

# Distributed filesystems

Mads Pultz & Niels Boldt  
Datalogisk institut

September 25, 2002

# Overview

- Distributed filesystems(DFS), including
  - Naming services.
  - Semantics of sharing.
  - Remote access method.
  - Caching.
  - Fault tolerance issues.
  - Availability and replication.
  - scalability.

# Overview

- Case studies. including
  - NFS
  - Andrew
  - Coda

# **Demands**

- Should look like conventional centralized filesystem.
- Should implement user mobility.
- Should be fault tolerant.
- Should be scalable.

## Naming schemes

There are three main approaches to naming schemes

- Files are named as a combination of their host name and local name. This scheme is neither location transparent nor location independent.
- The NFS way. Remote directories can be mounted to local directories. This provides a location transparent mapping.
- Global namespace identical for all users. Location independent.

## Naming schemes

- *Location*: A name of a machine and a reference to a location in the local filesystem.
- *Location transparency*: The name of the file does not reveal it's location, but the name must be changed when the files location is changed. This provides a static mapping of filenames to locations.
- *Location independence*: The name of the file does not reveal it's location, and the name does not change if the files location change. This provides a dynamic mapping mapping of filenames to location.

# Naming schemes

*Example:* XMLStore is location independent. Unix file system is location transparent.

## Semantics of sharing

- The semantics of sharing denote the effects of multiple clients sharing a file simultaneously.
- A area of special interest is modification of data. When or should the modifications at all, be observable by other clients.

## Unix semantics

- Every read sees the effect of all previous writes.
- But how to define previous in a distributed system. No global clock.
- Writes to an open file are visible to all clients who have this file open.

*Pros:* Each successive access sees the effect of the accesses preceding it.

*Cons:* The interleaving of accesses are arbitrary, because global clock is missing in distributed system. Very strict semantic, hard to implement in distributed environment. Mostly lead to an implementation where a file is associated with a physical object. Delay of accesses. Hard to replicate the file.

## Session semantics

A *session* is defined as all the accesses in a file between an open followed by an close of the file.

- Writes to an open file are visible to local clients.
- But not to remote clients who have the same file open.
- Modifications are only visible for remote clients after the file is closed.
- And only for remote clients who opens the file after the local client closed it.

## Session semantics

*Pros:* Replication is possible.

*Cons:* Programs caring about serialization should coordinate their accesses. Easy to lose modifications.

## Session semantics

*Example:* Consider three machines  $A$ ,  $B$ , a server  $S$  and a file  $1.v1$  residing on the server  $S$ .

1.  $A$  request  $1.v1$  open in write mode.
2.  $A$  modifies  $1.v1$ . The modified file will be denoted  $1.v2$ .
3.  $B$  request  $1.v1$  open in write mode.
4.  $A$  closes  $1.v2$  and the  $1.v1$  on  $S$  is replaced with  $1.v2$
5.  $B$  modifies  $1.v1$ . The modified file will be denoted  $1.v3$ .
6.  $B$  closes  $1.v3$  and the  $1.v2$  on  $S$  is replaced with  $1.v3$

*Conclusion:*  $A$ 's modifications are lost.

## Immutable shared file semantics

- After declaration of sharing a file cannot be modified.
- The name must not be reused.
- A name denotes a collection of data, not a location.
- Example: *XMLStore*

*Pros:* Simple to implement.

*Cons:* Garbage collection must be implemented.

## Transaction like semantics

A *file session* is defined as all the accesses in a file between an open followed by an close of the file.

- A file session is considered a transaction.
- The effect of a series of interleaved file sessions is the same as executing the file sessions in some serial order.

*Pros:* Easy for users to work concurrently on files without loss of data.

*Cons:* Hard to implement. Strict semantic

## Transaction like semantics

*Example:* Consider three machines  $A$ ,  $B$ , a server  $S$  and a file  $1.v1$  residing on the server  $S$ . Assume an implementation of transaction semantics where files cannot reside at different machines in conflicting modes.

1.  $A$  request  $1.v1$  open in write mode.
2.  $A$  modifies  $1.v1$ . The modified file will be denoted  $1.v2$ .
3.  $B$  request  $1.v1$  open in write mode. Cannot be fulfilled.
4.  $A$  closes  $1.v2$  and the  $1.v1$  on  $S$  is replaced with  $1.v2$
5.  $B$ 's request is fulfilled.

6.  $B$  modifies  $1.v2$ . The modified file will be denoted  $1.v3$ .

7.  $B$  closes  $1.v3$  and the  $1.v2$  on  $S$  is replaced with  $1.v3$

*Conclusion:*  $A$ 's modifications are not lost.

## Remote access methods

Consider a client process that request access to a remote file. There are two schemes for fulfilling the request:

- Remote service: Every request for access are delivered to the server, which in turn perform the accesses. The results are sent back to the client. We will call that a remote access.
- Caching: A copy of the data requested is sent to the client. The following request are satisfied from the cache. We will call that a local access.

Caching and remote service is a choice between simplicity or potential for improved performance.

# Caching design

To design a good caching scheme the following topics must be considered:

- The granularity of cached data.
- The location of data on the client. Memory or disk. Tradeoff between performance and reliability.
- Propagation of modified data.
- Consistency of cached data.

## Propagation of modified data

Data must be written back to the master copy after modification. The policy for flushing dirty block has critical effect on system performance and should be considered together with the semantics of sharing.

The following policies can be used:

- Write data through to the server as soon as it's written. This is denoted *Write Through*.
- Only write data to cache and delay the write at the server. This is denoted *Delayed Write*.

## Propagation of modified data

*Write through:* Good in corporation with unix semantics, using remote service for write accesses.

*Delayed Write:* Different schemes can be used, but this scheme is less reliable since unwritten data will be lost if the client crashes.

*Example:* NFS uses a delayed write for the propagation of modified data.

## Cache validation and consistency

When the client has cached data it must be assured that data are consistent with the master copy. There are two approaches which can be used.

- The client initiates the check which can be done at each access or after a fixed amount of time.

*Cons:* Every access coupled with a validity check is delayed. Validity check can cause a lot of network traffic if accesses occur frequently.

## Cache validation and consistency

- The server is responsible for the validity of the cache. This means that the server must keep a list of all opened files and their mode. The clients are notified when potential for inconsistency occurs.

*Cons:* The client server approach is destroyed. Scalability can be harder to achieve because a centralized entity have responsibility for the validity of cached data.

## Stateful vs. stateless

- If a server between requests from clients, stores information about the requests, we will call it stateful

*Pros:* Performance is improved because data such as pointers into the current position of the file are kept in main memory. Messages between machines can be smaller.

*Cons:* After a crash, time must be spent, recovering the state of the server before the crash.

## Stateful vs. stateless

- If a server stores no information we call it stateless.

*Pros:* Nothing to recover after crash.

*Cons:* Overhead on the network because of greater messages. Lesser performance, because each request is considered a new request.

# Availability

We say a file is *available* if it can be accessed whenever needed despite storage and machine crashes.

A file is not available if

- The machine where the file is kept are crashed.
- The network connection fails.
- The file cannot be located on the network because of machine failures.

## Improving availability

Availability can be improved if

- Directory information is cached.      pros:  
faster pathname traversal, overcomes unavailable  
directories
- Replication of files is implemented.      cons:  
Propagation of modified data.

To solve the consistency problems which occurs with replication, a readonly replication can be considered.

# Scalability

The following principles should be obeyed when designing scalable distributed filesystems.

- Bounded resources should be pursued meaning that a service demanded from one component should be bounded by a constant.
- Minimize cross machine interaction by hints, relaxed semantics and caching. Tradeoff: strictness of semantics. vs network load.
- Central resources and control schemes should be avoided.
- Peer to peer design is the ideal.