1

Distributed File Systems

1.1 Problems and Goals

When you want to have a file syustem and manage files, you need to consider following problems:

- Naming: allow uses to find files with a human-friendly name.
- Accessing: create, delete, read, write, append
- Physical Allocation
- Security and Protection: ensure privacy
- Resource Administration: enforce quotas and implement priorities.

Besides the problems in mind when design a DFS (actually general file system), we have some goals for DFS. **The big difference isn't what it does but the environment in which it lives.** A distributed file system typically operates in an environment where the data may be spread out across many, many hosts on a network, and the users of the system may be equally distributed.

- Coordinate file systems on machines.
- Hide the existence of distributed file system from the user.

Why we need a DFS. Actually this problem can be generalized into a problem of distributed system.

- More storage
- More fault tolerance
- Users are distributed who need to access file system from many places.

1.2 Operations

1.2.1 Unit of Transmission

There is a lot of data movement across the network, **how much do we move one time?** There are two intuitive answers to this question: whole files and blocks.

- File: Only the users know how to use the data in one file. File systems do not need to get known how a file is organized.
- Block: Reduce the payload for one operation. When the first block of the file arrives, the user can start to operate it.

1.2.2 Implementation Idea

Caching

Cache is the same usage. Cache data and reduce access to the servers. There are two kinds of caching in distributed systems:

- Cache and Validate Approach: ask servers that if the data is newest. This approach is used in *NFS*.
- Callback: if servers have some modification for files, they inform the users. This approach is used in *AFS* and *Coda*.

1.3 Coda

1.3.1 Disconnected Operations: cache and write conflict

hoard daemon.

keep certain files in the client's cache, requesting them as necessary, just in case the client should later find itself unable to communicate with the server. Cache and validate. This ensure the users get access the data efficiently.

write conflict.

keep a version number. Before a client writes a file to the server, it checks the version of the file on the server.

- If that version number matches the version number of the file that the client read before the write, the client is safe and can send the new version of the file. The server can then increment the version number.
- If the version number has increased, the client missed a callback promise. Then users must take care of the conflicts.

1.3.2 Replication: Volume Storage Group

This part is discussed in the Replication chapter. There are some topics for this.

First, **how it requests a file?**

1. It asks all replicas for their version number.
2. It asks the replica with the greatest version number for the file.
3. If there is a conflict, the client can direct the servers to update or inform them of the conflict.

Second, **how it writes a file?**

1. The clients sends the file to all servers, along with the original CVV.
2. Each server increments its entry in the file's CVV and ACKS the client.
3. The client merges the entries from all the servers and sends the new CVV back to each server.

Third, **what if partition happens?** If one or more servers fail, the client cannot contact the servers. The collections of volume servers that the client can communicate with is known as *Available Volume Storage Group*.

If the network is partitioned, Coda will still work, but generate some conflicts and inconsistency. (As is said in Chapter Replication.)

If the partitioned or failed servers become accessible, the files need to be updated. The client needs to check VVV or CVV for conflicts. If there is a conflict, the client drops all callbacks in the volume,
<u>because the servers should have all the callbacks</u>.

Conflicts

There are several kinds of conflicts appearing in this problem.

- read before write. When the client decide to write the updates to servers, it needs to check if the update reads before write. Otherwise, it may miss some callbacks. Situation could be that the client is broken down for a while and misses some callbacks.
- server consistency. After the updates written back to servers and the client getting the CVVs from servers, the client may find conflict in the CVVs. Situation could be that there is concurrent operation on the files.
- partitioned network recovered. Replication inconsistency conflict, users need to take actions.

1.3.3 Weakly Connected Mode

If a client finds itself with a limited connection to the servers, it will pick a server and send the update. This server will propagate the update to other servers.

1.4 File System Interface

There are some file systems that are designed at user level, rather than in the kernel.

1.4.1 MogileFS

Difference

Consider the file systems where files are not changed by users. Users are only allowed to upload and download the files.

Method

There are some key idea behind the implementation:

- RAID. Replication.
- Namespaces, rather than directory tree. In different applications. they can use a same MogileFS, but they can use different name.

Because there is little change happened, locks will be rarely used. It makes a lot of work efficient.

1.4.2 HDFS

Difference

HDFS supports the computation of Hadoop.

Idea

There are some key idea behind the implementation:

- The data needs to be very heavily distributed.
- The data needs to be local to each other, so the system requires location-awareness.
- The system needs to be implemented in a portable way at the user-level, to be operated at scale.