Cloud Storage

Fog and Cloud Computing

Domenico Siracusa, Fondazione Bruno Kessler (FBK)

13/05/2022

Introduction

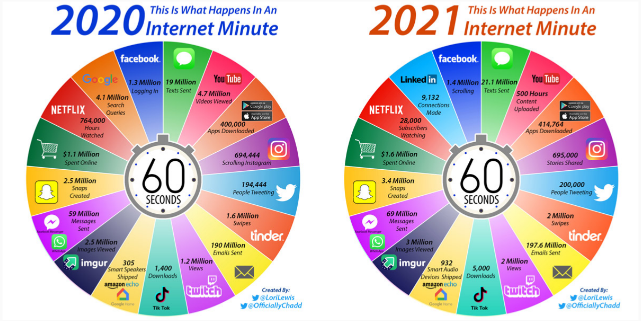
Preliminary information i

- Material
  - Dan C. Marinescu, Cloud Computing: Theory and Practice (Chapter 6)
  - Slides on Google Big Table are taken from presentation by Romain Jacotin (slideshare)
- Suggested readings
  - Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, *The Google file system*, In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)
  - Mike Burrows, *The Chubby lock service for loosely-coupled distributed systems*, In Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06)
  - Fay Chang et al., *Bigtable: A Distributed Storage System for Structured Data*, In Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06)

Preliminary information ii

- Suggested readings (2)
  - Jason Baker et al., *Megastore: Providing Scalable, Highly Available Storage for Interactive Services*, In Proceedings of the Conference on Innovative Data system Research (CIDR 11)
  - G. DeCandia, et al., *Dynamo: Amazon's Highly Available Key-value Store*, ACM Symposium on Operating Systems Principles (SOSP) 2007.
  - Documentation on OpenStack Swift, Cinder, and more. . . (curiosity!)

Data storage in the age of cloud computing



- Management of the large collection of storage systems poses significant challenges and requires novel approaches to system design
- Effective data replication and storage management strategies are critical to the computations performed on the cloud

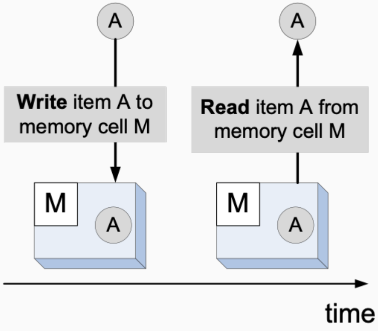
Major challenges

- Storage system design philosophy has shifted from performance-at-any-cost to reliability-at-the-lowest-possible-cost
- Data replication allows concurrent access to data from multiple processors and decreases the chances of data loss
- Maintaining consistency among multiple copies of data records increases the data management software complexity
    - Could negatively affect the storage system performance if data is frequently updated
- Sophisticated strategies to reduce access time and support multimedia access are necessary to satisfy timing requirements of data streaming and content delivery

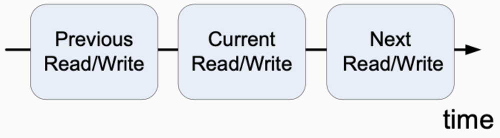
Storage and data models

- Storage model -> describes the layout of a data structure in a physical storage
  - a local disk, a removable media, or storage accessible via the network
- A data model -> captures the most important logical aspects of a data structure in a database

Properties of storage models


A
A
Write item A to memory cell M
Read item A from memory cell M
M
A
M
A
time

Read/Write coherence: the result of a **Read** of memory cell M should be the same as the most recent **Write** to that cell



Before-or-after atomicity: the result of every **Read** or **Write** is the same as if that **Read** or **Write** occurred either completely before or completely after any other **Read** or **Write**.

- Read/write coherence and before-or-after atomicity are two highly desirable properties of any storage model and in particular of cell storage

Types of storage (1): block storage

- Block storage manages data as blocks within sectors and tracks
  - Storage and has access to raw and formatted HW
    - Useful when speed and efficiency are most important
    - Examples of block storage: Cinder, Amazon EBS (Elastic Block Storage)
- Commonly used for data bases
  - DBMSs (Data Base Management Systems) use block storage to read and write structured data
    - Ideal for storing relational customer information

Types of storage (2): file storage

- File storage manages data in structured files
  - Exposes a file system that controls how data is stored and retrieved
- File systems are used on local data storage devices or provide file access via a network protocol
  - Examples of distributed file systems: NFS, Google File System
- Useful abstraction
  - Does not work very well with large amounts of data or high-demands for a particular piece of data

Types of storage (3): object storage

- Object storage manages data as objects
  - Each object typically includes the data itself, a variable amount of metadata, and a globally unique identifier
  - Allow access to whole objects or "blobs" of data via API that are system specific
  - Good to store contents that can grow without bounds (backups, archives)
    - Not suitable for relational databases or data requiring random access/updates within objects
  - Example of distributed object-storage for cloud: OpenStack Swift

Data Base Management System (DBMS) i

- Database -> a collection of logically-related records
- Data Base Management System (DBMS) -> software that controls the access to the database
  - Most cloud apps interact with the file systems through DBMS
- In cloud, data stores are usually distributed creating distributed databases where users store information on a number of nodes
  - Example of cloud distributed databases: Google Bigtable, Amazon's Dynamo
- Query language -> a dedicated programming language used to develop database applications

Block storage

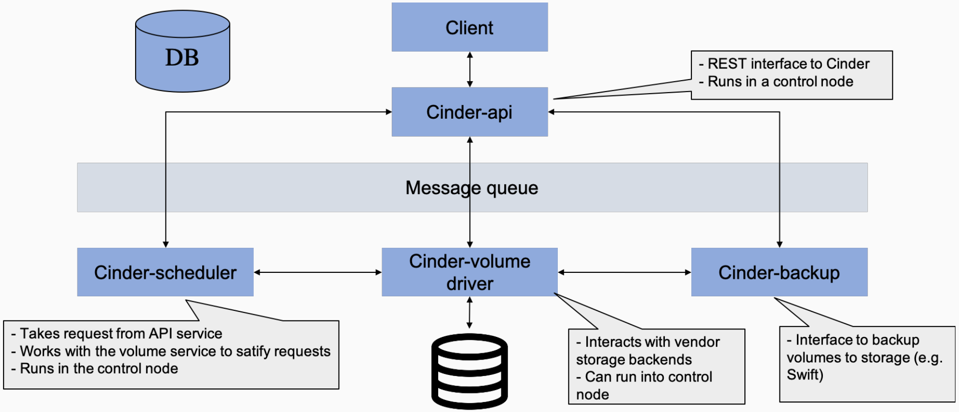
OpenStack Cinder

- Implements services and libraries to provide on-demand, self-service access to block storage resources
  - Software Defined block storage via automation on top of various traditional backend block storage devices
- Manages block storage
  - Different than shared file storage
  - Different than object storage (Swift)
- Documentation: `https://docs.openstack.org/cinder/latest/`

Where does it fit

- Provides API to interact with vendors' storage backends
- Exposes vendor's storage hardware to the cloud
- Provides persistent storage to VMs, containers, bare metal. . .
- Enables end users to manage their storage without knowing where that storage is coming from

Cinder architecture


DB
Client
- REST interface to Cinder
- Runs in a control node
Cinder-api
Message queue
Cinder-scheduler
Cinder-volume driver
Cinder-backup
- Takes request from API service
- Works with the volume service to satify requests
- Runs in the control node
- Interacts with vendor storage backends
- Can run into control node
- Interface to backup volumes to storage (e.g. Swift)

What Cinder supports

- Volume create/delete
- Volume attach/detach
- Snapshot create/delete
- Create volume from snapshot
- Copy image to volume
- Copy volume to image
- Clone volume
- Extend volume

Distributed file systems

Logical and physical organization of a file

- File -> a linear array of cells stored on a persistent storage device
  - Viewed by an application as a collection of logical records
  - Stored on a physical device as a set of physical records, or blocks, of size dictated by the physical media
- File pointer -> a cell used as a starting point for a read or write operation
- Organization of a file
  - Logical -> reflects the data model, the view of the data from the perspective of the application
  - Physical -> reflects the storage model and describes the manner the file is stored on a given storage media

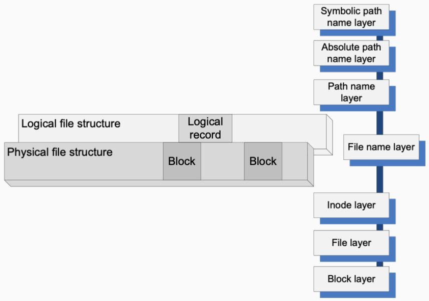
File systems i

- File system -> collection of directories, each directory provides information about a set of files (file system controls how data is stored and retrieved)
    - *Traditional* – Unix File System
    - *Distributed file systems* – Network File System, Storage Area Networksm Parallel File Systems

1. Unix File System (UFS)

- *Layered design* provides flexibility
    - Layered design allows UFS to separate the concerns for the physical file structure from the logical one
    - *inode* layer allows UFS to treat uniformly local and remote file access
- *Hierarchical design* supports scalability reflected by the file naming convention
    - Allows grouping of files directories, supports multiple levels of directories and collections of directories and files (the so-called file systems)
- *Metadata* supports a systematic design philosophy of the file system and device-independence
    - Metadata includes: file owner, access rights, creation time, time of the last modification, file size, the structure of the file and the persistent storage device cells where data is stored
    - *inodes* contain information about individual files and directories
        - inodes are kept on persistent media together with the data

UFS layering

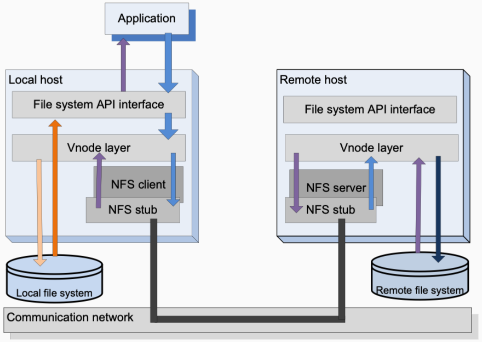


- Lower layers: physical organization
  - Block layer: to locate individual blocks on the physical device
  - File layer: reflects organization of blocks into files
  - Inode layer: provides metadata for the objects (files and directories)
- Upper layers: logical organization
- File name layer mediates between machine- and user-oriented views of the FS

2. Network File System (NFS)

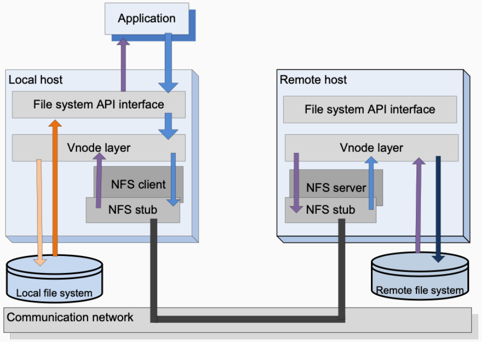
- Design objectives:
  - Provide the same semantics as a local Unix File System (UFS) to ensure compatibility with existing applications
  - Facilitate easy integration into existing UFS
  - Ensure that the system will be widely used -> support clients running on different operating systems
  - Accept a modest performance degradation due to remote access (network may have limited bandwidth)
- NFS is based on the client-server paradigm
  - Client runs on the local host while the server is at the site of the remote file system
  - They interact by means of Remote Procedure Calls (RPC)
- A remote file is uniquely identified by a file handle (*fh*) rather than a file descriptor
  - fh is a 32-byte internal name
    - combination of the file system identification, an *inode* number, and a generation number

NFS client-server interaction i



- *vnode* layer (corrensponding of inode) implements file operation in a uniform manner, regardless of whether the file is local or remote
- An operation targeting a local file is directed to the local file system while one for a remote file involves NFS

NFS client-server interaction ii

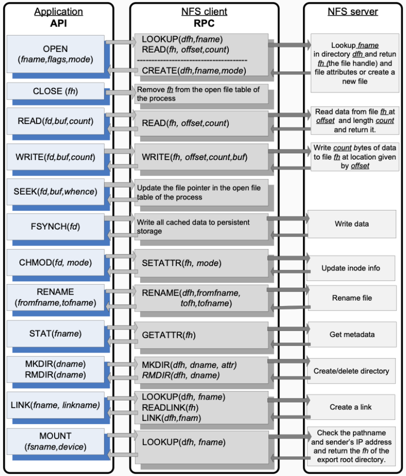


- Steps
    - NSF client packages the relevant information about the target
    - NFS server passes it to the *vnode* layer on the remote host
    - Remote *vnode layer* directs it to the remote file system

APIs i

- The API of the UNIX file system and the corresponding RPC issued by an NFS client to the NFS server
    - fd -> file descriptor
    - fh -> for file handle
    - fname -> file name
    - dname -> directory name
    - dfh -> the directory were the file handle can be found
    - count -> the number of bytes to be transferred
    - buf -> the buffer to transfer the data to/from
    - device -> the device where the file system is located

APIs ii



Application API	NFS client RPC	NFS server
OPEN (*fname,flags,mode*)	LOOKUP(*dfh,fname*) READ(*fh, offset,count*) ---------------- CREATE(*dfh,fname,mode*)	Lookup *fname* in directory *dfh* and retun *fh* (the file handle) and file attributes or create a new file
CLOSE (*fh*)	Remove *fh* from the open file table of the process	
READ(*fd,buf,count*)	READ(*fh, offset,count*)	Read data from file *fh* at *offset* and length *count* and return it.
WRITE(*fd,buf,count*)	WRITE(*fh, offset,count,buf*)	Write *count* bytes of data to file *fh* at location given by *offset*
SEEK(*fd,buf,whence*)	Update the file pointer in the open file table of the process	
FSYNCH(*fd*)	Write all cached data to persistent storage	Write data
CHMOD(*fd, mode*)	SETATTR(*fh, mode*)	Update inode info
RENAME (*fromfname,tofname*)	RENAME(*dfh,fromfname, tofh,tofname*)	Rename file
STAT(*fname*)	GETATTR(*fh*)	Get metadata
MKDIR(*dname*) RMDIR(*dname*)	MKDIR(*dfh, dname, attr*) *RMDIR(dfh, dname)*	Create/delete directory
LINK(*fname, linkname*)	LOOKUP(*dfh, fname*) READLINK(*fh*) LINK(*dfh,fnam*)	Create a link
MOUNT (*fsname,device*)	LOOKUP(*dfh, fname*)	Check the pathname and sender's IP address and return the *fh* of the export root directory.

- API of UFS and corrensponding RPCs issued by NFS client to NFS server

3. Other distributed file systems: precursors

- Andrew File System (AFS)
  - Developed by Carnegie Mellon University (CMU) and IBM
  - Envisioned very large number of workstations interconnected with relatively small number of servers
    - Each individual at CMU to have an Andrew workstation, system would connect up to 10k workstations
  - Emphasis on performance (caching), security (encrypted communication), simple management (small number of servers)
- Sprite Network File System (SFS)
  - Focus on caching and implementation of Least Recently Used (LRU) pages

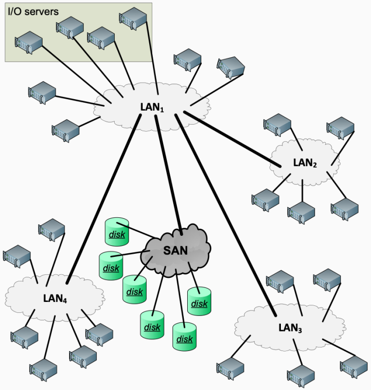
Design choices for distributed file systems

- Common policy: once file is closed, server will have the newest version on persistent storage
- Policy to write a block
    - *Delay in write-backs*: a block is first written to cache and writing on the disk is delayed for a time in the order of tens of seconds
        - Speeds-up writing and avoids writing when data is discarded before the time to write it to the disk
        - However, data can be lost in case of system failure
    - Alternative is *write-through*: block is written to the disk as soon as it is available on the cache (increased reliability, increased time for a write)
- Concurrency
    - *Sequential write-sharing*: a file cannot be opened simultaneously for reading and writing by several clients
    - *Concurrent write-sharing*: multiple clients can modify the file at the same time

4. General Parallel File System (GPFS) i

- Parallel I/O implies concurrent execution of multiple input/output operations
  - Support for parallel I/O is essential for the performance of many applications
- Concurrency control is a critical issue for parallel file systems
  - Several semantics for handling the shared access are possible, e.g.
    - When clients share the file pointer successive reads issued by multiple clients advance the file pointer
    - Allow each client to have its own file pointer
- GPFS
  - Developed at IBM in the early 2000s as a successor of the TigerShark multimedia file system
  - Designed for optimal performance of large clusters
    - it can support a file system of up to 4 PB consisting of up to 4,096 disks of 1 TB each
  - Maximum file size is (2^{63} -1) bytes
  - A file consists of blocks of equal size, ranging from 16 KB to 1 MB, stripped across several disks

4. General Parallel File System (GPFS) ii



- GPFS configuration:
  - Disks are interconnected via SAN
  - Compute servers are distributed in LANs

GPFS reliability

- To recover from system failures, GPFS records all metadata updates in a write-ahead log file
  - Write-ahead -> updates are written to persistent storage only after the log records have been written
- The log files are maintained by each I/O node for each file system it mounts
  - Any I/O node can initiate recovery on behalf of a failed node
- Data striping (segmenting logically sequential data, e.g. file, so that consecutive segments are stored on different physical storage devices) allows concurrent access and improves performance but can have unpleasant side-effects
  - When a single disk fails, a large number of files are affected
  - To further improve the fault tolerance of the system, GPFS data files as well as metadata are replicated on two different physical disks

5. Google File System (GFS)

- GFS -> developed in the late 1990s
    - Uses thousands of storage systems built from inexpensive commodity components to provide petabytes of storage to a large user community with diverse needs
- Design considerations
    - Scalability and reliability are critical features of the system
        - They must be considered from the beginning, rather than at some stage of the design
    - Vast majority of files range in size from a few GB to hundreds of TB
    - Most common operation is to append to an existing file
        - Random write operations to a file are extremely infrequent
    - Sequential read operations are the norm
    - Users process the data in bulk and are less concerned with the response time
    - Consistency model should be relaxed to simplify the system implementation but without placing an additional burden on the application developers
- *CloudStore*: open source C++ implementation of GFS

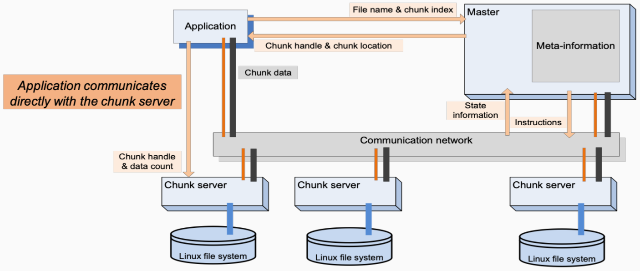
GFS: design decisions

- Segment a file in large chunks
- Implement an atomic file append operation allowing multiple applications operating concurrently to append to the same file
- Build cluster around high-bandwidth rather than low-latency interconnection net
  - Separate the flow of control from the data flow
  - Schedule high-bw data flows by pipelining data transfer over TCP connections
  - Exploit network topology by sending data to the closest node in the network
- Eliminate caching at the client site
  - Caching increases the overhead for maintaining consistency among cashed copies
- Ensure consistency by channeling critical file operations through a master, a component of the cluster which controls the entire system
  - Minimize the involvement of the master in file access operations to avoid hot-spot contention and to ensure scalability
- Support efficient checkpointing and fast recovery mechanisms
- Support an efficient garbage collection mechanism

GFS chunks

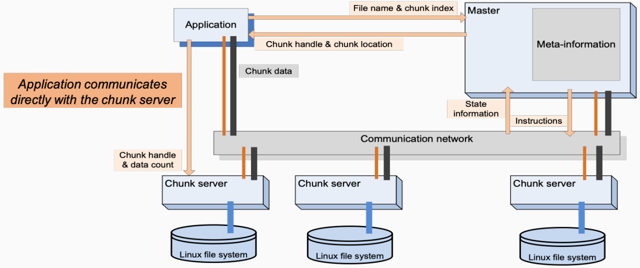
- GFS files are collections of fixed-size segments called chunks
  - Chunk size is 64 MB
    - Motivated by the desire to optimize the performance for large files and to reduce the amount of metadata maintained by the system
    - Large chunk size increases the likelihood that multiple operations will be directed to the same chunk thus, it reduces the number of requests to locate the chunk
    - Also allows the application to maintain a persistent network connection with the server where the chunk is located
  - A chunk consists of 64 KB blocks and each block has a 32 bit checksum
  - Chunks are stored on Linux files systems and are replicated on multiple sites
    - A user may change the number of the replicas, from the standard value of three, to any desired value
  - At the time of file creation each chunk is assigned a unique chunk handle

GFS cluster architecture



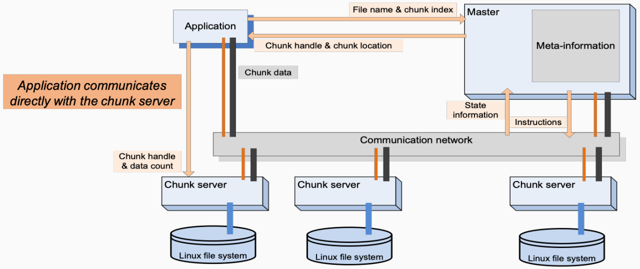
- Master maintains state information about all system components
  - It controls a number of chunk servers
- A chunk server runs under Linux
  - Uses metadata provided by master to communicate directly with the app
- Data and the control paths are shown separately
  - Data paths with thick lines and the control paths with thin lines
- Arrows show control flows between app, master and chunk servers

Steps of a write request i



1. Client contacts the master which assigns a lease to one of the chunk servers (no lease for that chunk exists)
   - Master replies with the ID of the primary and secondary chunk servers holding replicas of the chunk
2. Client sends data to all chunk servers holding replicas
   - Chunk servers stores data in internal LRU (Least Recently Used) buffer and send ack to client

Steps of a write request ii



3. Client send write request to primary chunk server once it got acks from all chunk servers holding replicas
4. Primary chunk server sends *write* to all secondaries
5. Each secondary applies mutations in the order of the sequence number and sends ack back to primary
6. After receiving ack from all secondaries, primary acks client

Locks and consensus

Locks and consensus

- Operating systems use lock managers to organise and serialise access to resources
    - Locks enable controlled access to shared storage and ensure atomicity of read and write operations
    - Sometimes, they provide reliable storage for loosely-coupled distributed systems
- To elect a leader or a reliable master to take decisions, consensus must be sometimes reached
    - E.g. in GFS master maintains state information about all system components
    - Can you name is the most famous consensus protocol?

Types of Locks

- Effect
  - Advisory locks -> based on the assumption that all processes play by the rules
    - Do not have any effect on processes that circumvent the locking mechanisms and access the shared objects directly
  - Mandatory locks -> block access to the locked objects to all processes that do not hold the locks, regardless if they use locking primitives or not
- Time
  - Fine-grained locks -> locks that can be held for only a very short time
    - Allow more application threads to access shared data in any time interval, but generate a larger workload for the lock server
    - When the lock server fails for a period of time, a larger number of applications are affected
  - Coarse-grained locks -> locks held for a longer time

Systematic approach to locking

- Two approaches possible:
  - Delegate to the clients the implementation of the consensus algorithm and provide a library of functions needed for this task
    - Depend on NO other servers
    - Cumbersome and prone to errors
- Create a locking service which implements a version of the asynchronous Paxos algorithm and provide a library to be linked with an application client to support service calls
  - Easier to maintain existing program structure and communication patterns
  - Lock service use several replicas to achieve high availability (=quorums), but even a single client can obtain lock and make progress safely -> reduces number of servers needed for a realiable client system to make progress
  - Scalability could be an issue

1. Chubby

- Chubby lock service developed by Google for use within a loosely-coupled distributed system
  - Large number of machines (e.g. 10k) connected by high-speed network
  - Provides coarse-grained locking
  - And reliable (low-volume) storage
- Chubby provides an interface much like a distributed file system
  - Whole file read and writes operation (no seek)
  - Advisory locks
  - Notification of various events such as file modification
- Design emphasis
  - Availability
  - Reliability
  - Not high performance or high throughput
- Uses asynchronous consensus: PAXOS with lease timers to ensure liveness

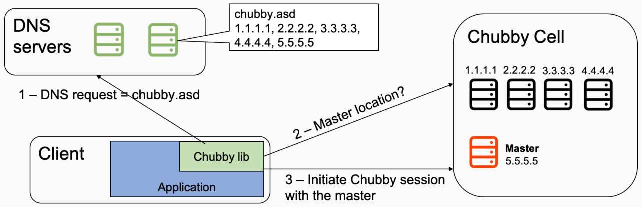
Chubby design choices

- Two key design decision
  - Google chooses lock service (but also provide PAXOS client library independently from Chubby for specific projects)
  - Serve small files to permit elected primaries (= client application masters) to advertise themselves and their parameters
- Decision based on Google's expected usage and environment
  - Allow thousands of clients to observe Chubby files -> event notification mechanism to avoid massive polling
  - Consistent cache semantics (protects lock service from polling)
  - Security mechanism (access control)
  - Coarse-grained locks (long duration lock -> low lock-acquisition rate -> less load on lock server)

Chubby design i

- System structure
  - Two main components that communicate via RPC
    - A replica server
    - A library linked against client applications
- Chubby cell consists of a small set of servers (typically 5) known as replicas
  - Replicas use a distributed consensus protocol (PAXOS) to elect a master and replicate logs
  - Read and write requests are satisfied by the master alone
  - If a master fails, other replicas run the election protocol when their master lease expire (new master elected in few seconds)
- Clients find the master by sending master location reqs to the replicas listed in the DNS
  - Non-master replicas respond by returning identity of the master
  - Once a client has located a master, it redirects all requests to it either until it ceases to respond or until it indicates it is no longer the master

Chubby design i



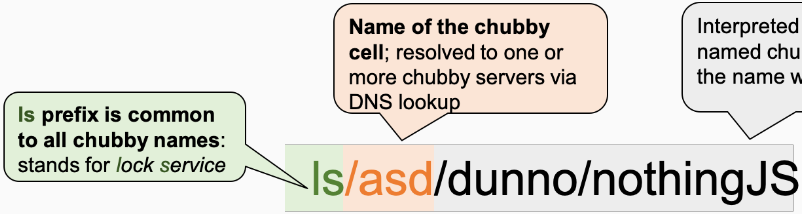
- Chubby cell consisting of 5 replicas
  - One of them elected as a master
  - All maintain copies of a simple database
- Clients use RPCs to communicate with the master via a chubby library
  - Send read/write only to the master
- Master
  - Propagates write to replica and replies after the write reaches a majority
  - Replies directly to reads, as it has most up to date state

So, Locks?

- If applications wants to get the lock over a shared resource (e.g. and operate that resource)
    - Resources can be registered in the chubby file system as nodes (files, see later)
    - Potential clients try to create a lock on chubby
    - The first one that gets the lock becomes the one that can use that resource

Chubby design: files and directories

- Chubby exports a file system interface simpler than Unix
  - Tree of files and directories with name components separated by /
  - Each directory contains a list of child files and directories (collectively called nodes)
  - Each file contains a sequence of un-interpreted bytes
  - No symbolic or hard links, no file move
  - No directory modified times, no last-access times
  - No path-dependent permission semantics: file is controlled by the permission on the file itself


ls prefix is common to all chubby names: stands for *lock service*
Name of the chubby cell; resolved to one or more chubby servers via DNS lookup
Interpreted within the named chubby cell (it is the name within the cell)
ls/asd/dunno/nothingJS

Chubby nodes

- Node: a file or directory
    - Any node can act as advisory reader/writer lock
- Node can be either permanent of ephemeral
    - Ephemeral used as temporary files, e.g., indicate a client is alive
- Metadata
    - Three names of ACLs used to control R/W/change ACL names
    - 64-bit file content checksum
- Handles
    - Clients open nodes to obtain Handles (like UNIX file descriptors)
        - Handlers include check digits, sequence number and more info

Chubby design: locks, sequencers and delay

- Each Chubby file and directory (any node) can act as reader/writer lock (advisory ones)
- Acquiring a lock in either mode requires write permission
    - Exclusive mode (writer): one client may hold the lock
    - Shared mode (reader): any number of client handles may hold the lock
- In case of contention, two solutions
    - Sequencer: string holding information about state of the lock
    - Lock-delay: time to wait before claiming the lock if the lock becomes free

Chubby design: API

- Open/close node name
- Read/write full contents
- Set ACL
- Delete node
- Acquire or release lock
- Set, get or check sequencer

Chubby: in short

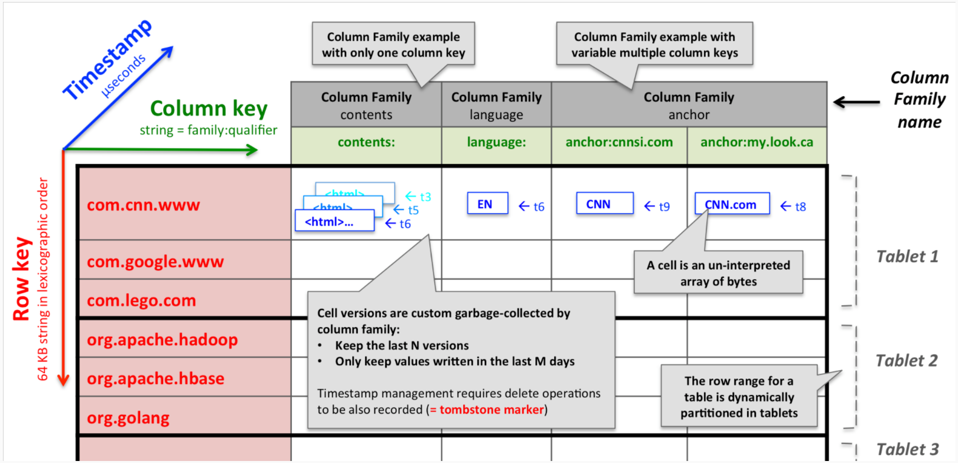
- Chubby is a distributed lock service for coarse-grained synchronization of distributed systems
  - Distributed consensus among few replicas for fault-tolerance
  - Consistent client-side caching
  - Timely notification of updates
  - Familiar file system interface
- Usage
  - Elect a primary from redundant replicas (GFS and Bigtable)
  - Standard repository for files that require high-availability (ACLs)
  - Well-known and available location to store a small amount of meta-data
  - Name service (became primary Google internal name service)
- Bigtable usage
  - To elect a master
  - To allow the master to discover the servers its controls
  - To permit clients to find the master

Distributed databases

1. Google Big Table

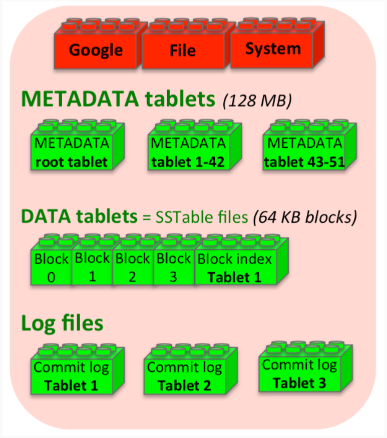
- Bigtable is a distributed storage system for managing structured data
    - Designed to scale to a very large size: petabytes of data across thousands of commodity servers
    - Not a relational database, it is a sparse, distributed, persistent multi-dimensional sorted map (key/value store)
- Many projects at Google store data in BigTable
    - Web indexing, Google earth and Google finance, . . .
    - Different data sizes: URL, web pages, satellite imagery, . . .
    - Different latency requirements: backend bulk processing to real-time data serving
- Bigtable is a flexible, high performance solution

Data model


Timestamp
μseconds
Column key
string = family:qualifier
Row key
64 KB string in lexicographic order
Column Family example with only one column key
Column Family example with variable multiple column keys
Column Family contents
Column Family language
Column Family anchor
Column Family name
contents:
language:
anchor:cnnsi.com
anchor:my.look.ca
com.cnn.www
com.google.www
com.lego.com
org.apache.hadoop
org.apache.hbase
org.golang
<html>
<html>
<html>...
t3
t5
t6
EN
t6
CNN
t9
CNN.com
t8
A cell is an un-interpreted array of bytes
Cell versions are custom garbage-collected by column family:
• Keep the last N versions
• Only keep values written in the last M days
Timestamp management requires delete operations to be also recorded (= tombstone marker)
The row range for a table is dynamically partitioned in tablets
Tablet 1
Tablet 2
Tablet 3

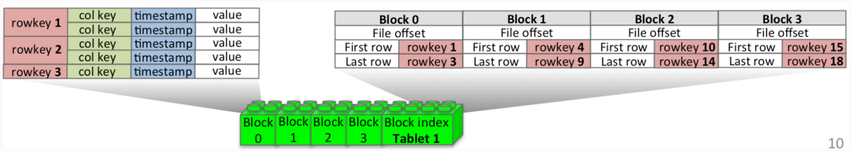
GBT Building blocks i

- Google File System
  - Bigtable uses the fault-tolerant and scalable distributed GFS file system to store
    - Metadata: METADATA tablets (store tablets location)
    - Data: SSTables collection by tablet
    - Log: tablet logs
- Google SSTable (Sorted String Table) file format
  - Used to store table data in GFS
  - Persistent, ordered immutable map from keys to values



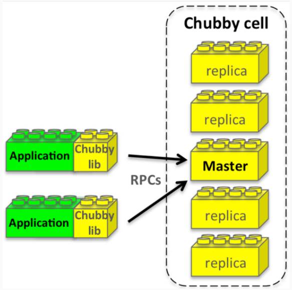
GBT Building blocks ii

- Google SSTable (Sorted String Table) file format
  - Contains a sequence of 64 KB blocks (size configurable)
    - Optionally, Blocks can be completely mapped into memory (= lookups and scans without touching disk)
  - Block index stored at the end of the file
    - Used to locate blocks
    - Index loaded in memory when the SSTable is opened
    - Lookup with a single seek
    - Find the appropriate block by performing a binary search in the in-memory index
    - Reading the appropriate block from disk



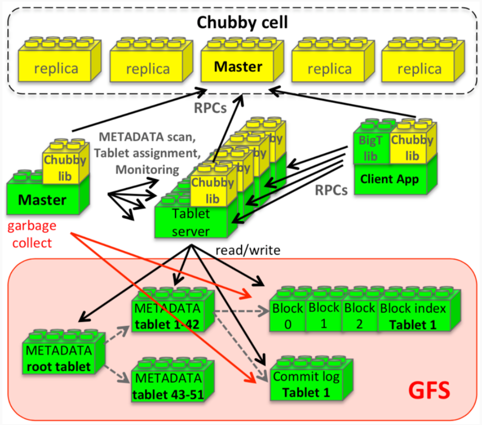
GBT Building blocks iii

- Google Chubby: Chubby server with 5 active replicas and one master to serve requests
- Chubby provides a reliable namespace that contains directories and small files (<256 KB)
- Bigtable uses Chubby for a variety of tasks
  - Ensure there is at most one active master at any time
  - Store the bootstrap location of Bigtable data (root tablet)
  - Discover tablet servers and finalize tablet server deaths
  - Store Bigtable schema information (column family information for each table)
  - Store ACLs
  - If Chubby is unavailable, then Bigtable is unavailable



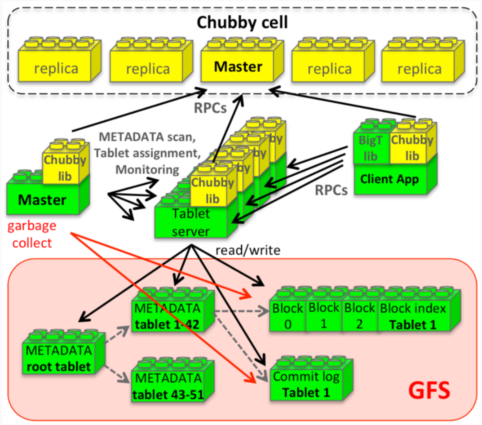
GBT implementation: major components i

- One Master server
  - Assigning tablets to tablet servers
  - Detecting the addition and expiration of tablet servers
  - Balancing tablet server load
  - Garbage collecting of files in GFS
  - Handling schema changes (table creation, column family creation/deletion)



GBT implementation: major components ii

- Many Tablet servers, each
  - Manages a set of tablets
  - Handles read and write request to the tablets
  - Splits tablets that have grown too large (100-200 MB)
- Clients
  - Do not rely on the master for tablet location and information
  - Communicates directly with tablet servers for reads and writes

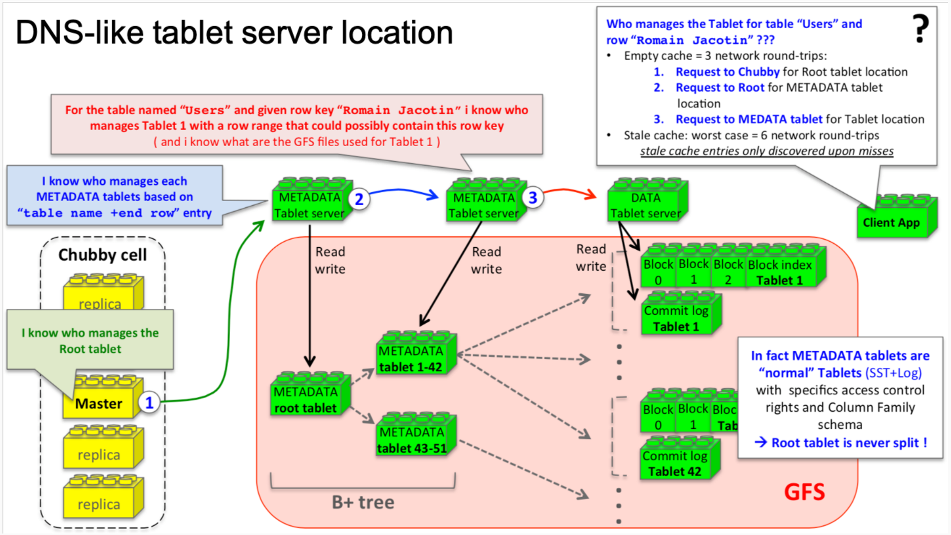


GBT implementation: API

- Tablet server (add or remove)
- Table (create or delete)
- Column family (create or delete)
- Tablet flags (access control rights and metadata)
- Column family flag (access control rights and metadata)
- Cell value: Put (rowkey, columnkey, value), Get/Delete (rowkey, columnkey)
- Lookup value for row -> Has(rowkey, columnfamily)
- Lookup value from table -> Scan (rowfilter, columnfilter, timestampfilter)

Single-row transactions (atomic read-modify-write sequence)

GBT implementation: tablet location


DNS-like tablet server location
For the table named "Users" and given row key "Romain Jacotin" i know who manages Tablet 1 with a row range that could possibly contain this row key (and i know what are the GFS files used for Tablet 1)
I know who manages each METADATA tablets based on "table name +end row" entry
Chubby cell
replica
I know who manages the Root tablet
Master
1
replica
replica
METADATA Tablet server
2
METADATA Tablet server
3
DATA Tablet server
Client App
Read write
Read write
Read write
Block 0
Block 1
Block 2
Block index Tablet 1
Commit log Tablet 1
METADATA tablet 1-42
METADATA root tablet
METADATA tablet 43-51
Block 0
Block 1
Bloc
Tab
Commit log Tablet 42
B+ tree
GFS

Who manages the Tablet for table "Users" and row "Romain Jacotin" ??? ?

- Empty cache = 3 network round-trips:
  1. Request to Chubby for Root tablet location
  2. Request to Root for METADATA tablet location
  3. Request to MEDATA tablet for Tablet location
- Stale cache: worst case = 6 network round-trips

  stale cache entries only discovered upon misses

In fact METADATA tablets are "normal" Tablets (SST+Log) with specifics access control rights and Column Family schema
→ Root tablet is never split !

GBT implementation: tablet assignment

- Each tablet is assigned to one tablet server at time
- Master keeps tracks of
    - Set of live tablet servers (tracking via Chubby)
    - Current assignment of tablet to tablet servers
    - Currrent unassigned tablets
- When a tablet is unassigned, master assigns the tablet to an available tablet server by sending a tablet load request

GBT implementation: tablet server discovery

- When a tablet server starts, it creates and acquires an exclusive lock on a uniquely-named file in a specific Chubby directory (*“servers” directory*)
    - Master monitors this directory to discover tablet servers
- A tablet server stops serving its tablets if it loses its exclusive Chubby lock
    - If the Chubby file no longer exists, then the tablet server will never be able to serve again, so (sadly) it kills itself

GBT implementation: tablet server monitoring

- Master is responsible for detecting when a tablet server is no longer serving its tablets, and for reassigning those tablets
- Master periodically asks each tablet server for the status of its lock to detect when a tablet server is no longer serving its tablets
- If a tablet server reports that it has lost its lock, or if the master was unable to reach a server during its last attempts, the master attempts to acquire acquire the lock for the Chubby file
    - If the master is able to acquire the lock, then Chubby is live and the tablet server is dead or isolated, the master deletes its server file to ensure that the tablet server can never serve again
    - Then master can assign all the tablets that were previously assigned to this tablet server into the set of unassigned tablets

GBT implementation: master isolated

- To ensure that a Bigtable cluster is not vulnerable to networking issues between the master and Chubby, the master kills itself if its Chubby session expires (master failures do not change the assignment of tablets to tablet servers)

GBT implementation: master startup

- When a master is started by the cluster management system, it needs to discover the current tablet assignments before it can changes them:

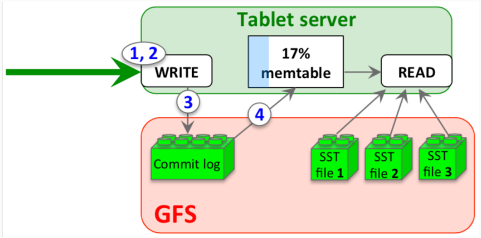
    1. Master grabs a unique master lock in Chubby to prevent concurrent master instantiations
    2. Master scans the servers directory in Chubby to find the live tablet servers
    3. Master communicate with every live tablet servers to discover what tablets are already assigned to each server
    4. Master adds the root tablet to the set of unassigned tablets if an assignment for the root tablet is not discovered in step 3
    5. Master scans the METADATA table to learn the set of tablets (and detect unassigned tablets)

GBT implementation: tablet merging/splitting

- The set of existing tablets only changes when a tablet is created or deleted
    - Two existing tablets are merged to form one larger tablet
    - Existing tablet is split into two smaller
- Master initiates Tablets merging
- Tablet server initiate tablet splitting
    - Commit the split by recording information for new tablet in the METADATA tablet
    - After committed, the tablet server notifies the master

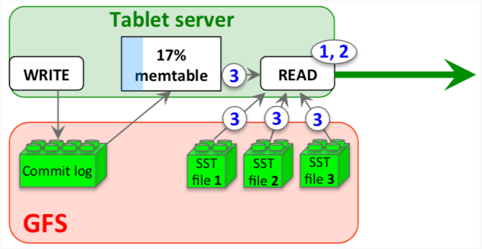
GBT implementation: tablet serving i

- Write operation
    1. Server checks request is well-formed
    2. Server checks sender is authorized to write (list of permitted writers in a Chubby file)
    3. Valid mutation is written to commit log that stores redo records (group commit to improve throughput)
    4. After mutation is committed, its contents are inserted into memtable (= in memory sorted buffer)



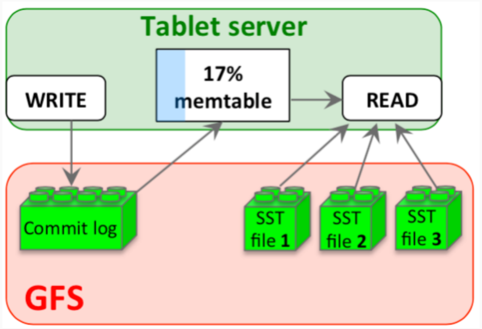
GBT implementation: tablet serving ii

- Read operation
    1. Server checks that the request it is well-formed
    2. Server checks that the sender is authorized to read (list of permitted readers from a Chubby file)
    3. Valid read operation is executed on a merged view of the sequence of SSTables and the memtable



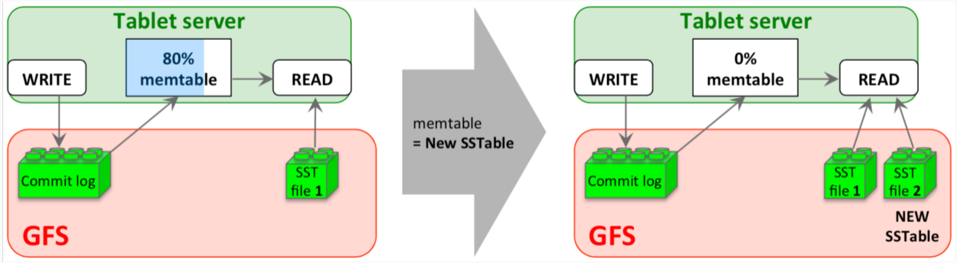
GBT implementation: tablet serving iii

- Tablet Recovery
    1. Tablet server reads its metadata from the METADATA table (lists of SSTables that comprise a tablet and a set of a redo points, which are pointers into any commit logs that may contain data for the tablet)
    2. The tablet server reads the indices of the SSTables into memory and reconstructs the memtable by applying all of the updates that have a committed since the redo points



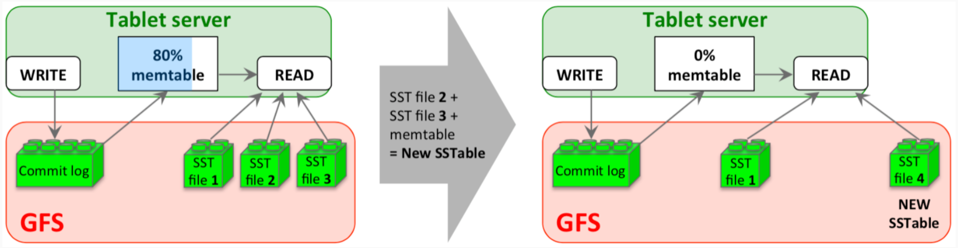
GBT implementation: compactions i

- Minor compaction
    - When memtable size reaches a threshold, memtable is frozen, a new memtable is created, and the frozen memtable is converted to a new SSTable and written to GFS
    - Two goals: shrinks the memory usage of the tablet server, reduces the amount of data that has to be read from the commit log during a recovery



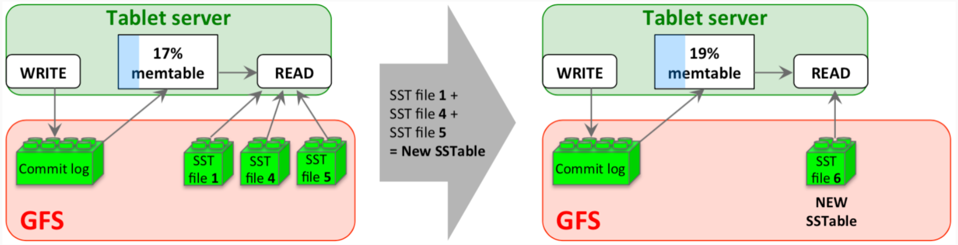
GBT implementation: compactions ii

- Merging compaction
  - Problem: every minor compaction creates a new SSTable (-> arbitrary number of SSTables!)
  - Solution: periodic merging of a few SSTables and the memtable



GBT implementation: compactions iii

- Major compaction
  - It is a merging compaction that rewrites all SSTables into exactly one SSTable that contains no deletion information or deleted data
  - Bigtable cycles throught all of it tablets and regularly applies major compaction to them (=reclaim resources used by deleted data in a timely fashion)



Distributed object storage

OpenStack Swift

- Distributed object storage system designed to scale from a single machine to thousands of servers
  - Optimized for multi-tenancy and high concurrency
  - Ideal for backups, web and mobile content, and any other unstructured data that can grow without bound
- Provides a simple, REST-based API fully documented at *https://docs.openstack.org/swift/latest/*
- Originally developed as the basis for Rackspace's Cloud Files and waa open-sourced in 2010 as part of the OpenStack project
- More info at *https://docs.openstack.org/swift/latest/*

In short, Swift is

- Object/blob store
    - Analogous to S3
- Does not care about type of data
- Apache licensed (python)
- Production quality
- In essence: software-defined object storage

Features of Swift i

- Distributed
- Scalable to several peta-bytes
- Eventually consistent (CAP theorem)
    - If no new updates are made to a given data item, eventually all accesses to that item will return the last updated value
- High availability
- Robust
- Offer REST APIs

Features of Swift ii

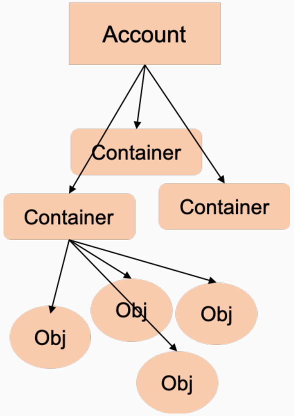
- Support S3 APIs
- Can work on commodity HW
- Quotas and access control
- Various storage policies
- Offers authentication

What you cannot do with Swift

- Mount it
- Have file hierarchies
- Store a live database
- Format it to a file system
- Do anything that API doesn't allow
- Store objects sized more than a certain amount (5 GB, but check it!)

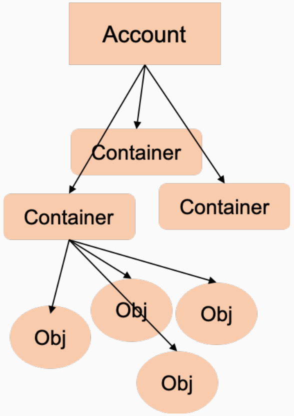
Swift object storage system: a hierarchy i

- Account: top level of hierarchy
  - Service provider creates your account and you own all resources in that account
  - Defines a namespace for containers(!), container can have same name in 2 accounts
  - Synonymous of project or tenant



Swift object storage system: a hierarchy ii

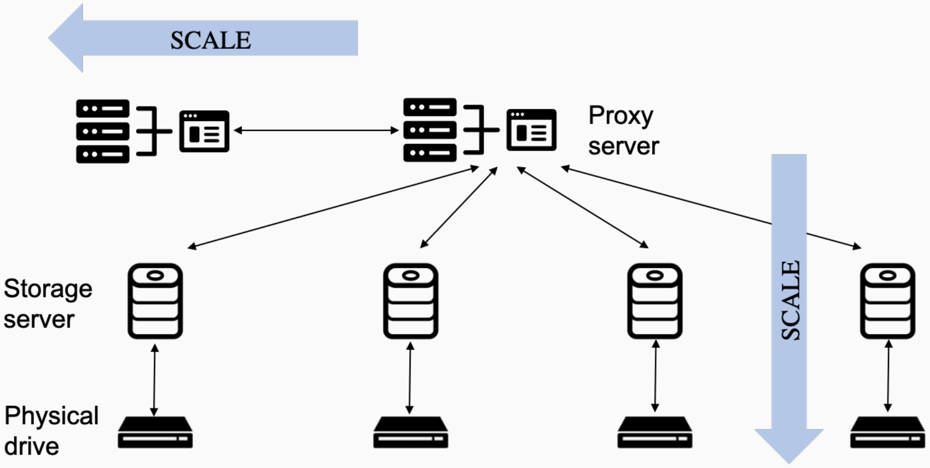
- Container: namespace for objects (think it as a folder)
  - Object with the same name in two containers represents two different objects
  - Freedom to create any number of containers within an account
  - ACLs to access objects are associated to containers (not individual objects)
  - You can define storage policy on a container
- Object: stores data content
  - Uncompressed and unencrypted format, with metadata
  - Documents, images



Swift architecture

- Proxy server
- Ring
- Storage servers: object server, container server, account server
- Consistency servers: replication, reconstruction, updaters, auditors

Simplified architecture



Proxy server

- Responsible for tying together the architecture
    - For each request: look up the location of the account, container or object in the ring (see later) and route the request accordingly
        - Accepts user requests
        - Responsible for implementing Swift APIs
        - Makes sure that communication with storage servers is up and working
        - Answers to clients
        - Handles failures

SWIFT API

- Most important ones
    - GET: downloads objects, lists the contents of containers or accounts
    - PUT: uploads objects, creates containers, overwrites metadata headers
    - POST: creates containers if they do not exist, updates metadata (accounts or containers), overwrites metadata (objects)
    - DELETE: deletes objects and containers that are empty
    - HEAD: retrieves header information for the account, container or object

Swift storage policies

- Way to differentiate service levels
  - Exposed to client via abstract name
  - Each storage policy has a ring, which may include a subset of HW
  - Per-container basis
    - All objects of a container will follow the policy
- Examples
  - Some containers only uses 2x replication
  - Some containers only use SSDs

Swift rings i

- Determine where data should reside (consistent-hashing ring, mostly a map)
  - Determines which device is used for handoff in case of failure
- Separate ring for accounts, containers and individual storage
  - Externally management (servers do not modify them)
  - Each ring maintains mapping using availability zones, partitions and replicas

Swift rings ii

- Partitions replicated 3 times across the cluster (default policy)
    - Locations for partition stored in the mapping maintained by the ring
- Replicas in each partition isolated into as many distinct regions, zones, server and devices as possible -> Isolated failure domains
    - Regions (geographical, e.g. west or east cost), zones (physical location, power separation, network separation or any other attribute that would lessen multiple replicas being available at the same time)
    - You may have less failure domains than replicas of the partition (e.g. 3 replica cluster with 2 servers), warning is issued by the ring management tool
- Data is evenly distributed across the capacity available based on weights
    - Weights are useful with different sized drives within a cluster

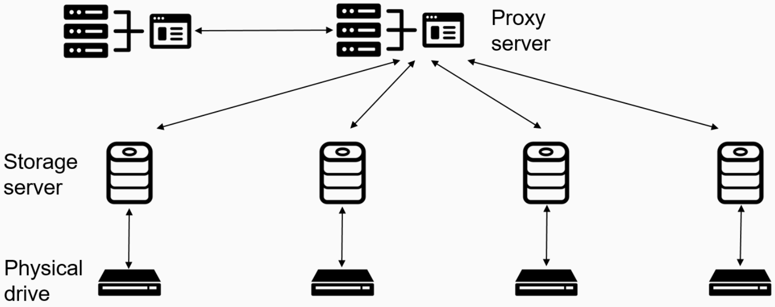
Storage servers

- Object server
    - Simple storage server that stores, retrieves and deletes objects stored on local devices
    - Objects stored as binary files on the filesystem with metadata
- Container server
    - Handle listings of objects
        - Does not know where object are, just that they are in a specific container
        - Listings kept in a replicated SQLite DB
        - Keep statistics
- Account server
    - Similar to container server, lists containers rather than objects

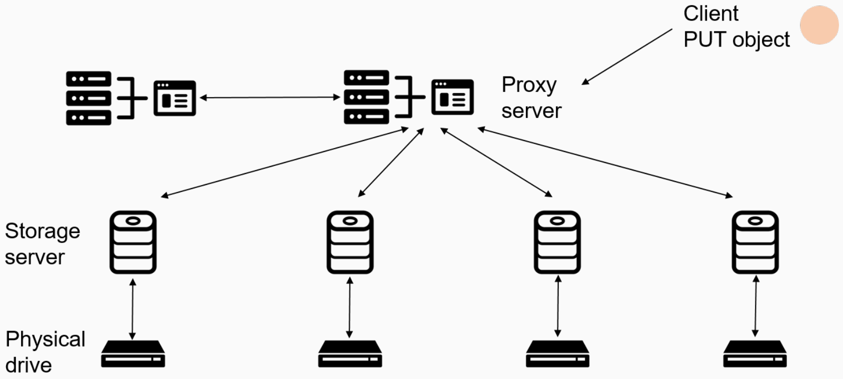
Consistency servers

- Replicator
    - Keep the system in a consistent state in the face of temporary error conditions (network outages or drive failures)
- Updater
    - Sometimes container or account data cannot be immediately updated (congestion or failure)
    - Update is queued locally on the filesystem and updater will process failed updates
- Auditors
    - Crawl local server checking integrity of objects, containers and accounts
        - If corruption is found file is quarantined and replicator will replace the bad file from another replica
        - Errors are logged

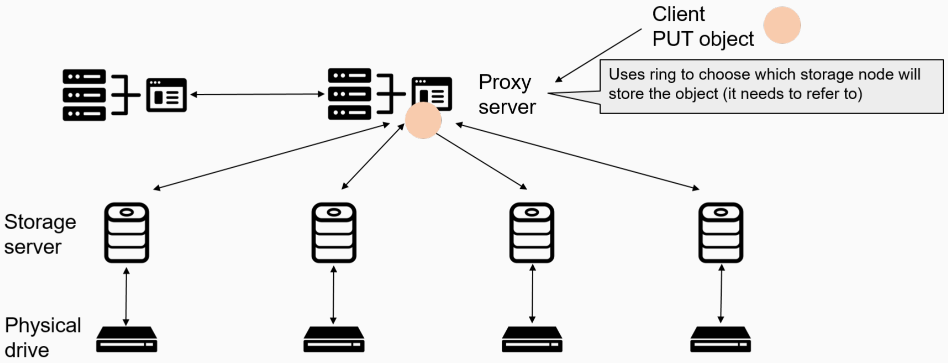
Swift put operation in essence



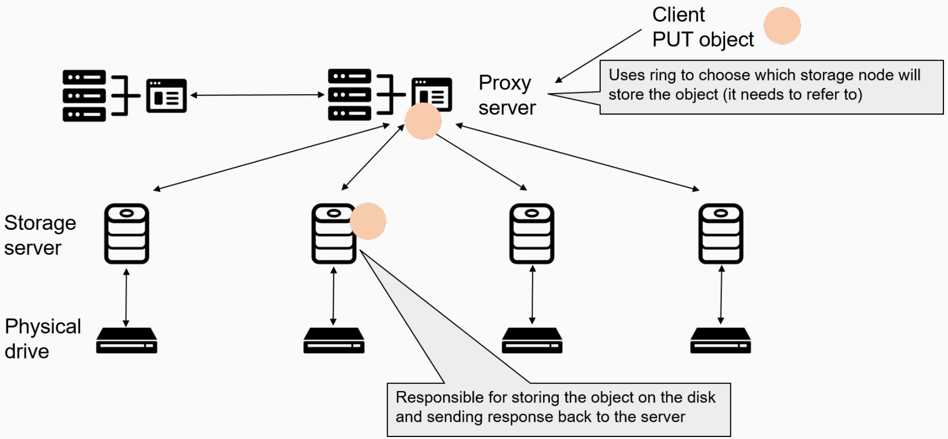
Swift put operation in essence



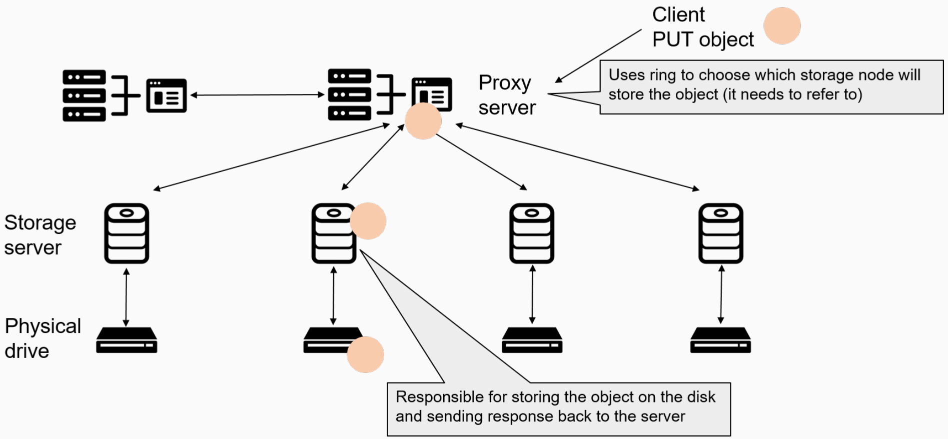
Swift put operation in essence


Client
PUT object
Uses ring to choose which storage node will store the object (it needs to refer to)
Proxy server
Storage server
Physical drive

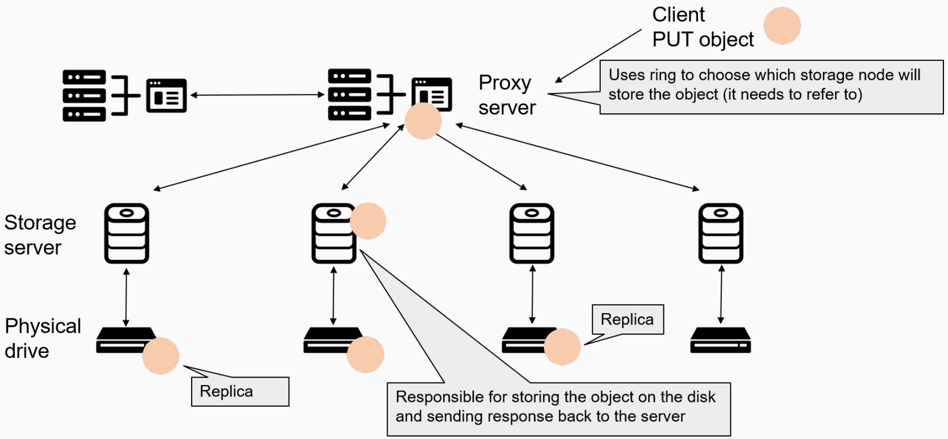
Swift put operation in essence


Client
PUT object
Uses ring to choose which storage node will store the object (it needs to refer to)
Proxy server
Storage server
Physical drive
Responsible for storing the object on the disk and sending response back to the server

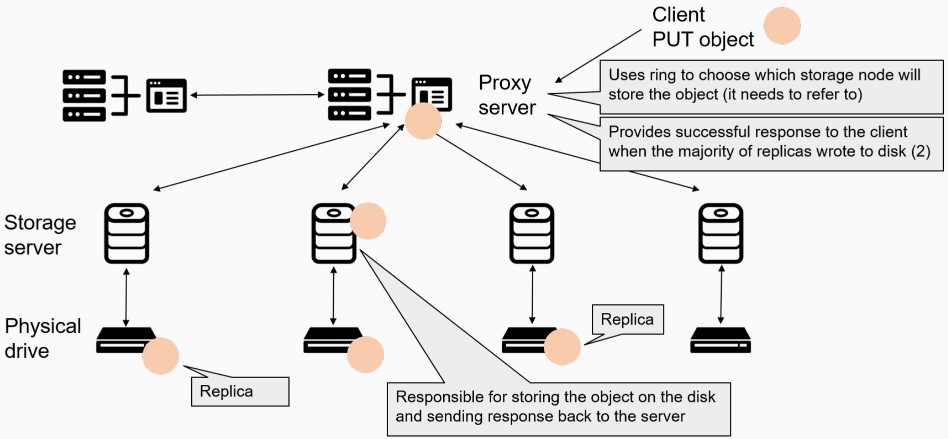
Swift put operation in essence


Client
PUT object
Uses ring to choose which storage node will store the object (it needs to refer to)
Proxy server
Storage server
Physical drive
Responsible for storing the object on the disk and sending response back to the server

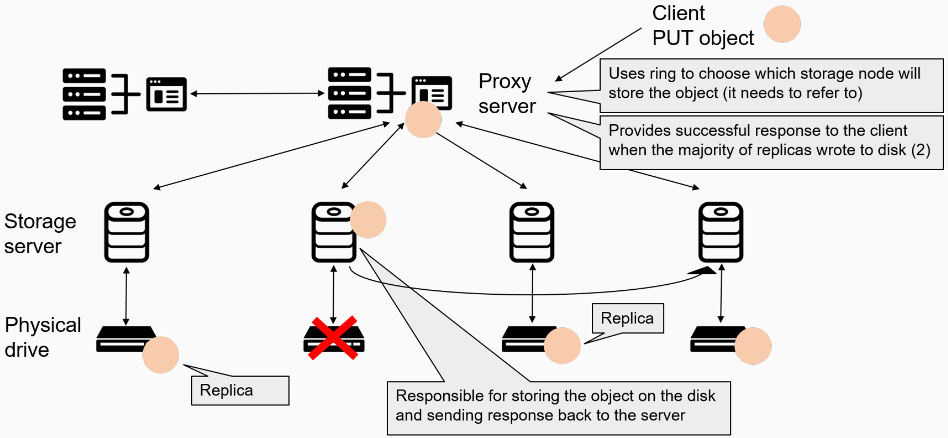
Swift put operation in essence


Client
PUT object
Proxy
server
Uses ring to choose which storage node will store the object (it needs to refer to)
Storage
server
Physical
drive
Replica
Replica
Responsible for storing the object on the disk and sending response back to the server

Swift put operation in essence


Client
PUT object
Proxy server
Uses ring to choose which storage node will store the object (it needs to refer to)
Provides successful response to the client when the majority of replicas wrote to disk (2)
Storage server
Physical drive
Replica
Replica
Responsible for storing the object on the disk and sending response back to the server

Swift put operation in essence



Client
PUT object
Proxy server
Uses ring to choose which storage node will store the object (it needs to refer to)
Provides successful response to the client when the majority of replicas wrote to disk (2)
Storage server
Physical drive
Replica
Replica
Responsible for storing the object on the disk and sending response back to the server