Distributed File Systems

Ioan Raicu
Center for Ultra-scale Computing and Information Security
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 395 / EECS 495
Hot Topics in Distributed Systems: Data-Intensive Computing
January 28th, 2010

Distributed File Systems: State of the Art

- GFS: Google File System
  - Google
  - C/C++
- HDFS: Hadoop Distributed File System
  - Yahoo
  - Java, Open Source
- Sector: Distributed Storage System
  - University of Illinois at Chicago
  - C++, Open Source

Filesystems Overview

- System that permanently stores data
- Usually layered on top of a lower-level physical storage medium
- Divided into logical units called “files”
  - Addressable by a *filename* (“foo.txt”)
  - Usually supports hierarchical nesting (directories)
- A file *path* joins file & directory names into a **relative** or **absolute** address to identify a file (“/home/aaron/foo.txt”)

Shared/Parallel/Distributed Filesystems

- Support access to files on remote servers
- Must support concurrency
  - Make varying guarantees about locking, who "wins" with concurrent writes, etc...
  - Must gracefully handle dropped connections
- Can offer support for replication and local caching
- Different implementations sit in different places on complexity/feature scale

GFS: Motivation

- Google needed a good distributed file system
  - Redundant storage of massive amounts of data on cheap and unreliable computers

- Why not use an existing file system?
  - Google's problems are different from anyone else's
    - Different workload and design priorities
  - GFS is designed for Google apps and workloads
  - Google apps are designed for GFS

GFS: Assumptions

- High component failure rates
  - Inexpensive commodity components fail all the time
- "Modest" number of HUGE files
  - Just a few million
  - Each is 100MB or larger; multi-GB files typical
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads
- High sustained throughput favored over low latency

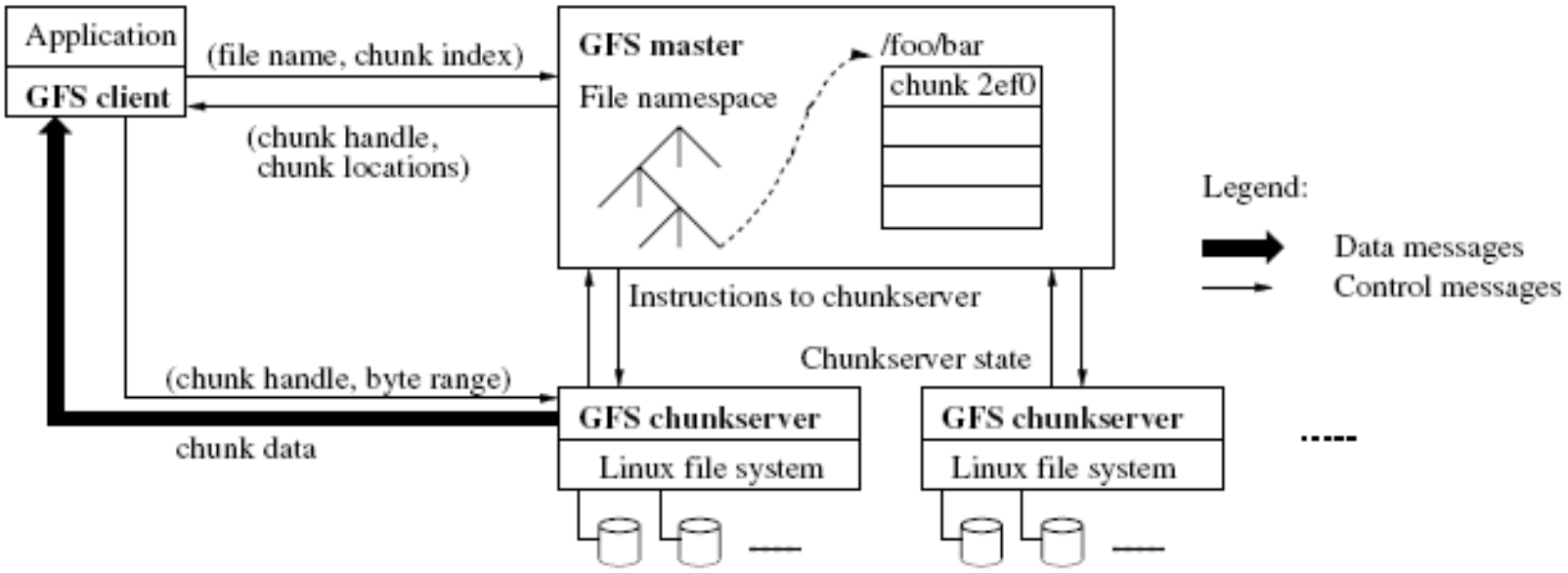
Google Workloads

- Most files are mutated by appending new data – large sequential writes
- Random writes are very uncommon
- Files are written once, then they are only read
- Reads are sequential
- Large streaming reads and small random reads
- High bandwidth is more important than low latency
- Google applications:
  - Data analysis programs that scan through data repositories
  - Data streaming applications
  - Archiving
  - Applications producing (intermediate) search results

GFS Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ *chunkservers*
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large data sets, streaming reads
- Familiar interface, but customize the API
  - Simplify the problem; focus on Google apps

GFS Architecture



GFS Architecture

- Single master
- Multiple chunk servers
- Multiple clients
- Each is a commodity Linux machine, a server is a user-level process
- Files are divided into chunks
- Each chunk has a handle (an ID assigned by the master)
- Each chunk is replicated (on three machines by default)
- Master stores metadata, manages chunks, does garbage collection, etc.
- Clients communicate with master for metadata operations, but with chunkservers for data operations
- No additional caching (besides the Linux in-memory buffer caching)

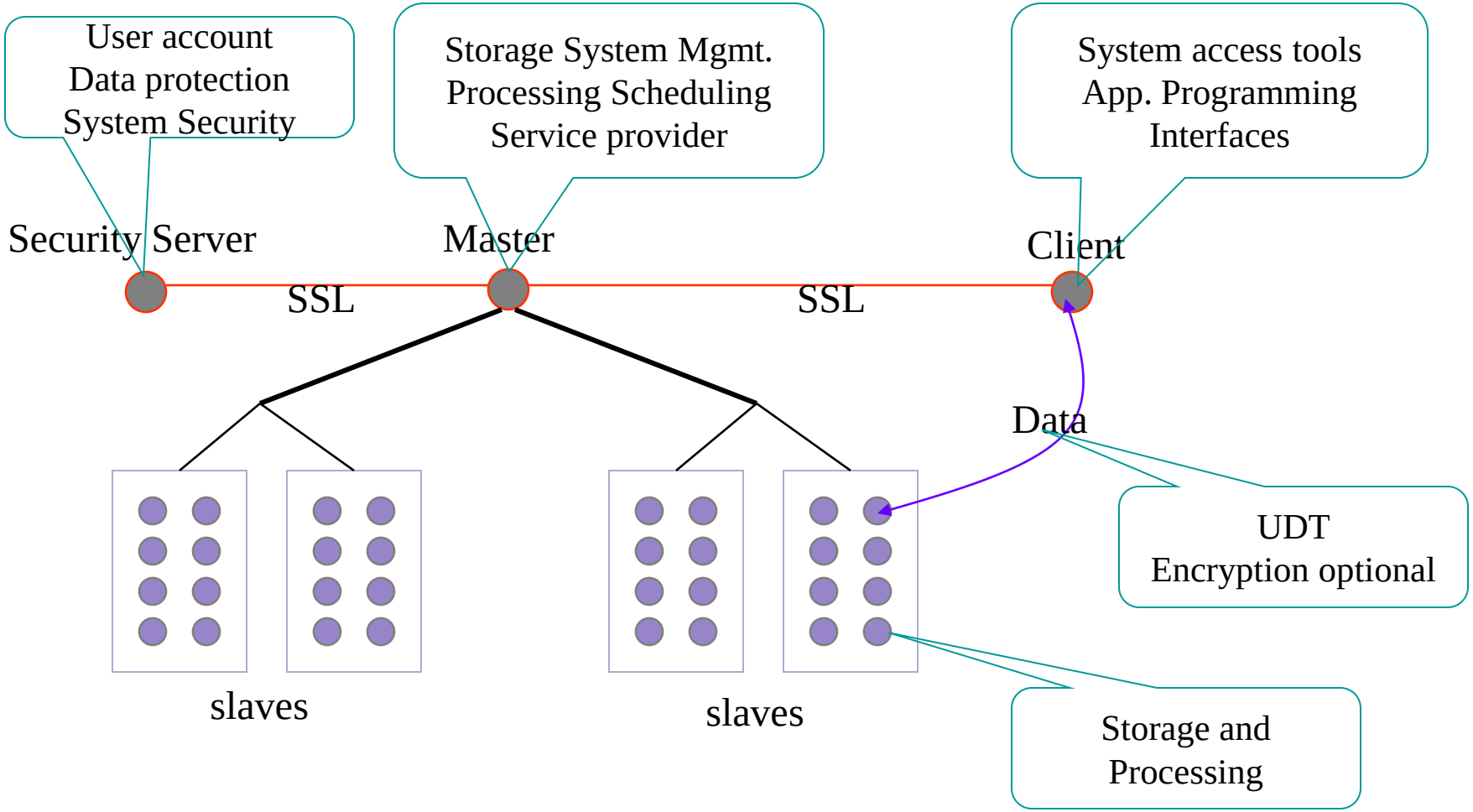
GFS Discussion

- Client/GFS Interaction
- Master
- Metadata
- Why keep metadata in memory?
- Why not keep chunk locations persistent?
- Operation log
- Data consistency
- Garbage collection
- Load balancing
- Fault tollerance

What is Sector/Sphere?

- Sector: Distributed Storage System
- Sphere: Run-time middleware that supports simplified distributed data processing.
- Open source software, GPL, written in C++.
- Started since 2006, current version 1.18
- http://sector.sf.net

Sector: Distributed Storage System



Sector: Distributed Storage System

- Sector stores files on the native/local file system of each slave node.
- Sector does not split files into blocks
  - Pro: simple/robust, suitable for wide area
  - Con: file size limit
- Sector uses replications for better reliability and availability
- The master node maintains the file system metadata. No permanent metadata is needed.
- Topology aware

Sector: Write/Read

- Write is exclusive
- Replicas are updated in a chained manner: the client updates one replica, and then this replica updates another, and so on. All replicas are updated upon the completion of a Write operation.
- Read: different replicas can serve different clients at the same time. Nearest replica to the client is chosen whenever possible.

Sector: Tools and API

- Supported file system operation: ls, stat, mv, cp, mkdir, rm, upload, download
  - Wild card characters supported
- System monitoring: sysinfo.
- C++ API: list, stat, move, copy, mkdir, remove, open, close, read, write, sysinfo.

Questions