

Distributed File Systems

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.

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
- can replicate information for greater reliability
 - if a site goes down the files may still be accessible from another location

Different types

Different approaches to providing access to files over networks.

Remote file transfer approach

- No requirement for the machines to even be running the same operating system.
- A user can explicitly connect to another machine on the network and download files. e.g. ftp
- Can't directly use a file on the remote site.
- We end up with multiple copies and no method of maintaining consistency.
- The user must know exactly where a file is (including the host).

Direct access using explicit site names

- Each file is prefixed with a site identifier e.g. cs26.auckland.ac.nz:/home/robert-s/310exam
- Can use a file directly from another site.
- The user still must know exactly where a file is located.
- No replication possible.

Better methods

It is better if we don't have to specify the host name when accessing remote files.

Keep information with each directory pointing to the machine where the files are actually stored.

- files can be used directly on the remote host
- the user sees no difference between accessing remote and local files
- remote files may not be visible from all machines on the network, even if they are they may have different pathnames
- moving remote directories entails changes on all local machines

Keep a server(s) with location information and associate a standard directory base with all remote files.

- direct use and the same view of the file system regardless of where you are logged on from
- can move files without any information needing to be sent to local machines
- can replicate files

Transparency

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*.

Need location transparency to implement.

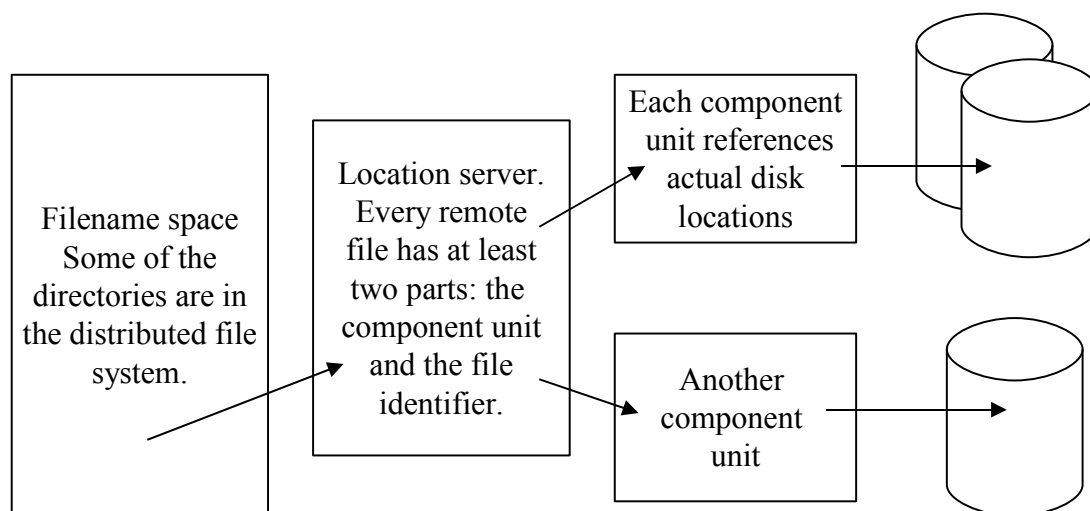
Collections of files

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:



Using remote files

Once we have discovered where a file is we have to move all data accessed from the file over the network.

This is expensive and has problems with consistency.

We could:

- send only required blocks back and forth between the server and the client (**remote service**), every file access results in messages across the network
- keep copying large chunks when required and **cache** them at the client
- copy complete files when accessed and copy back when done (only if written to)

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 algorithm.

If we modify data in the cache

- 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)
- 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

The way changes in data get distributed between processes accessing the same files is known as consistency semantics.

Two major types:

- UNIX semantics – any change made by any process is immediately visible to any other process.
- Session semantics – the process gets a copy of the file when it is opened and changes are not visible to other processes until a file is closed.

Both can be worked with but programmers need to know which is used on the system they are programming.

Stateful remote service

Server knows:

- who has the file open
- for what type of access
- and 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 requesting processor.

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.

Stateless

Server has no idea.

Client open and close calls don't send messages to the server. Handled locally.

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

Read from the textbook

15.1.2.1 – Network Operating Systems

16.2 – Naming and Transparency

11.5.3 – Consistency Semantics

16.3 – Remote File Access

16.4 – Stateful Versus Stateless Service