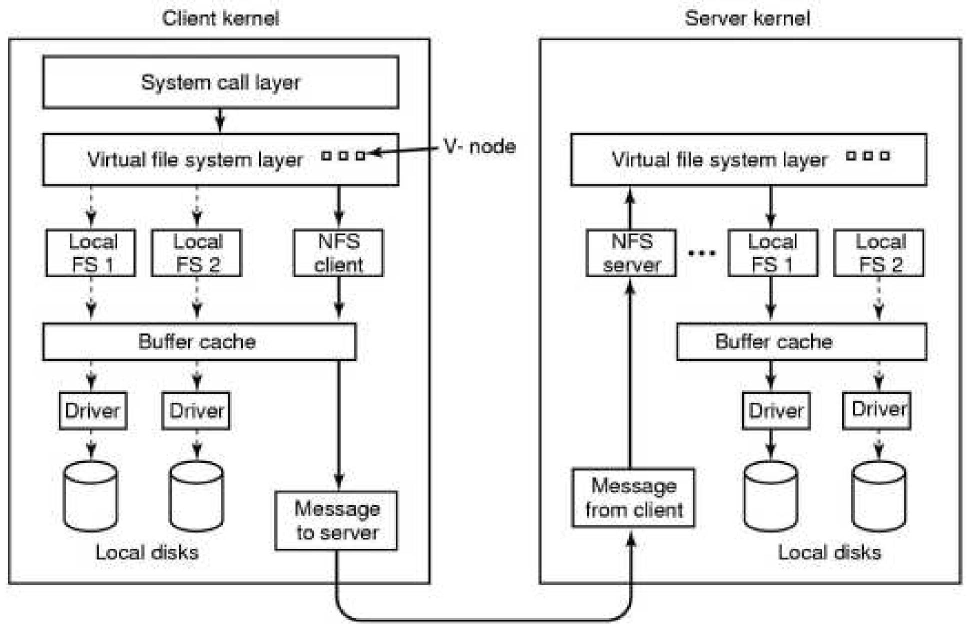
- Approach 3: total integration of file systems – a single global name structure spans all files.

  Example: AFS – worldwide pathnames! `/afs/rpi.edu/home/85/teresj`

  The "cell server" `rpi.edu` knows where to find files in the `rpi.edu` cell. Files can migrate among local servers and path does not change.

Network File System (NFS)

We share files in our lab using NFS, originally developed by Sun, now available in many systems.



- NFS is a means to share filesystems (or part of filesystems) among clients transparently.
- a remote directory is mounted over a local file system directory, just as we mount local disk partitions into the directory hierarchy.
- mounting requires knowledge of physical location of files – both hostname and local path on the server.
- usage, however, is transparent – works just like any filesystem mounted from a local disk.
- mount mechanism is separate from file service mechanism; each using RPC (remote procedure call).
- interoperable – can work with different machine architectures, operating system, networks.
- mount mechanism requires that the client is auhorized to connect (see `/etc/exports` in most Unix variants, `/etc/dfs/dfstab` in Solaris) – `mountd` process services mount requests.
- when a mount is completed, a *file handle* (essentially just the inode of the remote directory) is returned that the client operating system can use to translate local file requests to NFS file service requests – `nfsd` process services file access requests.
- NFS fits in at the virtual file system (VFS) layer in the operating system – instead of translating a path to a particular local partition and file system type, requests are converted to NFS requests.
- NFS servers are *stateless* – each request has to provide all necessary information – server does not maintain information about client connections.
- two main ways for clients to know what to request and from where: entries in file system table (`/etc/fstab` or `/etc/vfstab`), or automount tables. `fstab` entries are mounted when the system comes up, automount entries are mounted on demand, and are unmounted when not active.
- many NFS implementations include an extra client-side process, `nfsiod`, that acts as an intermediary, allowing things like read-ahead and delayed writes. This improves performance, though the delayed writes add some danger (see below).

Caching Remote File Access

Caching is important – we can use the regular buffer cache on the client, and could use client disk space as well. The server will automatically use its buffer cache for NFS requests as well as any local requests.

Cache on disk vs. main memory:

- Advantages of disk caches:

  - reliability (non-volatile)
  - can remain across reboots
  - can be larger

- Advantages of memory caches:
  - can be used for diskless workstations
  - faster
  - already have memory cache for local access, and putting a remote file cache there allows reuse of that mechanism

Cache Update Policy – mostly the same issues we have seen in other contexts.

- *Write-through* – write data through to disk as soon as write call is made – reliable, but performance is poor.
- *Delayed-write* – write to cache now, to server "later" – much faster write, but dangerous!
  - Advantage: some data (temp files, for example) may be overwritten or removed before ever being written to the disk at all.
  - Danger: unwritten data may be lost if a client machine crashes.
  - Can have system scan cache regularly to flush modified blocks.
  - write-on-close: flush all cache blocks for a file when it is closed.

Cache Consistency: we need to know if the copy of a disk block in our cache is consistent with the master copy.

- *client-initiated*: a client that wants to reuse a block checks with the server to ensure that the cached version is still valid.
- *server-initiated*: the server notifies clients of any changes from other processes, so the client need communicate with the server if it has not been notified of a change.

For a stateless system like NFS, the server has no knowledge of which clients are connected, let alone what is in their cache. There, any cache consistency protocol must be client-initiated.

Still, it is very difficult to guarantee anything here. Even if the client can check with the server, it is possible that a dirty block remains in another client's cache. NFS generally deals with this with an approach that any modified cache blocks are written back to disk "reasonably" quickly, usually in a few seconds. So in practice, problems arise only in systems where there are many concurrent writes. In these situations, the user processes should do some sort of file locking to ensure that the cache will not lead to inconsistencies.

Stateless vs. Stateful File Service

We said that NFS requests are stateless – each request is self-contained. No open and close operations for the server, as the file is reopened and gets data at a specific position in the file. This seems inefficient...

A stateful file server would "remember" what requests have been made, so a request could be something like "read the next block of this file".

This can improve performance by allowing fewer disk accesses on the server side: read-ahead to get blocks into the server's cache to anticipate upcoming requests.

The problems come when thinking about failure recovery:

- a stateful file server loses all of this state if it crashes.
- may need to contact all possible clients to reconstruct state.
- or could return error conditions to any client requests and force them to start over.
- The server will need a way to decide that a client has gone away – it is maintaining information about each client, and the client may have forgotten to close a file, or forgotten to unmount a partition, or simply crashed.
- a stateless server can usually recover seemlessly from a failure.
- in our lab, if `babyred` reboots, your processes may stop and you might get a "NFS server babyred not responding" message, but as soon as they come back up, the clients can continue what they were doing.

Replication

Similar issues arise when we want to replicate some files to enhance reliability, efficiency, availability.

- reliability/availability: if one server goes down, just use another that has a replica.
- efficiency: use the closest or least-busy server.
- this is a technique used by web servers – a request for a file at `www.something.com` may be silently redirected to one of a number of servers, like `www2.something.com`, `www28.something.com`, etc.
- main issue: keeping replicas up to date when one or more is changing.
- if there is a "master copy" we can use caching ideas – just treat replicas like cached copies.
- if there is no master, any change must be made to all replicas.

AFS

The *Andrew File System*, now known just as *AFS*, is a globally-distributed file system. Originally from CMU, later supported by a company called Transarc, which has since become part of IBM. IBM has released AFS as an open source product.

- we saw the naming convention that includes a site name in the path.
- use of a file cache on local disks – important as network latency is now (potentially) over the Internet.
- the system caches entire files locally, not individual disk blocks.
- file permissions are now very important, as many users can browse – AFS supports more complicated file permissions, including ACLs:

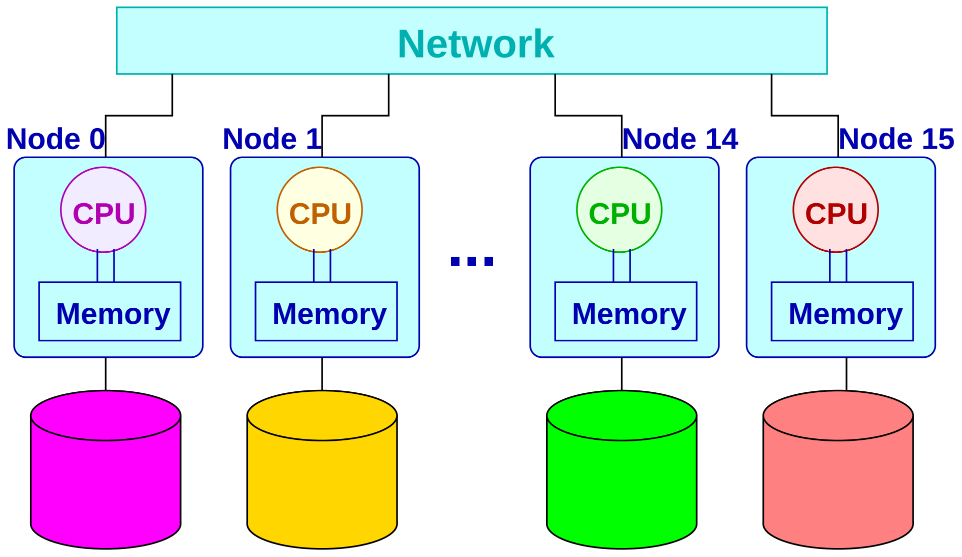
  ```
  17:50:32 24.cortez:~ -> fs la
  Access list for . is
  Normal rights:
    system:backup l
    system:anyuser rl
    teresj rlidwka
  ```

  Permissions are set for directories rather than individual files, and can be set for read, list, insert, delete, write, lock, and administer. See `http://cac.engin.umich.edu/resources/storage/afs/afs_acl.html` for more on AFS access rights.
- files can move among servers in the same cell without its name changing.

For more on OpenAFS, see `http://www.openafs.org/`

Parallel File Systems

Consider a situation where we have a cluster, where each node has one or more CPUs, a local memory, and a large local disk.



User processes should be able to run on any node, and have access to a common file space (home directory, data sets, etc.).

One option is to have all nodes access shared files from an external file server, or to designate each node in the cluster as having certain files on its local disk, and the nodes share the files using, *e.g.*NFS.

The former bullpen cluster used a combination of these approaches. All nodes had access to departmental file servers, plus files on the front end node of the system. Each node had a local disk, and these were shared (NFS *automounted*) among the nodes.

Potential problems:

- File server or network to the file server may become a bottleneck if many nodes are accessing files at once.
- If we try to get around this by writing to local disks, we lose some of the transparency – we need to know which node's disks we used, etc.

Potential solution: borrow ideas from RAID, and make one big *parallel file system* out of the disks attached to cluster nodes!

Just as RAID hides the details of which disk actually stored a given file, a parallel file system hides which disk and which node stores a file.

Issues to consider:

- how do we allocate disk blocks?
- how much shared directory information is needed on all nodes?
- how do we access files on remote disks? cache locally? migrate to local disk?

- new files written to local disk?
- use striping to keep all disks active and spread out the load?
- what about fault tolerance? if one node goes down, is the entire file system unavailable?
- are we making the network an unmanageable bottleneck?
- how much complexity are we adding to the kernel's file system modules?
- what about concurrent writes? concurrent reads/writes?
- some decisions may depend on expected access patterns – scientific computing is likely to result in large reads/writes, but rarely will have a write conflict, whereas a database is likely to have many small transactions.

Examples of systems that do this kind of thing are linked from the lecture page about this topic.