Distributed File System

- Ch15: File-System Internals
- Ch15.2 File-System Mounting
- Ch15.4 File Sharing
- Ch15.6 Remote File Systems
- Ch15.7 Consistency Semantics
- Ch19.6 Distributed File Systems

Remote File Systems

- Sharing of files across a network

1. Sharing each file manually – programs like ftp
2. **Distributed file system** (**DFS**)
    - Remote directories visible from local machine
3. World Wide Web
    - A bit of a revision to first method
    - Use browser to locate file/files and download /upload
    - Anonymous access doesn't require authentication

Distributed File System

A Distributed File System (**DFS**) is a file system which has data stored in several different sites or hosts (computers and associated devices) on a network.

- **Service** – software entity running on one or more machines and providing a particular type of function to a priori unknown clients
- **Server** – service software running on a single machine
- **Client** – process that can invoke a service using a set of operations that forms its client interface

- A client interface for a file service is formed by a set of primitive file operations (create, delete, read, write)
- Client interface of a DFS should be transparent; i.e., not distinguish between local and remote files
- Sometimes lower level inter-machine interface need for cross-machine interaction

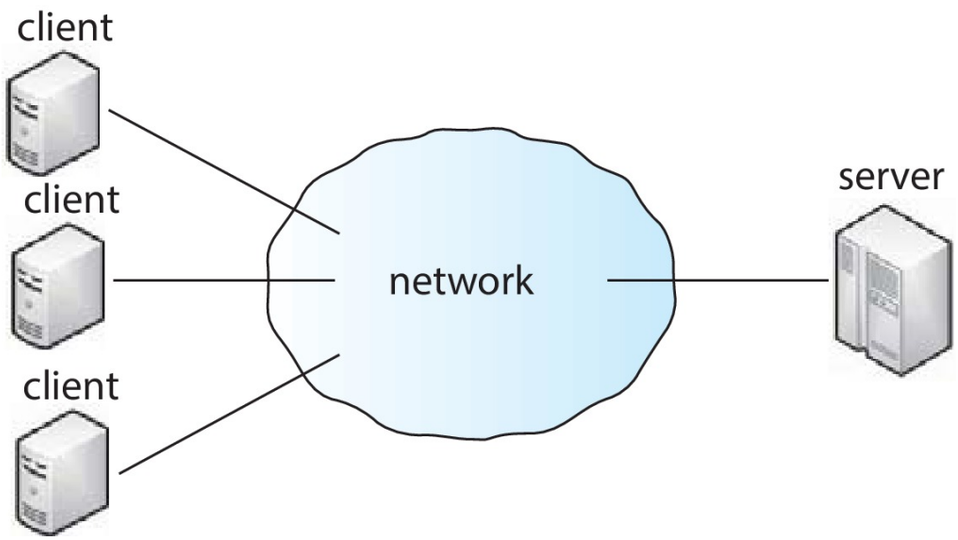
Distributed File System

Advantages

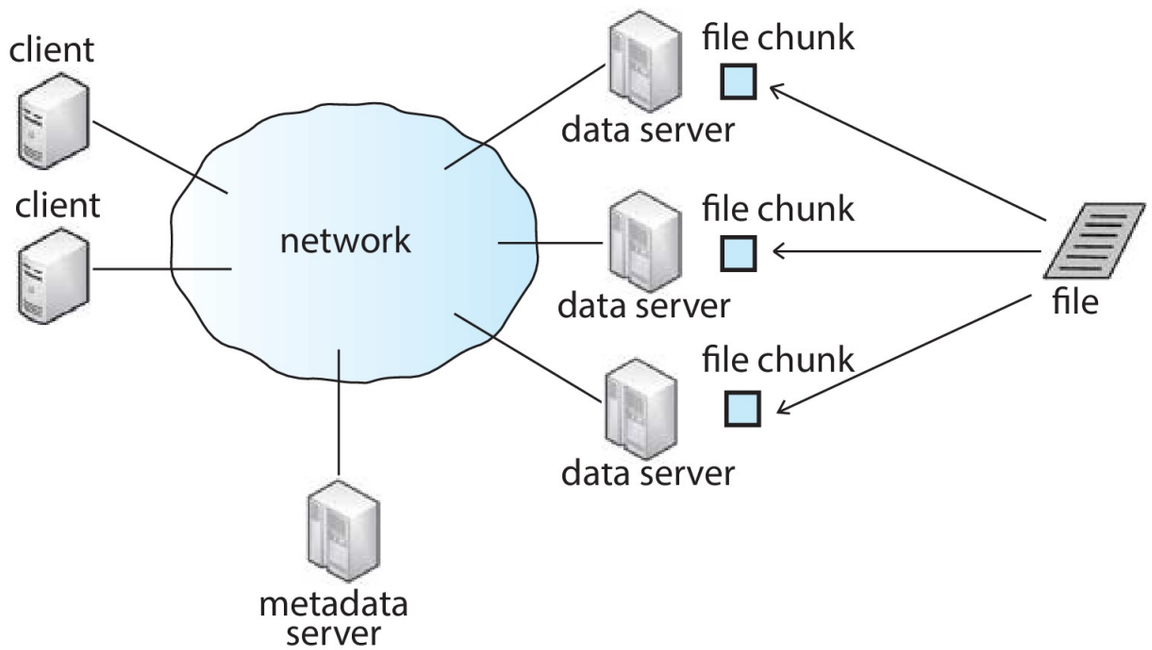
- Greater amount of storage
- Greater flexibility for administration and sharing purposes
  - Users can log on to any machine and have access to all of their files
  - Files can be stored close to where they are normally used but are still available elsewhere
- Replicate information for greater reliability
  - If a site goes down the files may still be accessible from another location

Distributed File System

- Two widely-used architectural models include **client-server model** and **cluster-based model**
- Challenges include:
  1. Naming and transparency
  2. Remote file access
  3. Caching and cache consistency



client-server model



cluster-based model

Client-Server Model

- Sharing between a **server** (providing access to a file system via a network protocol) and a **client** (using the protocol to access the remote file system)
- Identifying each other via network ID can be spoofed, encryption can be performance expensive
- NFS an example
  - User auth info on clients and servers must match (UserIDs for example)
  - Remote file system mounted, file operations sent on behalf of user across network to server
  - Server checks permissions, file handle returned
  - Handle used for reads and writes until file closed

Cluster-Based Model

- Built to be more fault-tolerant and scalable than client-server DFS

- Examples include the Google File System (**GFS**) and Hadoop Distributed File System (**HDFS**)
  - Clients connected to master metadata server and several data servers that hold “chunks” (portions) of files
  - Metadata server keeps mapping of which data servers hold chunks of which files
    - As well as hierarchical mapping of directories and files
  - File chunks replicated n times

Naming and Transparency

- Naming – mapping between logical and physical objects
- Multilevel mapping – abstraction of a file that hides the details of how and where on the disk the file is actually stored
- A transparent DFS hides the location where in the network the file is stored
- For a file being replicated in several sites, the mapping returns a set of the locations of this file's replicas; both the existence of multiple copies and their location are hidden

Naming Structure

- An ideal – the distributed system should look like a single machine and associated files.
- It is called transparency because the user should not be able to see the differences (and complications).

Location transparency

- No (obvious) connection between the name of a resource (file) and its position on the system.

Migration transparency

- Resources (files) can move around the system without programs needing to be changed (or stopped and restarted).
- This is similar to what the textbook calls *Location independence* - the name doesn't have to be changed with the location changes.
- It needs location transparency to implement.

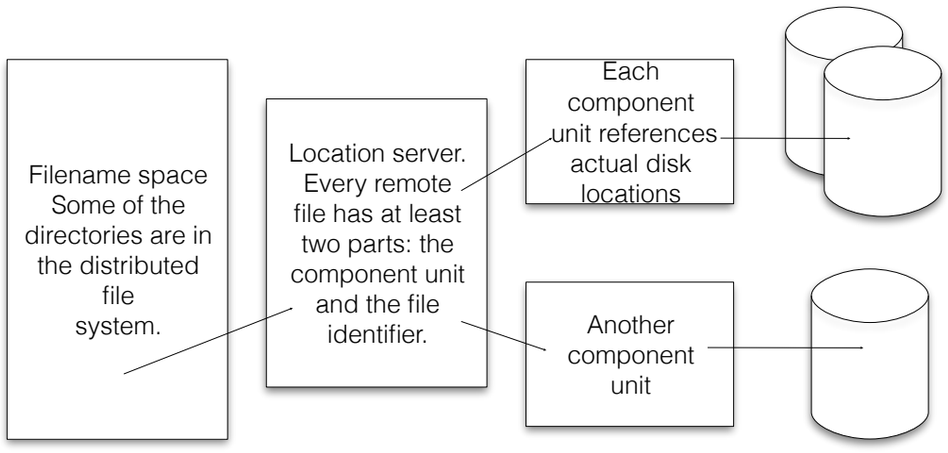
Collections of Files

- All common distributed file systems group files into collections for simplicity and administration purposes.

- The collections (component units in the textbook) are commonly subtrees stemming from particular directories.

- So the subtrees are shared and moved and replicated together.

- If we are going to transparently migrate component units we need to have a structure something like this:



Remote File Access

- Consider a user who requests access to a remote file. The server storing the file has been located by the naming scheme, and now the actual data transfer must take place.

1. **Remote-service mechanism** is one transfer approach.
    - A requests for accesses are delivered to the server, the server machine performs the accesses, and their results are forwarded back to the user.
    - One of the most common ways of implementing remote service is the **RPC** paradigm

Remote File Access

2. Reduce network traffic by retaining recently accessed disk blocks in a **cache**, so that repeated accesses to the same information can be handled locally
    - If needed data not already cached, a copy of data is brought from the server to the user
    - Accesses are performed on the cached copy
    - Files identified with one master copy residing at the server machine, but copies of (parts of) the file are scattered in different caches
- **Cache-consistency problem** – keeping the cached copies consistent with the master file

Caching

- Blocks of files are cached locally.
- All accesses come from the cache.
- If a block is not in the cache it is requested from the server and then cached.
- Old blocks can be replaced using Least Recently Used (LRU) algorithm.

Cache Update Policy:

- Q: When do we send the information back to the server?
  - **Write-through** – every write requires the block to be sent back to the server
  - **Delayed-write** – send the block to the server at a later time (check every 30 secs, or when the file is closed or when the cached block is needed for another block)
- Q: How do we cope with writing to the cache when other processes (on other sites) also have the file open?

Pros and Cons

- **Caching** - usually faster, more efficient, scales better than remote service

- **Remote Service** - simpler to implement because of no consistency problem, uses less local memory (primary or secondary), matches local file access

- Some systems use both schemes, basically providing a remote service solution but with some caching for efficiency reasons.

Consistency Semantics

Consistency Semantics: the way changes in data get distributed between processes accessing the same files.

Two major types:

- **UNIX Semantics** – any change made by any process is immediately visible to any other process.
- **Session Semantics** (Andrew file system (OpenAFS))
    - Writes to open file not visible during session, only at close
    - Can be several copies, each changed independently
- Both can be worked with but programmers need to know which is used on the system they are programming.

Remote Service - Stateful

Server knows:

- who has the file open
- for what type of access
- where it is in the file etc.
- When the client calls open it receives an identifier to be used to access the file.
- Looks very similar to traditional local file access to the client process.
- Efficient, the needed data may be read ahead by the server.
- Information about the file is held in memory.
- If the server crashes
  - it is difficult to start again since all the state information is lost.
- Server has problems with processes which die
  - needs to occasionally check.

Remote Service - Stateless

- Server does not maintain information on the state of the system. It merely responds to requests.
- Open and close calls don't send messages to the server. Handled locally. (Except for access privileges.)
- Requesting processor has to pass all the extra information with each read/write
  - e.g., the current file location the process is reading from,
  - accessibility information
- Server doesn't have to worry about processes stopping
  - it doesn't keep any records and is not taking up any memory space on the server.
- No complicated recovery process if the server goes down.
  - A new server (or the recovered old one) just starts handling requests again.

Before Next Time

Distributed File Systems

- Ch15.8 NFS
- Ch19.6 Distributed File Systems