14.5 Case study: a Distributed File System for very large files

To conclude this introductory part, we study a simple distributed service: a file system that serves very large data files (hundreds of Gigabytes or Terabytes). The architecture presented here is a slightly simplified description of the Google File System and of several of its descendants, including the HADOOP Distributed File System (HDFS) available as an open-source project. The technical environment is that of a high speed local network connecting a cluster of servers. The file systems is designed to satisfy some specific requirements: (i) we need to handle very large collections of unstructured to semi-structured documents, (ii) data collections are written once and read many times, and (iii) the infrastructure that supports these components consists of thousands of connected machines, with high failure probability. These particularities make common distributed system tools only partially appropriate.

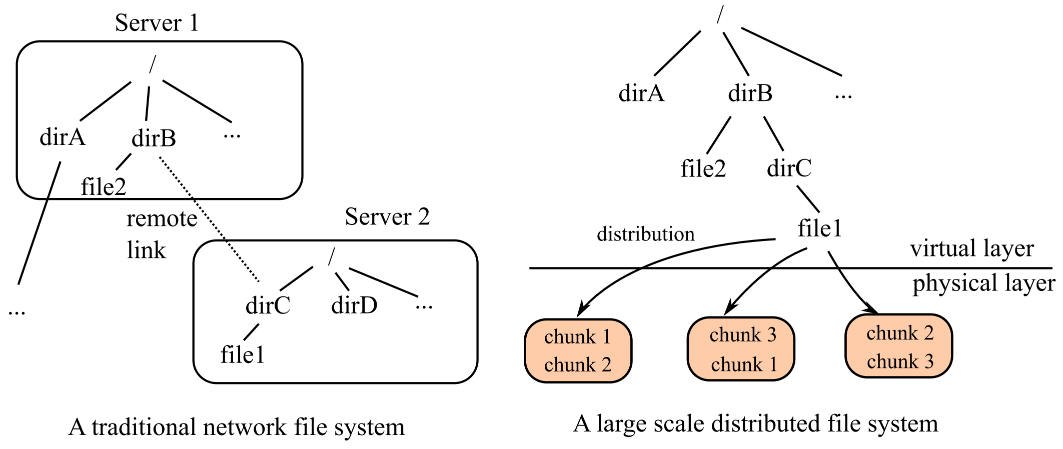

Figure 14.9: Distributed file systems for large files

14.5.1 Large scale file system

Why would we need a specialized architecture for distributing large files (DFS) in the first place? The answer is summarized by Figure 14.9 that shows, on the left side, a standard solution to share files among computers (a widespread implementation of this solution is NFS, the Network File System, in the Unix world). Assume that server A needs to access the files located in the directory *dirC* on server B. The DFS allows *dirC* to be "mounted" in the local file system as, say, a subdirectory of *dirB*. From the user point of view, this is transparent: s/he can navigate to the files stored in */dirA/dirB/dirC* just as if it was fully located on its local computer. The network calls that maintain *dirC* as part of the Server A namespace are handled by the DFS.

Modern distributed systems like NFS care about reliability and availability, and provide for instance mechanisms to replicate files and handle node failures. In the context of large scale data-intensive applications, this solution is nevertheless not convenient because it breaks several of the principles mentioned so far, and does not satisfy some of its expected properties. The main broken principle is data locality. A process running on Server A in charge of manipulating data stored on Server B will strongly solicit the network bandwidth. Regarding the properties, one notes that the approach is hardly scalable. If we store 10% of our data set in *file1* and 90% in *file2*, Server B will serve (assuming a uniform access pattern) 90% of the Client requests. One could carefully monitor the size and location of files to explicitly control load balancing, but this would lose the benefits of using a transparent file system namespace.

An NFS-like system is not natively designed to meet the specific requirements of a large scale repository. The right part of Figure 14.9 shows a different approach which explicitly addresses the challenge of very large files. Essentially, the difference lies in the fact that a file is no longer the storage unit, but is further decomposed in "chunks" of equal size, each allocated by the DFS to the participating nodes (of course, this works best for systems consisting of large files).

There exists a global file system namespace, shared by all the nodes in the cluster. It defines a hierarchy of directories and files which is "virtual", as it does not affect in any way the physical location of its components. Instead, the DFS maps the files, in a distributed manner, to the cluster nodes viewed as blind data repositories. File *file1* in the right part of Figure 14.9 is for instance split in three chunks. Each chunk is duplicated and the two copies are each assigned to a distinct node.

Because the DFS splits a file is equal-size chunks and evenly distributes the files, a fair balancing is natively achieved. Reliability is obtained by replication of chunks, and availability can be implemented by a standard monitoring process.

14.5.2 Architecture

We now turn to the architecture of GFS, summarized on Figure 14.10. The distributed system consists of a Master node and many server nodes. The Master plays the role of a coordinator: it receives Client connections, maintains the description of the global file system namespace, and the allocation of file chunks. The Master also monitors the state of the system with "heartbeat" messages in order to detect any failure as early as possible. The role of Servers is straightforward. They receive files chunks, and must take appropriate local measures to ensure the availability and reliability of their (local) storage.

A single-master architecture brings simplicity to the design of the system but gives rise to some concern for its scalability and reliability. The scalability concern is addressed by a Client cache, called *Client image* in the following. Let us examine in detail how the system handles a *read()* request, as illustrated on Figure 14.10 with dotted arrows:

1. The Client sends a first *read(/dirB/file1)* request; since it knows nothing about the file distribution, the request is routed to the Master (1).

2. The Master inspects the namespace and finds that *file1* is mapped to a list of chunks; their location is found in a local table (2).

3. Each server holding a chunk of *file1* is required to transmit this chunk to the Client (3).

4. The Client keeps in its cache the addresses of the nodes that serve *file1* (but *not* the file itself); this knowledge can be used for subsequent accesses to *file1* (4).

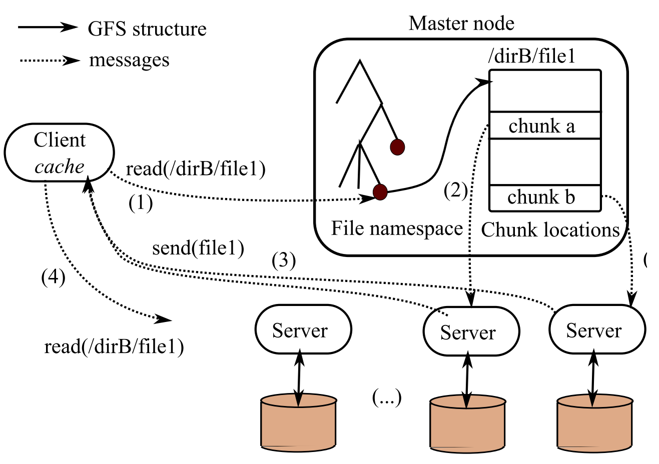

Figure 14.10: Architecture of the GFS system (after [71])

The approach is typical of distributed structures, and will be met in several other distributed services further on. The Client cache avoids a systematic access to the Master, a feature that would make the structure non scalable. By limiting the exchanges with the Master to messages that require metadata information, the coordination task is reduced and can be handled by a single computer.

From the Client point of view[1], the distributed file system appears just like a directory hierarchy equipped with the usual Unix navigation (`chddir`, `ls`) and access (`read`, `write`) commands.

Observe again that the system works best for a relatively small number of very large files. GFS (and similar systems) expects typical files of several hundreds of MBs each, and sets accordingly the chunk size to 64 MBs. This can be compared to the traditional block-based organization of centralized storage systems (e.g., databases) where is block size is a small multiple of the disk physical block (typically, 4KB-8KB in a database block).

The design of GFS is geared toward batch processing of very large collections. The architectural choices are in line with the expected application features. For instance, having large chunks limits the size of the internal structure maintained by the Master, and allows keeping them in memory. On the other hand, it appears clearly that using a GFS-like system for a lot of small files would be counter-productive. The Master would have to deal with a lot more references that could not be held in memory anymore, and each file would consist of a single chunk, with poor exploitation of the distribution leverage.

14.5.3 Failure handling

Failure is handled by standard replication and monitoring techniques. First, a chunk is not written on a single server but is replicated on at least 2 other servers. Having three copies of the same chunk is sufficient to face failures (the number of replicas can be chosen by administrator to adapt to special applications). The Master is aware of the existing replicas because each server that joins the clusters initially sends the chunk that it is ready to serve.

[1]We recall that "Client" here technically means a component integrated to the Client application and implementing the communication protocol with the system.

Second, the Master is in charge of sending background heartbeat messages to each server. If a server does not answer to a heartbeat messages, the Master initiates a server replacement by asking to one of the (at least 2) remaining servers to copy to a new server the chunks that fell under their replication factor.

The Master itself must be particularly protected because it holds the file namespace. A recovery mechanism is used for all the updates that affect the namespace structure, similar to that presented in Figure 14.5, page 289. We refer the reader to the original paper (see last section) for technical details on the management of aspects that fall beyond the scope of our limited presentation, in particular access rights and data consistency.