Distributed File Systems

Aris Paphitis
CUT

Key Point

Distributed File Systems - NFS, GFS, HDFS

1 Network File System -NFS

Developed by Sun Microsystems, used in all modern Linux systems to join the file systems on separate computers into a logical whole. The basic idea behinf NFS is to allow a group of clients and servers to share a common file system. Each NFS server exports one or more of its directories for access by remote clients.Clients acess exported directories by mounting them. When a client mounts a remote directory, it becomes part of its directory hierarchy. The mount point is entirely local to the clients; the server does not know where it is mounted on any of its clients. To support heterogeneity, NFS defines an interface between clients and servers with two protocols. The first handles mounting. The second is for directory and file access. Clients send messages to servers to manipulate directories and read/write files; but they cannot *open* or *close* files. The advantage of this scheme is that the server does not have to remember anything about open connections in between calls to it. A server that does not maintain state information about open files is called *stateless*. In contrast to NFSv3, described above, versio 4 is a statefull file system.

2 Google File System - GFS

The Google File System, was specifically designed and implemented by `Google, Inc` to meet its needs, driven by their application workloads and technological environment. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability and availability. However traditional choices have been reexamined and different points were explored. Observations that led to specific design decisions: 1) Component failures are the norm rather than an exception 2) Files are huge; in the order of many GB. 3) Most files are mutated by appending new data rather than overwriting existing data. 4) Co-designing the applications used over the filesystem and the file system (API) itself benefits the overall system. (An application specifies a file name and a byte offset. The GFS client translates it to a chunk index within a file).

2.1 Design Assumptions

The system is built from many inexpensive commodity components that often fail. The system stores a rather small number of large files. Workloads consists of two kinds of reads: a) Large streaming reads and b) Small random reads. Workloads also have many large,sequential writes that append data to files. The system must handle multiple concurrent appends efficiently. High sustained bandwidth is more important than low latency.

2.2 Interface

Familiar file system interface but not implemented through a standard API. It supports operations like *create*, *delete*, *open*, *close*, *read*, and *write* files with the additional operations of *snapshot* and *record*.

2.3 Architecture

A GFS cluster consists of a single `master` and multiple `chunkservers` and is accessed by many `clients`. Each of them is typically a commodity Linux machine. Files are divided into fixed-size chunks. Each chunk is identified by a globally unique and immutable 64 bit `chunk handle`. Chunkservers store chunks locally on stable storage and read/write chunk data specified by a chunk handle and a byte range. Each chunk is replicated on multiple chunkservers for reliability. The master maintains all the file system's metadata. Namespaces, access control information, the mappings from files to chunks and the locations of each chunk. The master communicated periodically with each chunkserver to send instructions and verify its state through `HeartBeat` messages. Each application on a client machine, implements the GFS API and communicates with the master and the chunkservers to read or write data. Clients interact with the master for metadata but all data is transfered directly to them from the chunkservers. No caching at either side of the communication. Clients never read and write file data through the master. Instead, a client asks the master which chunkservers to contact. This information is cached to the client for a limited time. The client and chunkservers interact directly for any subsequent operations.

Notes

2.4 Key Design parameters

Chunk size: in GFS is 64MB. Advantages: reduces client - chunkserver interaction, reduces network overhead, and reduces the size of metadata stored on master. Disadvantages: A small file consists of a small number of chunks. Chunkservers can become hot spots if many clients are accessing the same file.

Metadata - 3 types of: namespaces, mappings, and locations. All metadata are stored in the master's memory. In-memory state enables an efficient implementation of chunk garbage collection, re-replication in the presence of failures and chunk migration for load balancing. Chunk locations are not kept persistently but chunkservers provide this data at startup and periodically therafter. This eliminates the problem of syncing as chunkservers join or leave the cluster, fail or restart. Operation Log is a historical record of critical metadata. It defines the order of concurrent operations. It is replicated accross multiple machines.

2.5 Consistency Model

Relaxed consistency. Guarantees: file namespace mutations are atomic.

(a file region is *consistent* if all clients will always see the same data, regardless of which replica they read from.) (a region is *defined* after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety.) (a region is consistent but undefined after successfull concurrent mutations: all client see the same data, but not what each one mutation has written) (a failed mutation makes a region inconsistent)

This relaxed consistency affects applications but it can be accomodated through simple techniques. *Appending is far more efficient and resilient to application failures than random writes.*