A Distributed File System

(that has the name spaces and semantics that resemble those of the Windows File System)

Design Overview Document

Submitted by:
Surabhi Ghaisas (07305005)
Rakhi Agrawal (07305024)
Election Reddy (07305054)
Mugdha Bapat (07305916)
Mahendra Chavan(08305043)
Mathew Kuriakose (08305062)
Sameera Deshpande (08405003)

Issues in Implementation of Distributed File System

1. Stateful vs stateless file system
2. File sharing semantics
3. Namespace server and permissions
4. File caching
5. Locking and granularity

Stateful vs Stateless File System

Stateful file system

In stateful file system, the state of the file processing activity is remembered in metadata. In stateful file system,

- Client opens the file.
- Server fetches information about the file from its disk.
- It loads that file in its memory
- A connection identifier unique to the client and the open file is given to the client.
- This identifier is used by client to perform subsequent accesses to file until the session ends.
- Its server's responsibility to reclaim the main memory used by clients that are no longer active.

Advantages:

- It is simple.
- Fewer disk accesses are needed at server side.
- Server knows if the file was opened for sequential access, and hence can read ahead the next blocks.
- It reduces the network traffic as client need not send state information with every call to server.

Disadvantages:

- When the client crashes, the file processing activity should be abandoned, and the file state should have to be restored to its previous consistent state, so that the client can restart its file processing activity.
- The resources allocated to aborted file processing should have to be released.
- When the server crashes, the state information stored in file server metadata is lost, hence the file processing activity should be terminated. Thus client knows about server crashes.

Stateless file system

In stateless file system, the state information about file accessing activity is not maintained by the server. Instead, client maintains state information about the file processing activity. The information about the desired activity is provided to the server when the service is requested through file system call.
In stateless file service,

- Client performs system call which contains information about which file to be accessed, which operation to be performed, which part of the file to be accessed, etc.
- At this call, server opens the desired file, and performs the action on the specified part of the file.
- When client receives the response from server, it assumes that the operation requested has been completed successfully.
- If the file server crashes, timeouts and retransmission occurs at client site.
- When server site recovers, it answers to the retransmitted request of client. Hence, client is unaware of server crashes.

Advantages:

- Stateless server is more robust than stateful file server, hence they provide fault tolerance.
- Lost connections can't leave a file in an invalid state.
- Rebooting the server does not lose state information.
- Rebooting the client does not confuse a stateless server.

Disadvantages:

- As the server doesn't maintain any state information about file processing activity, it cannot detect and discard duplicate requests from client.
- To ensure consistency of files, the requests made by clients must be idempotent.
- The stateless file server incurs performance penalty due to:
  - The file server opens the file at every file operation.
  - When client performs write operation, the changes are needed to be reflected in disk copy of the file.
  - Stateless file server cannot employ file caching or disk caching.

In windows, stateful file system is used. We plan to implement stateful file system. The state of the file consists of:

1. File id (information about file can be retrieved from metadata of file system)
2. Id of next record to be read
3. Location where the file exists
4. Client for which the file is being accessed

File Sharing Semantics

There are different types of file sharing semantics:

- Unix Semantics - Every operation on a file is instantly visible to all processes.
- Session Semantics - No changes are visible to other processes until the file is closed.
- Immutable files - A file once created cannot be changed. Such a file cannot be opened for modification, it can be opened only for reading. In case of immutable files cache coherency becomes easier to implement.
- Atomic Transaction - To access a file a process executes a begin transaction to notify that all futher operations will be executed indivisibly. After completion it executes an end transaction.

We plan to use session semantics for our distributed file system. In this case, as mentioned above, changes to a file are not visible until the file is closed. So we need to limit the concurrent access to a file by different processes in the system by use of a distributed locking mechanism.

Namespace Server and Permissions

There are two strategies which we can adopt for implementing the name space server:

1. Stand-alone (Centralized)
2. Domain-based (Distributed).

We are planning to go for domain-based namespace server. In our case it would be a two level model. At the root level would be the RootDNS. Root DNS would be maintaining the list of all the files/folders created immediately under /. For each of the files/folders in this list, it would keep the addresses of the respective DomainDNSs on which they were created. Please note that RootDNS will *not* store the links and permissions for any of the files/folders. Below the root level would be a number of DomainDNS servers. Every DomainDNS would serve for a number of nodes. Each DomainDNS server would maintain a virtual tree of all files/folders created by the nodes in its domain under /.

We would be providing read, write, modify, read&execute, fullcontrol permissions for files and read, write, modify, read&execute, fullcontrol, listfoldercontents for folders. Permissions for folders and files would be maintained along with the links in the DomainDNS namespaces. Whenever a user tries to access a file/folder, permissions will be checked at DomainDNS level and will be given access to actual file/folder only if he/she has the requested permission to access.

Example

As shown in the Figure 1 nodes will be divided into domains say A,B,C,... .Each domain will have one DomainDNS. There will be one RootDNS. DomainDNS_A will maintain the information about folders (level 1) and files(level 1) created by nodes in DomainA.

DomainDNS_A also maintains information about folders and files created under its level 1 folders by any node in any Domain.

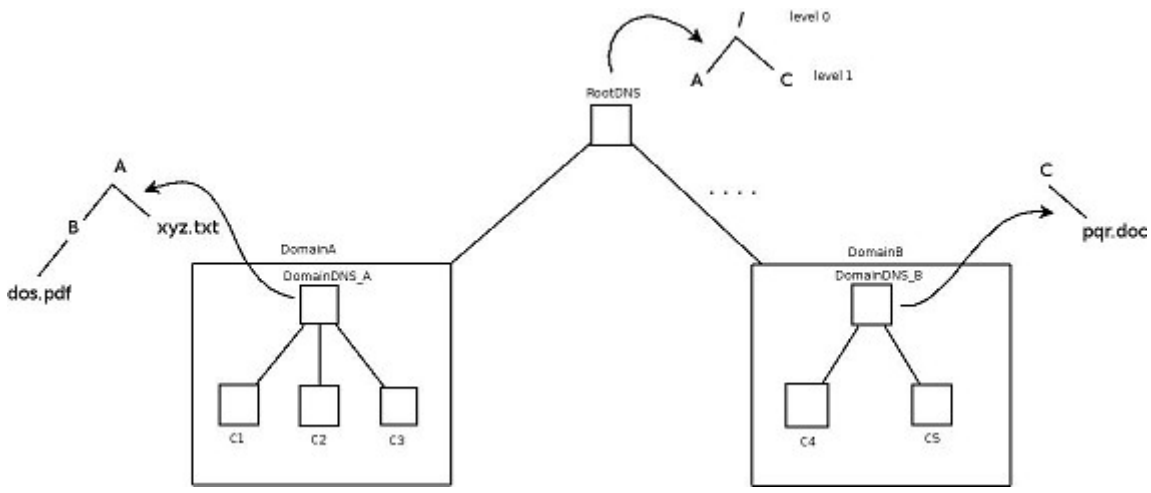

Figure 1

Few scenarios

In the following scenarios, we assume that each node has one user with name same as the node name.

1. Node C1 requests DomainDNS_A to create a folder /A

   DFS Actions:
   a. A folder would be created on C1 and DomainDNS_A would create a folder entry for folder /A in its virtual tree along with a link to the folder and permissions bitmap (Full Control in this case) in its virtual tree.
   b. DomainDNS_A would pass on the folder name /A to RootDNS.
   c. RootDNS would insert (/A,DomainDNS_A) in its list of files/folders.

2. Node C2 requests DomainDNS_A to create a folder /A/B

   DFS Actions:
   a. A folder would be created on C2 and DomainDNS_A would create a folder entry for folder /A/B along with the link to the folder and permissions bitmap (Full Control in this case) in its virtual tree. (Please note that, information about /A/B is not passed to the RootDNS)

3. Node C4 requests DomaionDNS_B to create a file A/B/dos.pdf

   DFS Actions:

a. File dos.pdf would be created on C4.
b. DomainDNS_B would lookup its virtual tree for the directory /A/B. It would not find it. It would pass the file creation request to the RootDNS. (It would pass the link of the created file and the permissions also)
c. RootDNS would lookup its list for the folder /A. It would find and entry (A,DomainDNS_A). It would pass the request to DomainDNS_A
d. DomainDNS_A would lookup /A/B. It would find the node /A/B and would create an entry in the namespace under B for dos.pdf with a link for the file dos.pdf (which is on C4) and Permissions bitmap for the current user.

4. Node C2 gives ListFiolderContents permissions to Node C5 for folder /A/B

   DFS Actions:
   a. DomainDNS_B would create a entry (C5, PermissionsBitmap) in the permissions list for folder /A/B in the namespace tree.

5. Node C5 requests DomainDNS_B to read the file /A/B/dos.pdf

   DFS Actions
   a. DomainDNS_B would lookup the file in its virtual tree. It would not find it, would pass on the request to RootDNS.
   b. RootDNS would lookup to find out the entry (A,DomainDNS_A). It would pass on the file access request to DomainDNS_A.
   c. DomainDNS_A would lookup the file /A/B/dos.pdf and check the permissions for user C5. Since C5 does not have permissions, it would deny the file request. The Permission denied response would be sent back to the RootDNS and then to DomainDNS_B which wil send it to C5.

File Caching

We can employ caching to improve system performance. There are four places in a distributed system where we can hold data:

1. On the server's disk
2. In a cache in the server's memory
3. In the client's memory
4. On the client's disk

The first two places are not an issue since any interface to the server can check the centralized cache. It is in the last two places that problems arise and we have to consider the issue of cache consistency. Several approaches may be taken:

1. Write-through: All accesses would require checking with the server first (adds network congestion) or require the server to maintain state on who has what files cached. Write-through also does not alleviate congestion on writes.

2. Write-back (delayed writes): Data can be buffered locally (where consistency suffers) but files can be updated periodically. A single bulk write is far more efficient than lots of little writes every time any file contents are modified. Unfortunately the semantics become ambiguous.
3. Write on close: This is admitting that the file system uses session semantics.
4. Centralized control: Server keeps track of who has what open in which mode. We would have to support a stateful system and deal with signaling traffic.

We plan to implement cache in each client's disk. Cache consistency will be ensured using *write on close* approach. This requires that at any point of time server should maintain the list of clients that have accessed file since the last latest write to the file. Server should ensure the following:

- Not allow caching when concurrent-write sharing occurs i.e. disable cache when a file is opened in conflicting modes.
- Allow many readers.
- If a client opens for writing, inform all the clients to purge their cached data.

Locking and Granularity

Each file has a lock associated with it. Before accessing the file, client should acquire the lock in one of the following modes:

- Shared lock – for reading file
- Exclusive lock - for writing to file

Shared mode is compatible with shared mode, while shared mode with exclusive mode is incompatible. Exclusive mode with another exclusive mode is also incompatible.
If a request is made for a file which is already granted an incompatible lock then the request fails.