Chapter 15: Distributed File System

Sunday, February 10, 2008
4:03 PM

- **Distributed File System** (DFS) = a file system that allows users to share files and storage resources in distributed computer system.
  - Important features: files are stored at a lot of sites in a distributed system.
  - Vocabs
    - **Service** = a software that run on one or more machines.
    - **Server** = a service software that runs on a single machine.
    - **Client** = a process that can invoke a service.
  - DFS Interface: Same as traditional file service.
    - Create, delete, read, write
    - Via OS system calls.
  - Implementation approach
    - As a part of a distributed OS.
    - A software layer that connects the traditional OS to the file system software.
  - DFS should hide the multiplicity and the dispersion of sites and files from the user. Such as DFS is **transparent.** A transparent DFS supports **user mobility**.
  - How good is a DFS?
    - Measured by the time it takes to service a request.
    - This can be affected by:
      - Time to deliver request.
      - Time for response to come back.
      - Time to transfer file content and metadata.
      - Time to run the communication protocol.

- Naming
  - User should be able to refer to files by names that are easy to understand.
  - DFS needs to do name resolution that maps names to blocks on storages of some machines in the network.
  - Unlike traditional file system, where files are located in one disk of the machine the user uses, files can have **replicas** located on many computers. A DFS must map the name to one of the replicas.
  - Nice properties of naming in a DFS
    - **Location transparency** = the name of the file does not contain any metadata about the physical storage.
    - **Location independence** = the name of the file does not need to change if the file is moved from one physical storage to another.
    - Location independence is stronger than location transparency.
    - Nice things about location independence:
      - File name is completely decoupled from its physical storage.
      - The system can balance utilization of disks across the system.
  - Approaches
    - host:local-name
      - host is host name, just like the internet host name.
      - local-name is the Unix file name.
      - This is not location transparent.
    - Client can **mount** directories on servers. Then can refer to remote files using the same naming scheme as local files.
      - Used by Sun's NFS.
      - This mounting can be done on demand, providing great flexibility.

        - Every machine has its own local mapping of remote files.
        - This approach is difficult to administer.
            - Users can access some files at one workstation, but not at another.
            - When a server goes down, random directories on clients become unusable.
    - Indirection: A single naming system is responsible for resolving file names.
        - Used by Andrew File System (AFS).
        - Users see a single file system that feels like traditional file system on all machines
        - Difficult to achieve because some Unix files are special such as device files.
- The third approach is the most promising. So we'll discuss its implementation.
    - Cannot have a system that keeps track of the mapping of every file.
        - Too many files on the system to handle.
    - Instead, break file systems into **component units** = groups of files that are always stored together in one machine.
    - Typically, a component unit is partition along directory boundaries. That is, files in a directory or subdirectories of that directory are always stored together.
    - Now, we can implement a **two-level system-wide database**.
        - The first level maps directory names to an identifier that contains its corresponding component units. This is often called **location-independent file identifier.**
        - The second level maps component units to their physical locations.
    - A location-independent file identifier is a structured name. It contains:
        - A part that identifies the component unit.
        - A part that identifies a file within the component unit.

- Remote File Access
    - Two main approaches
        - **Remote-service machanism =** use RPC to carry out requests such as create(), delete(), read(), and write()
            - A lot of network overhead. Too slow to be usable.
        - **Caching**
            - Transfer all or some part of the file to local machine.
            - Transfer the changes back to the location that the file is actually stored.
    - Things to consider about caching.
        - To keep the cache size bounded, there needs to be a replacement policy.
            - The most used one seems to be LRU (least recently used).
        - Cache granularity = how large is an element of the cache?
            - AFS uses 64KB blocks. Other systems caches disk blocks (4KB).
            - Increase the size of a cache block increases hit ratio, but incurs a lot of overhead in transporting the block. Thus, a miss is very expensive.
            - Figuring out cache granularity, you need to think of:
                - Disk block size.
                - Network transfer unit = how many bytes are transferred in one packet?
        - Cache location = where to keep the cache?
            - Disk
                - Disk is non-volatile. So cache survives resets, power outages, and other failures.
                - But it is very slow.
            - Main-memory
                - Fast.
                - We have a lot more memory day-by-day.
                - Server will caches files in main-memory anyway. So, if we use main memory, we can use the same code for both client and server.

        - Old versions of NFS does not provide disk cache, but new versions comes with **cachefs** that caches files on the disk.
    - Cache-update policies
        - When and how to update the master copy?
        - **Write-through policy**
            - Write data to the master copy as soon as the data is placed on a cache.
            - High reliability. When a client crashes, not a lot of data is lost.
            - Slow. Use a lot of network bandwidth. Not scalable.
        - **Delayed-write policy**
            - Data is written to the cache.
            - Update to the master copy is performed later after many writes.Locally fast.
            - A lot less redundant information transfer.
            - A lot of unwritten data can be lost when a client crashes.
            - A lot of variation: When to update master copy?
                - ◇ When a block is about to be flushed from a cache.
                    - ► Reduce communication between client and server.
                    - ► But a block can sit in a cache for a long time without being flushed to the server.
                - ◇ A process scans the cache from time to time, and transfer blocks that have been updated to the server.
                - ◇ This approach is used by Sprite OS.
        - **Write-on-close policy**
            - Data is updated on the server only when a file is closed.
            - If a file is open for a short period, this does not help much.
            - But when a file is open for a long time, and written to a lot, this approach reduces a lot of network traffic.
            - Terminating a process can be delayed because the remote file update.
        - A combination of the above approaches can be used.
            - NFS uses delayed-write policy for file content, but uses write-through policy for file metadata. This prevents directory structure and file name from being inconsistent.
    - **Cache-consistency**
        - When an update is done to a file, all the caches at various machines in the system needs to be updated.
        - Keeping caches the same as the master copy is the problem of **cache-consistency.**
        - To ensure cache consistency, clients need some way to know if the cached copy it has is invalid or not.
        - Two approaches:
            - **Client-initiated** = The client asks the server if the master copy has changed since the last time the client accessed it.
                - ◇ Performance determined by how often this check is invoked. If checks too often, the server may become overloaded, and the network might become full.
                - ◇ Client can check every time it accesses the file. But this is very slow.
                - ◇ Alternatively, it can check for consistency periodically. But this has greater probability of the client using inconsistent cache.
            - **Server-initiated** = The server keeps track of which file is cached on which client. When an update takes place, the server send messages to clients that cache the file to tell them that the cached copies are invalid.
                - ◇ The server can also react to conflicting uses of the file by disable caching and force the clients to access the file by RPC instead.
- Caching pros and cons:

        - Caching reduces network use, and thus makes the system more scalable.
        - Caching allows the system to be further optimized because it now knows that it only have to transfer infrequent large chunks of data instead of frequent small responses.
        - When writes are frequent, ensuring cache consistency can incurs a lot of overhead.
        - Systems that use caching is a lot more complicated than systems that use RPC because low-level interface is different from high-level interface (OS system calls).

- Managing server-side information.
    - **Stateful service**
        - When a client opens a file, server fetches the file's information to main memory.
        - It then gives a client a **session** of communication.
        - Throughout the session, the server responds to client request using the session's information to help improve performance and network usage.
        - Since the server keeps a lot of information, the communication between client and server can be reduced.
            - For example, the server can keep the file pointer, and the client just have to specify how many more bytes it wants to read or write.
        - The session ends when the client closes the file.
        - AFS is a stateful file system.
    - **Stateless service**
        - Every request is self-contained. It contains information about which file and which bytes of the file the client wants to access.
        - The server keeps no information about which client opens which file. It merely serve those requests as directed.
        - NFS is a stateless file system.
    - Comparisons
        - A stateful file system is much faster. It has less communication overhead.
        - However, stateful file system is less robust.
            - Consider when the server crashes midway of a session. The information stored in its main memory is loss. Complicated recovery protocol needs to be followed so that the file system can be restored to a consistent state.
            - A crashed client can waste the server's main memory. So the server has to detect sessions that become idle after a long time to potentially reclaim the main memory. This is called **orphan detection and elmination**.
        - A stateless file system does not have to do anything to recover from server or client crashes.
        - However, all operations in a stateless file system has to be **idempotent** so as to allow client to retry a request many times if it doesn't get response from the server. This can complicates the design of the file system somewhat.

- Replication
    - A file may be replicated on many different machines.
    - Benefits
        - Fault tolerance: If a site that contains the file crashes, the file is still available on other sites.
        - Performance: A client can choose from the servers that gives the best performance.
    - Needs to make sure that replicas are located on machines which are **failure independent**, in other words, machines that do not share fate with one another: if one machine fails, other machines do not fail together.
    - The file system needs to ensure that replication is transparent from the user.
        - It must map the file name to one replica.
        - When an update is done to a replica, it must also be done to all replica.
    - Replica consistency is quite the same as cache consistency.
        - We can achieve a lot of consistency by a lot of validity checking, but this incurs a lot

of network overhead.
        - We can sacrifice consistency or availability for performance.

- Case study: Andrew File System
    - Developed at Carnegie Mellon University.
    - Now an open source software. Available at http://www.openafs.org.
    - AFS provides a global file system that is the same when viewed from all machines in the system. This file system can be mounted to a local directory. For example, a user's home directory may be mapped to AFS. So the user can access his file from any workstation in the system.
    - Features
        - Location independent file names.
        - Client-side caching.
        - Secure authentication and data transmission.
        - Protection: Each file has its own access control list.
        - Clear interface allows machines with different hardware configurations to use the system.
        - Very scalable: 1000+ nodes of workstation.
    - AFS classify machines into two types: clients and servers.
        - Servers are dedicated files servers. Runs software called *Vice*.
        - Clients are workstation that users work on. Runs software called *Venus* that communicates with the servers via *Virtue* protocol.
    - Clients are servers are grouped into **clusters**.
        - Machines in a cluster are connected by LAN.
        - A cluster has one dedicated server. Most clients in the cluster should use the service on the cluster's servers.
        - Clusters are connected to one another by WAN.
    - In order to reduce load on the server, clients cache data in units of 64KB.
    - Files in the systems are partition into component units called **volumes**.
        - Files in a volume are always stored together.
        - Can be thought of a disk partition. But is quite different.
        - A volume is stored as a whole in a disk partition. May volumes can reside in the same disk partition.
        - Has an array that contains inode numbers of the files in a volume.
        - Most of the time, a volume is associated with files of a single user or a single directory.
    - In order to support location independence, a file name is resolved to a low-level identifier called **fid**. A fid has three components:
        - A *volume number*.
        - A *vnode* = an index into an array that contains inodes of files.
        - A *uniquifier* = a number that allows the same vnode to be used several times. It is useful when, for example, making direct links.
    - Location of a volume is kept in a system-wide **volume-location database**.
    - When a volume is moved to a new location:
        - The old location is left with a forwarding information. Any requests to any file in the volume is carried out locally and forwarded to the new location. In this way, the volume-location database can be updated some time after the move took place.
        - Once the volume has finished moving. It is temporarily disabled so that recent updates to files can be carried out. After that, the volume is available from the new location.
        - Volume moving is atomic. So it recovers graciously from server crashes.
    - Venus is a user-level process that runs on each client and does the caching of file content from the server. Each system call that involves files on AFS are forwarded by the kernel to Venus.

- Venus caches the file to disk as it is opened, and updates the copy only when the file is closed.
  - **An update may not be visible elsewhere immediately.**
  - After Venus has transferred the file content to the local disk, read/write to the file are performed by the local file system.
  - A cache is just a local directory on the workstation. Files in this directories are just placeholders for cached entries.
  - The local file system maintains a buffer cache orthogonal to Venus.
  - Venus manages both file data cache (on disk) and metadata cache (in main memory). The metadata cache allows software to query information such as file name or size rapidly.
- Callback mechanism
  - AFS uses server-initiated approach to deal with cache consistency.
  - When a client caches a file, we say that the client has a **callback** to the file.
  - A client can only update a file when it has a callback to the file.
  - When a system updates the file, the server is informed so, and the server remove callbacks from other clients by sending messages to tell them. After that, the server does not allow other clients to modify the file until it has recached the file again.
  - This mechanism requires the server to maintain callback information for each of client-file access. If it runs out of space, it may revoke callbacks from some clients.
- There is another set of cache validation:
  - When a client is rebooted, it sends cache validation requests to server for all the files in its cache.
- Name resolution
  - Names in AFS are just like regular unix names.
  - Venus does name resolution component-by-component. It resolves one directory in the path name at a time. Note that a directory in AFS is a file that contains mappings mappings between names and fids.
  - Venus asks the volume-location database for location of the server a volume resides in.
- Server
  - Implemented as a user process running on the server machine.
  - The process has many threads scheduled using non-preemptive scheduling.
  - These threads form a thread pool and are used to service user requests.
  - Using thread pools allow threads to use a common cache of file content.
  - However, the threads share fate. One thread's failure can result in the whole server going down.