Chunky: a distributed GFS-based file store

Matthew Lee
Stanford University

Abstract

We present Chunky, a distributed file store based on the design of the Google File System (GFS) [2]. Like GFS, Chunky replicates file chunks across multiple chunkservers for fault-tolerance and employs a single central master to coordinate access to files. Unlike GFS, Chunky implements write-once semantics for files, which simplifies the design considerably (obviating the need for a lease mechanism) while still enabling a practical, interesting system that demonstrates robustness in the face of multiple chunkserver failures.

1 Introduction

Designers of distributed file systems sometimes find it convenient to deviate from the traditional POSIX file API. Chunky is no exception. Motivated by the need to store more data than can fit on a single machine across multiple servers in a way that is resilient to the failure of some servers, Chunky exposes a simple file API and implements a GFS-like chunk replication scheme.

2 Chunky: Design and API

2.1 Write-Once: Not entirely useless

Write-once file semantics may seem too restrictive for a useful system, but on the contrary, there are in fact important classes of workloads that are essentially write-once in nature, such as content publishing. Video content alone, which by itself constitutes more than 60 percent of Internet traffic [1], is a case where video files are uploaded once - written to a distributed sytem of video content hosts - and thereafter only read by clients.

More generally, write-once semantics are not actually an impediment to useful computation: the entire paradigm of functional programming manages to make do with immutable "variables".

2.2 Chunky API

Chunky exposes a simple API for file manipulation.

open a function which returns a `ChunkyFile` that is a handle to data stored at `path`. At this point no space has been allocated for the data.

reserve a function that declares the size of a ChunkyFile in `bytes`, callable once per file.

write a function which writes `data` to a ChunkyFile segment given by `range`, callable once per file.

read a function which reads a `range` of data from a ChunkyFile and populates the `data` field provided by the caller.

2.3 Important system interactions

Heartbeats The chunkservers periodically send heartbeats to the master, enabling the master to detect when new chunkservers come online and when chunkservers fail. As in GFS, chunkservers are the source of truth for the chunk handles that they store; the master requests chunkservers that have just come online to send their list of chunk handles.

Chunk Handle Lookup The master maintains a mapping from (file name, intra-file chunk index) tuples to chunk handles. This mapping is populated at the time of file allocation. The client library never caches this information; every read by a client is begun by a query to the master for the chunk handle that stores the data of a file.

Chunk Server Lookup The master also maintains the mapping from chunk handle to a list of chunkservers that stores that chunk. This mapping is populated when chunkservers report their chunk handle lists. The client library queries the master for the chunk servers that own a particular chunk in order to read or write to the correct chunkservers.

Chunk Allocation When the client library reserves space for a file, the master computes the number of chunks needed and chooses k random chunkservers per chunk. The allocation of chunk data occurs on the chunkserver side by creating a file on the underlying Linux filesystem to store the chunk data.

Data Movement After clients have queried the master for the list of chunkservers that own a particular chunk, data movement between the client and chunkservers occurs without the mediation of the master. Like in GFS, this design ensures that the master is never on the datapath; the master only sends control messages, which helps prevent the master from bottlenecking the entire system.

2.4 Implementation trivia

The system is implemented in C++, using gRPC for communication between components. Chunkservers store file data on the underlying filesystem in order to take advantage of sparse file support (which avoids consuming physical space for allocated but unwritten chunks). File metadata on chunkservers is stored on a LevelDB database.

3 Evaluation

We instantiated a system with 5 chunkservers and a replication factor of 3. To observe the effect of a chunkserver failure on the system, we triggered a crash on chunkserver 3 at about the halfway point. Later, we revived chunkserver 3.

The figure on the following page shows the total throughput of the system in requests served per second. The failure of the chunkserver causes the total throughput to dip momentarily as the client is blocked while waiting for a request to the failed chunkserver to time out. After the master detects the failure of the chunkserver, the master deletes the failed chunkserver from its chunk handle to chunkserver mapping and therefore the client no longer attempts to send read requests to the failed chunkserver. As a result the blocking behaviour is not seen after the failure. The request load is redistributed to the other chunkservers and the total system throughput resumes. Later, when the failed chunkserver comes back online, the system returns to its former state, with all chunkservers sharing the load.

Availability

The source and driver scripts are available at `https://github.com/mattlkf/chunky`.

References

[1] *The Global Internet Phenomena Report*. Sandvine, 2019. `https://www.sandvine.com/global-internet-phenomena-report-2019`.

[2] Gobioff H. Ghemawat S. and Leung S.-T. The google file system. In *ACM Symposium on Operating Systems Principles*, 2003. `https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf`.

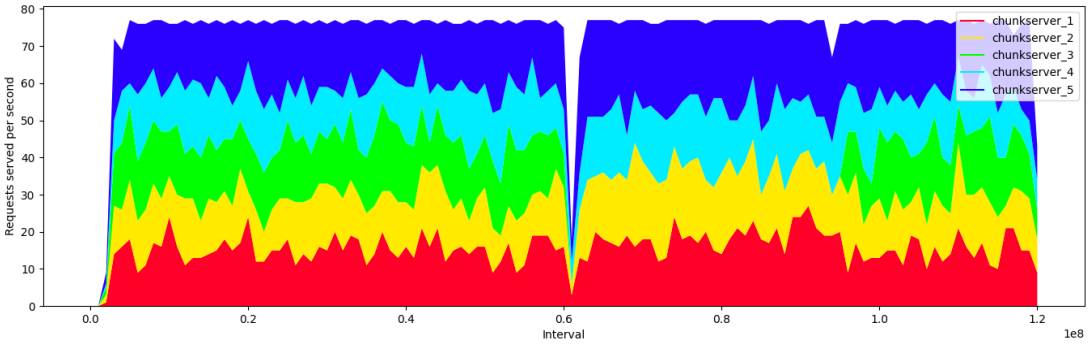

Figure 1: Request throughput in presence of node failure and recovery