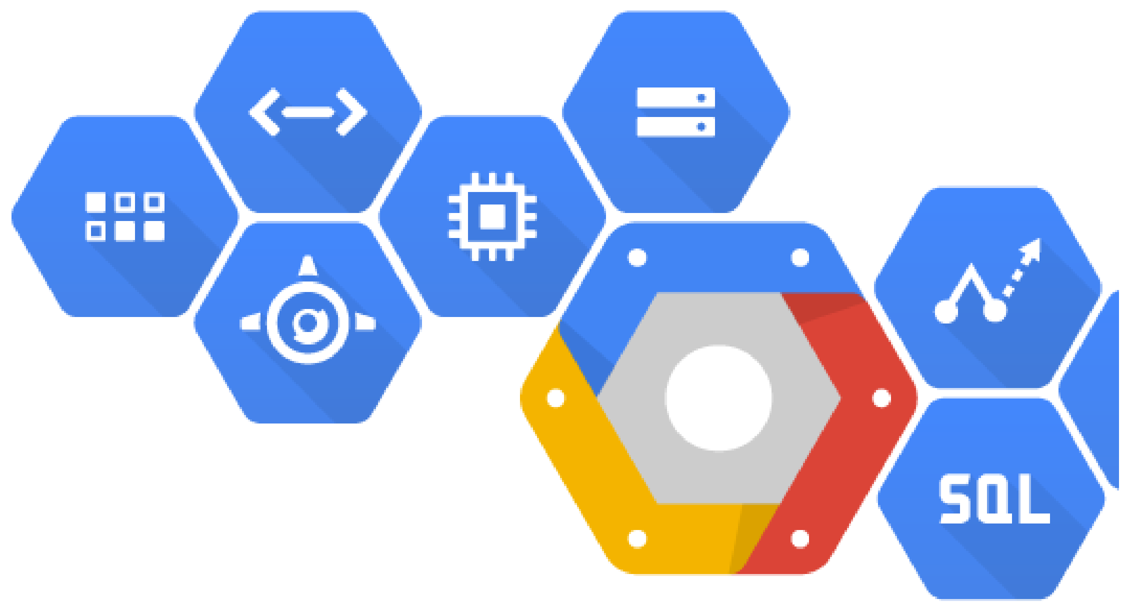
SCC311

Designing Complex Distributed Systems: Google Infrastructure

Part 2

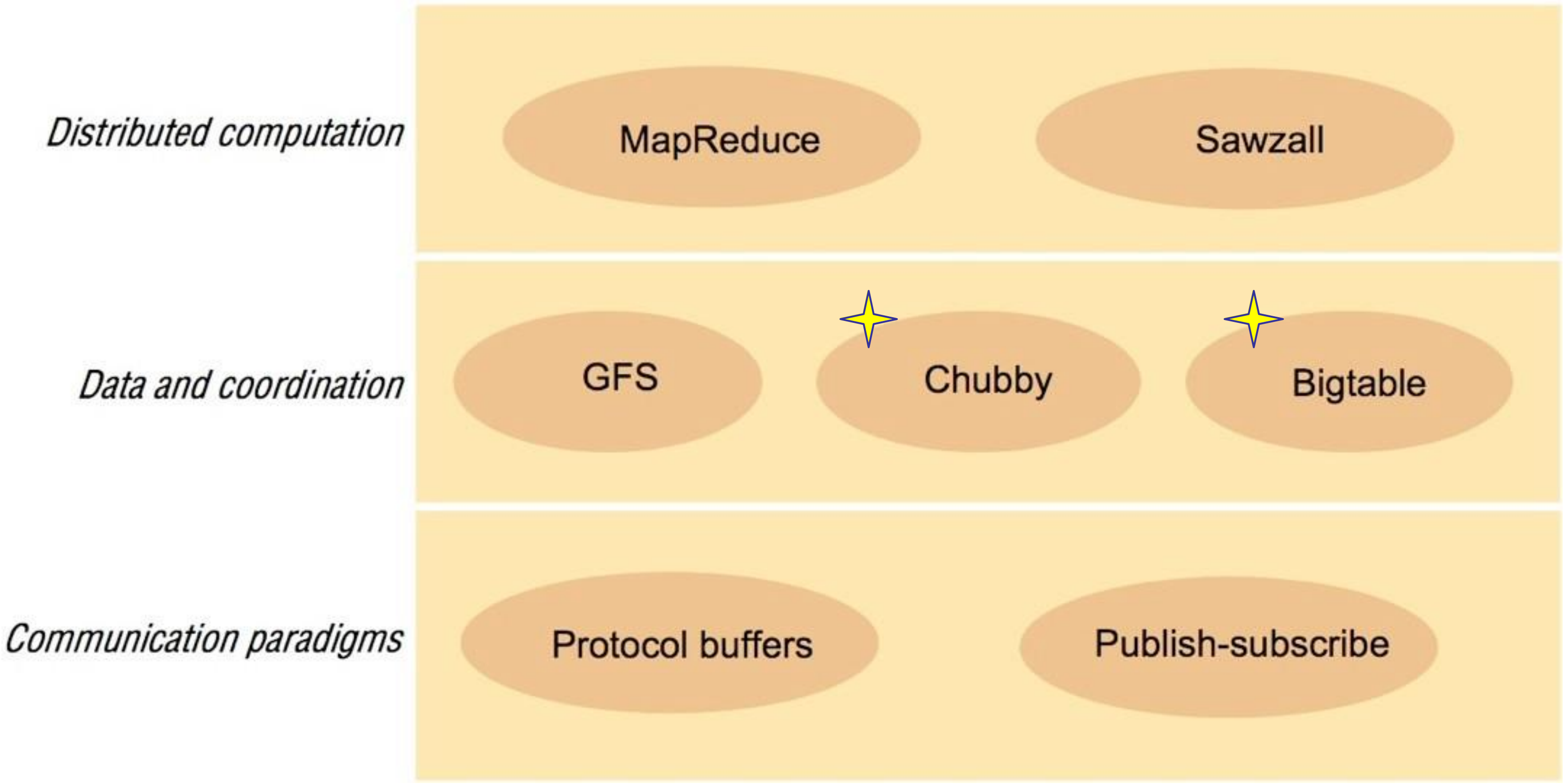
Onur Ascigil



Key Design Philosophies

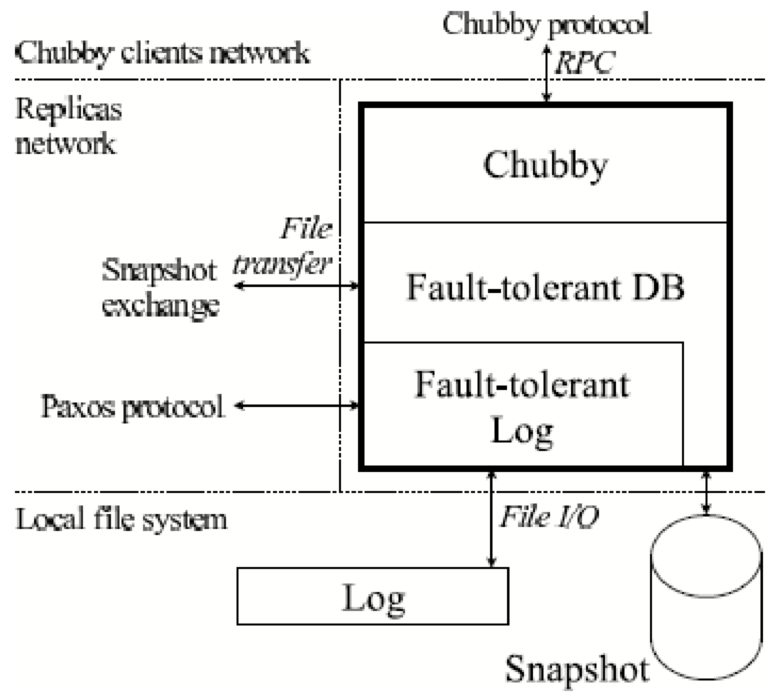
- ***Simplicity***
  - ➔ do one thing and do it well, avoiding feature-rich designs
- ***Performance***
  - ➔ ‘every millisecond counts’
  - ➔ estimate performance through back-of-the-envelope calculations of primitive operations
    - e.g. memory/disk access, sending packets, locking/unlocking a mutex, etc.
- ***Testing***
  - ➔ ‘if it ain’t broke, you are not trying hard enough’
  - ➔ complemented by a strong emphasis on *logging* and *tracing* to detect and resolve faults
- Not adhering to any convention/school of thought. Whatever works best.
- Use cheapest commodity hardware (risk analysis).

Distributed Systems Infrastructure



Chubby

- A filesystem to store small files and locks
- **Primary focus**: reliability and availability to a moderately large set of clients
  - Throughput and storage capacity were considered secondary
- Supports coarse grained synchronisation
  - Typical use case: locks that are acquired for long term (minutes or more)
- Example uses
  - ➔ To achieve distributed coordination
  - ➔ As a file system to store small files
    - **e.g., for storing meta-data**



Chubby

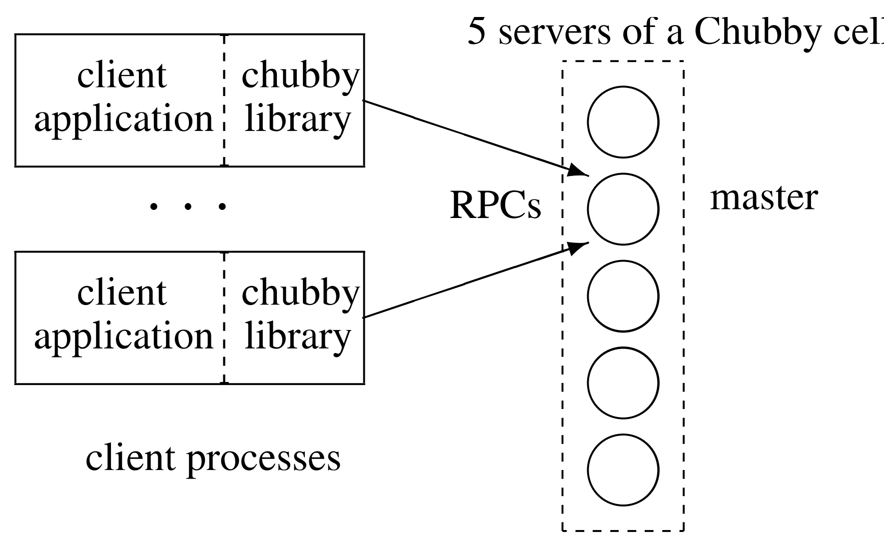
- *Initially*: each file is a lock
- *Then*: associate a (small) data object with each lock
- **Atomic** read and write operations that are designed for small files
  - ➔ whole-file operations (open close, delete, …) to discourage storing large files
  - ➔ no moving of files
  - ➔ minimal metadata

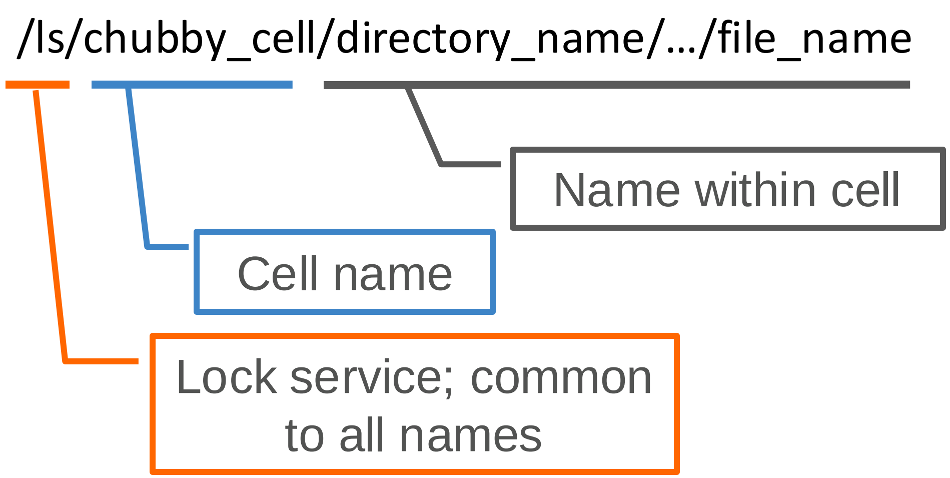
Chubby API

Role	*Operation*	*Effect*
General	*Open*	Opens a given named file or directory and returns a handle
	Close	Closes the file associated with the handle
	Delete	Deletes the file or directory
File	*GetContentsAndStat*	Returns (atomically) the whole file contents and metadata associated with the file
	GetStat	Returns just the metadata
	ReadDir	Returns the contents of a directory – that is, the names and metadata of any children
	SetContents	Writes the whole contents of a file (atomically)
	SetACL	Writes new access control list information
Lock	*Acquire*	Acquires a lock on a file
	TryAquire	Tries to acquire a lock on a file
	Release	Releases a lock

Chubby

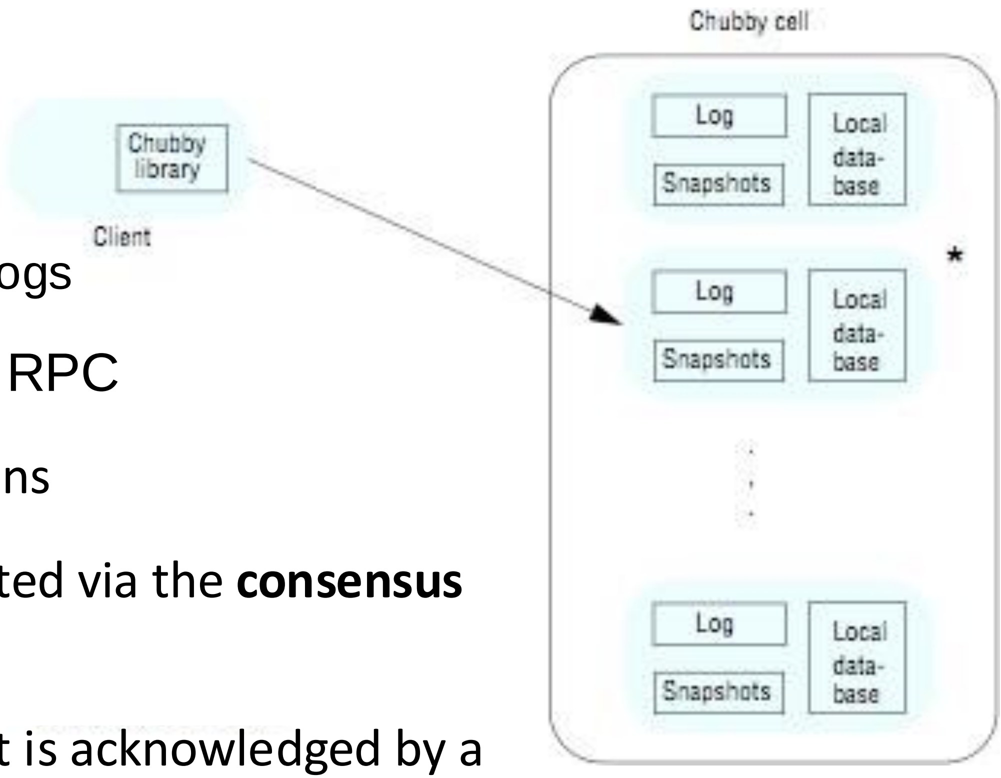
- Simple directory structure
  - root=cells
  - leaves=files



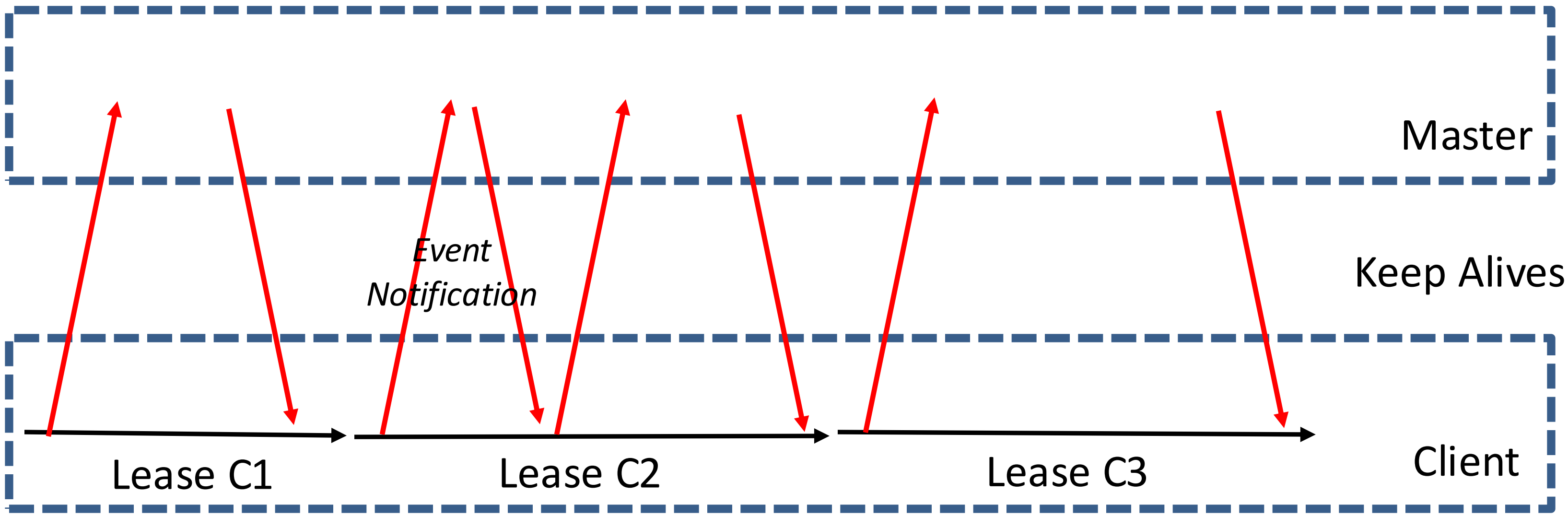


Replication in Chubby

- Each cell consists of replicas (default=5)
- Each replica is made up of:
  - ➔ a DB of directories and files
  - ➔ logs to ensure consistency
  - ➔ snapshots to reduce size of logs
- Clients connect to cells using RPC
- Atomic Read and Write operations
  - Write requests are propagated via the **consensus protocol** to all replicas.
  - Writes are accepted when it is acknowledged by a majority of the replicas in the cell.
  - Read requests are satisfied by the master alone



Caching and Event Subscription



- Clients connect to cells using RPC
  - ➔ Sessions maintained using *KeepAlive* RPC calls
- Client caching of files is used to improve performance
  - ➔ Consistency maintained by invalidating all caches (piggybacked on *KeepAlives*) during which a write call is blocked
- **Subscription**: an application can register for event notification such as modification on a file (notification sending is piggybacked on KeepAlives)

Use-cases

- Name Service
  - A very simple protocol for cache consistency with deterministic semantics
- Important for client services
  - ➔ e.g. BigTable stores Access Control Lists (ACLs)
- Master replica election

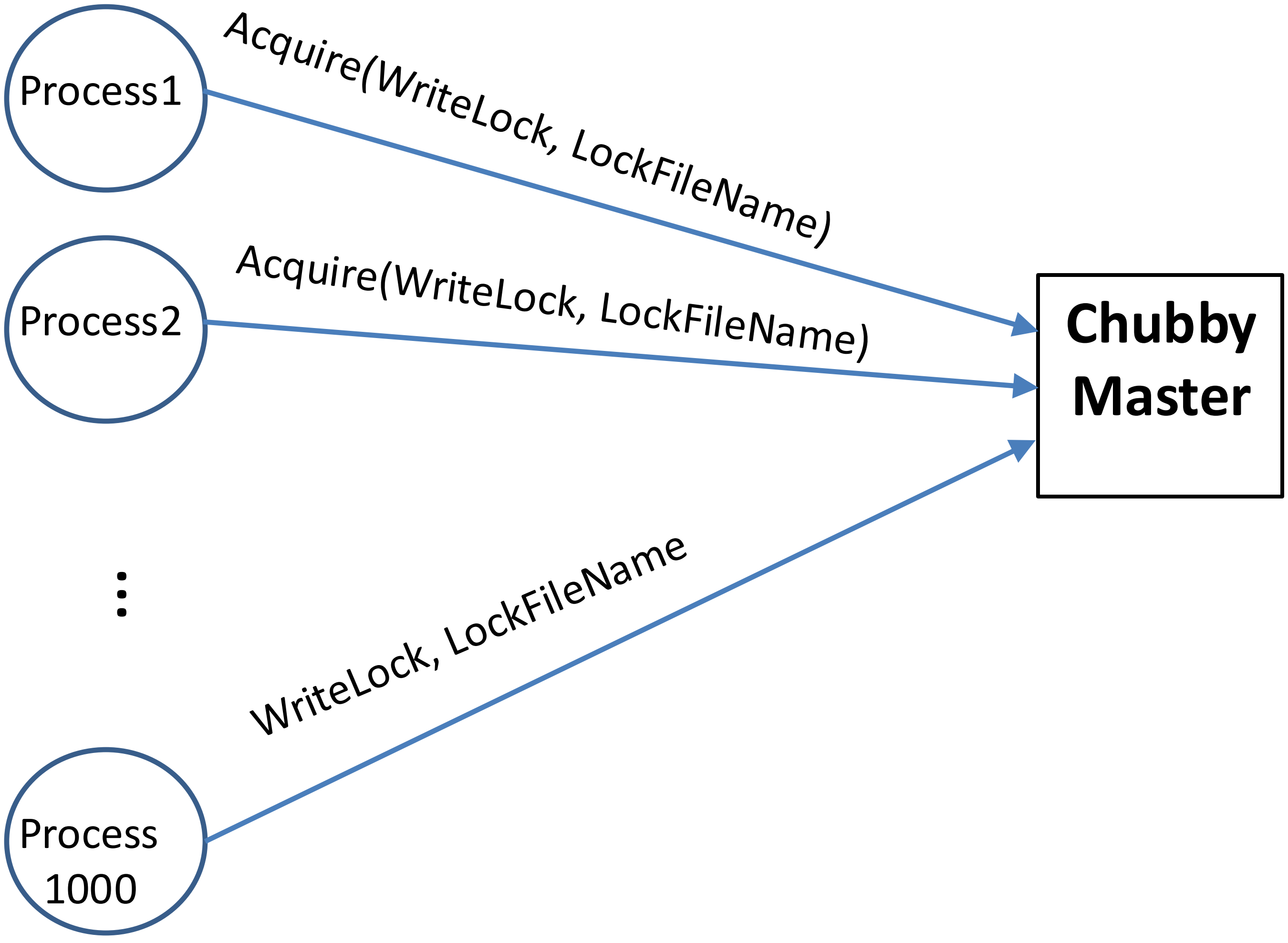
Name Service using Chubby

- Applications run jobs involving **thousands of processes**.
  - Each process to communicate with every other leads to a quadratic number of DNS lookups.
- DNS supports caching with Time-to-Live (TTL)
  - A cached entries is discarded when its TTL expires
- A small TTL value is desirable for Google for prompt replacement of failed services
  - However, a small TTL can lead to a massive number of requests at the Google's DNS server.
  - Example: 3,000 clients would require about 150,000 lookups per second (estimate by Google)
- The caching semantics provided by Chubby invalidates cached entries pro-actively and avoids large number of periodic DNS resolution requests.

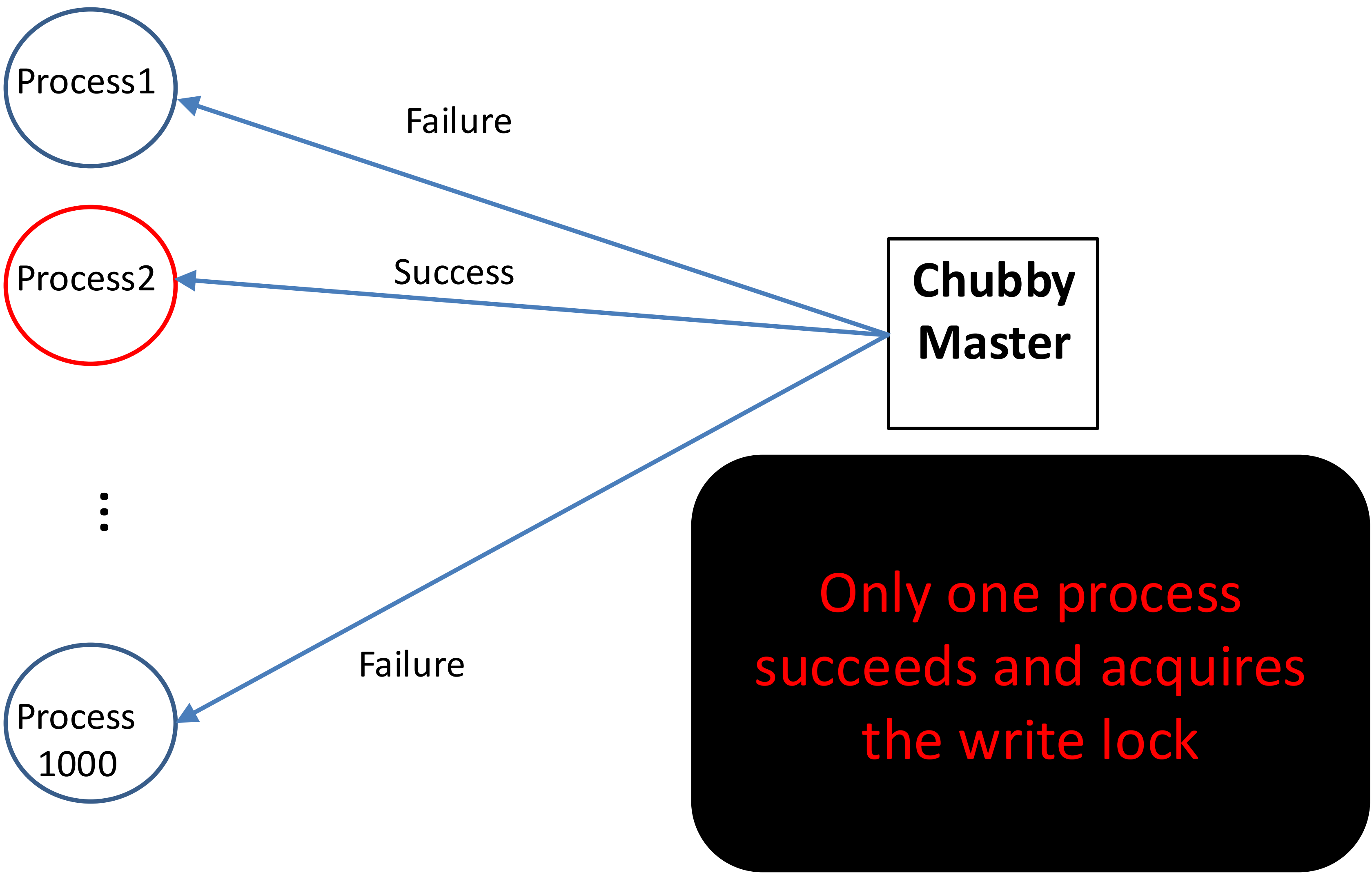
Master Election using Chubby

- Both GFS and BigTable use Chubby to elect a master
- Processes can use the Chubby API to open a lock file and attempt to acquire a write lock.
- Only one succeeds and becomes the master.
- The others act as replicas.
- The primary then writes its identity to the lock file with SetContents().
- The other replicas can call GetContentsAndStat() on the lock file (perhaps in response to a file modification event) to find out who the primary is.

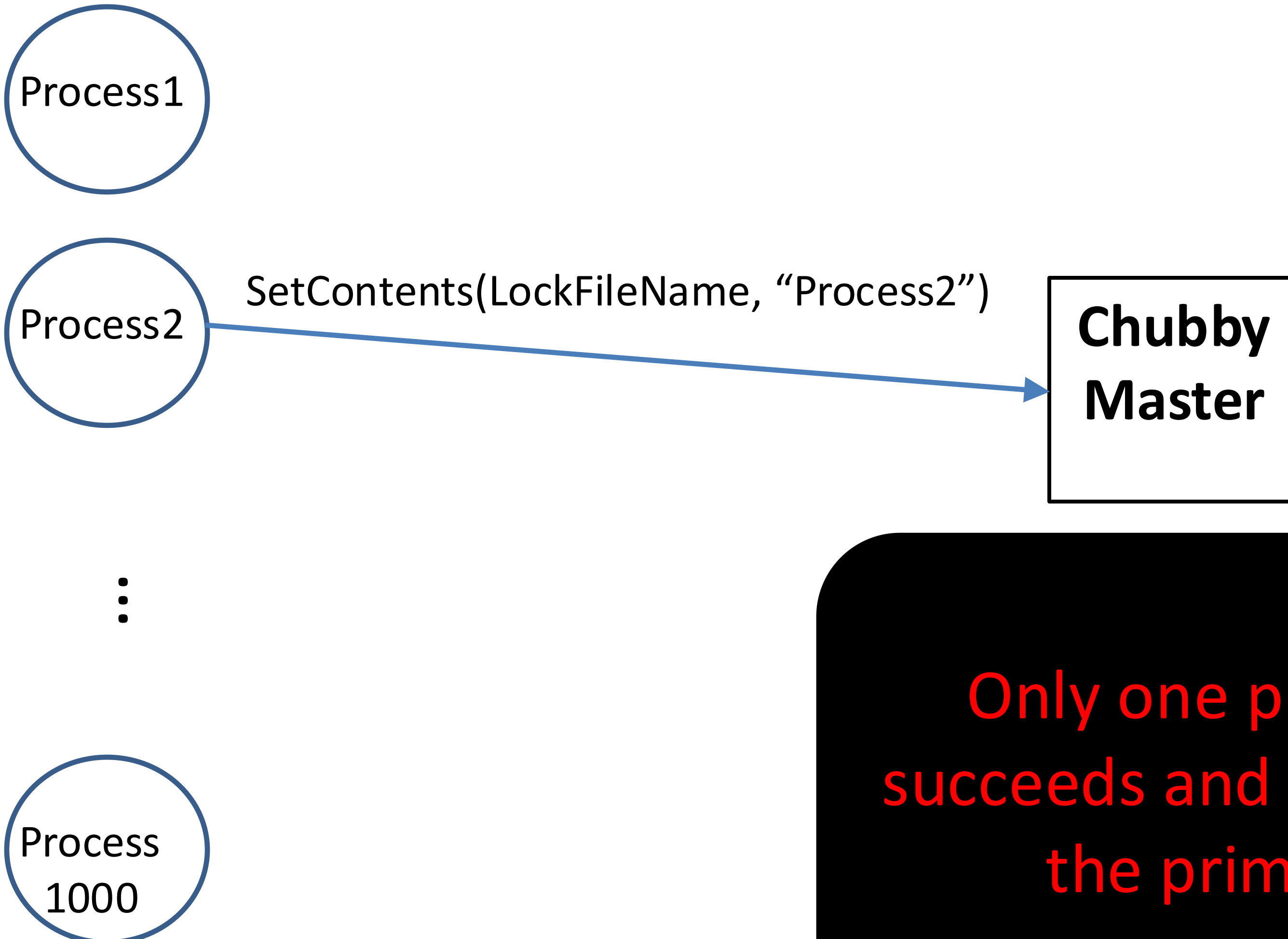
Primary Election using Chubby



Primary Election using Chubby

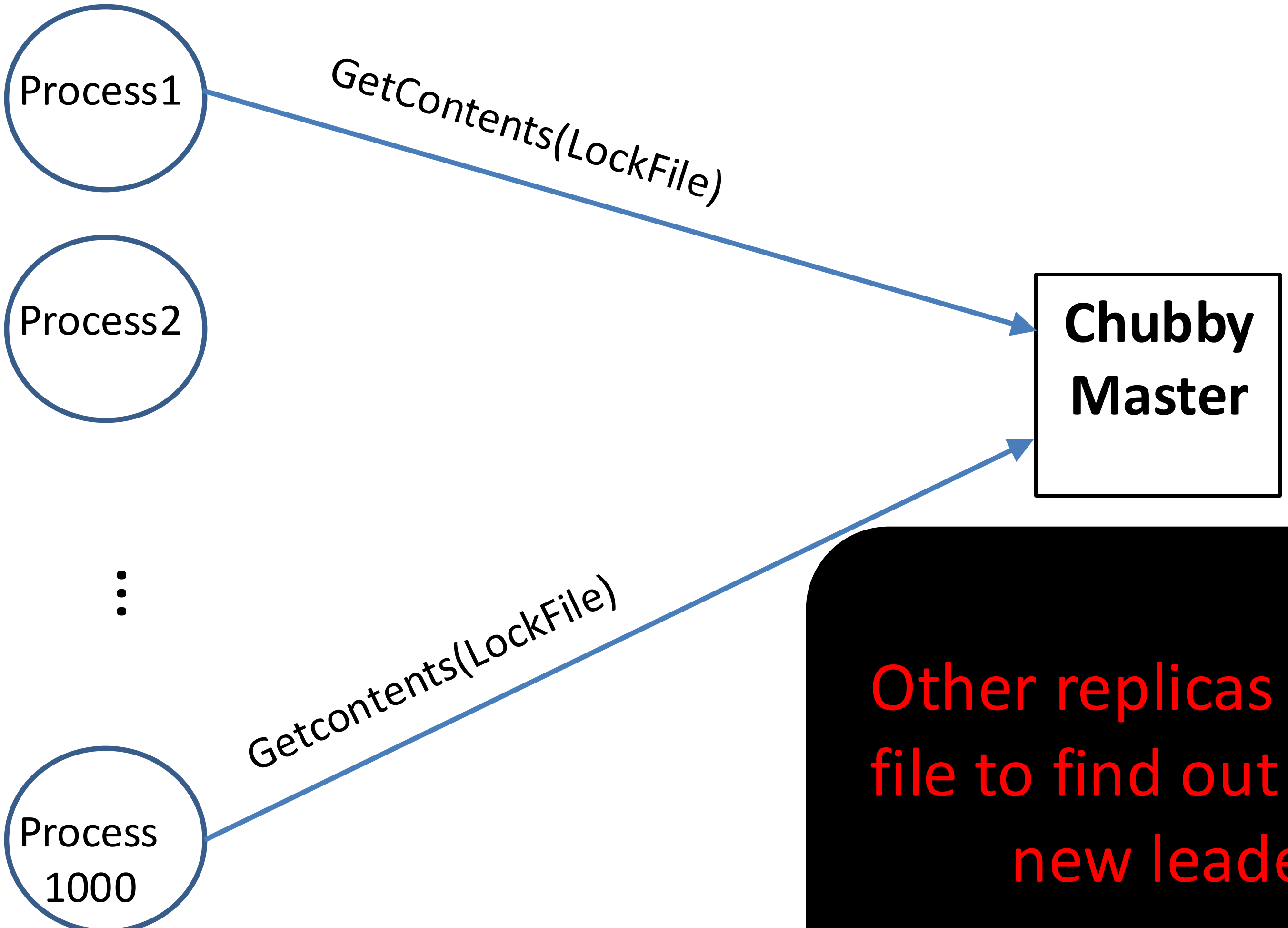


Primary Election using Chubby



Only one process succeeds and becomes the primary

Primary Election using Chubby



Other replicas read the file to find out who the new leader is

Primary Election using Chubby

- **Main advantage:** by using Chubby to elect a leader, other applications and services do not have to implement their own distributed consensus library.
- Implementing a master (i.e., primary) election requires only a couple of RPC calls
- GFS and BigTable both use Chubby to elect a master.

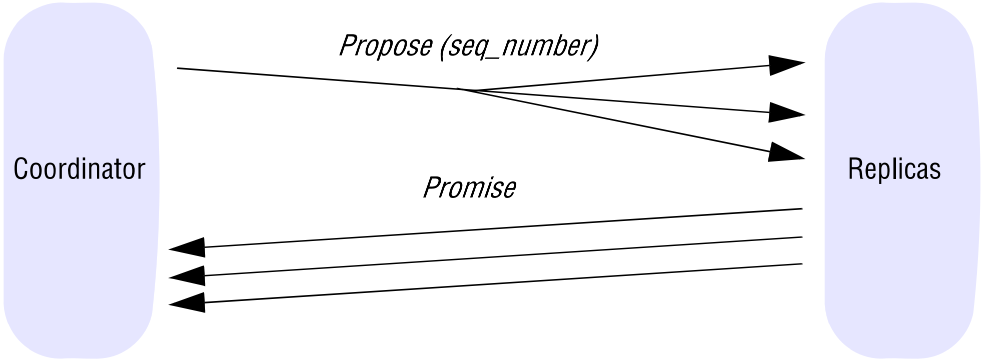
Chubby's use of Paxos

- Any node in a distributed system can fail, including a leader
  - *Paxos*, a distributed consensus protocol for asynchronous systems, can be used for implementing a flexible election process (See: Week 5, Lecture 2)

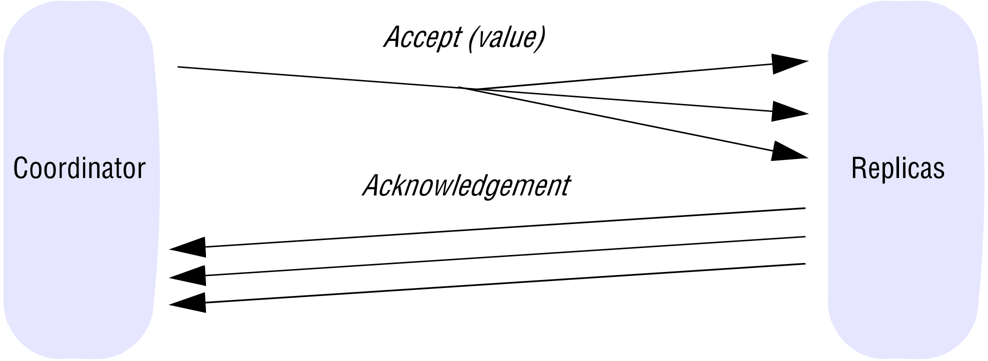
- **Step 1: Electing a leader**
  - *Leaders are ordered using sequence numbers*
  - *Each node maintains the highest sequence number seen so far*
  - *If bidding to be leader, a node will pick a higher unique number, and broadcasts this to all other nodes [Propose]*
  - *On receiving a Propose message, other nodes will reply with a [Promise] message if they have not seen a higher bidder*
  - *If a majority of promise messages is received (a quorum is achieved), a bidding node knows it has mandate to start acting as a leader*

Chubby's version of Paxos

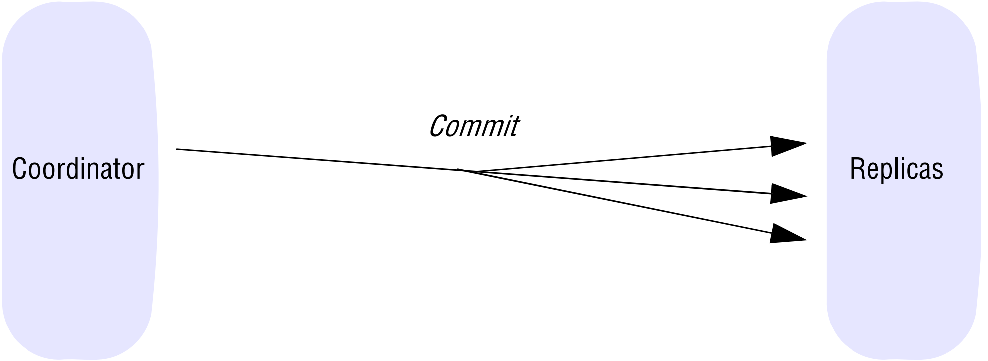
Step 1: Electing a leader



Step 2: Seeking a Consensus



Step 3: Achieving Consensus



Leader Election in Chubby

- The replicas use Paxos to elect a master
- A lease-based leadership
- The master must obtain votes from a majority of the replicas and promises that those replicas will not elect a different master for an interval of a few seconds known as the master lease
- The master lease is periodically renewed by the replicas provided the master continues to win a majority of the vote.

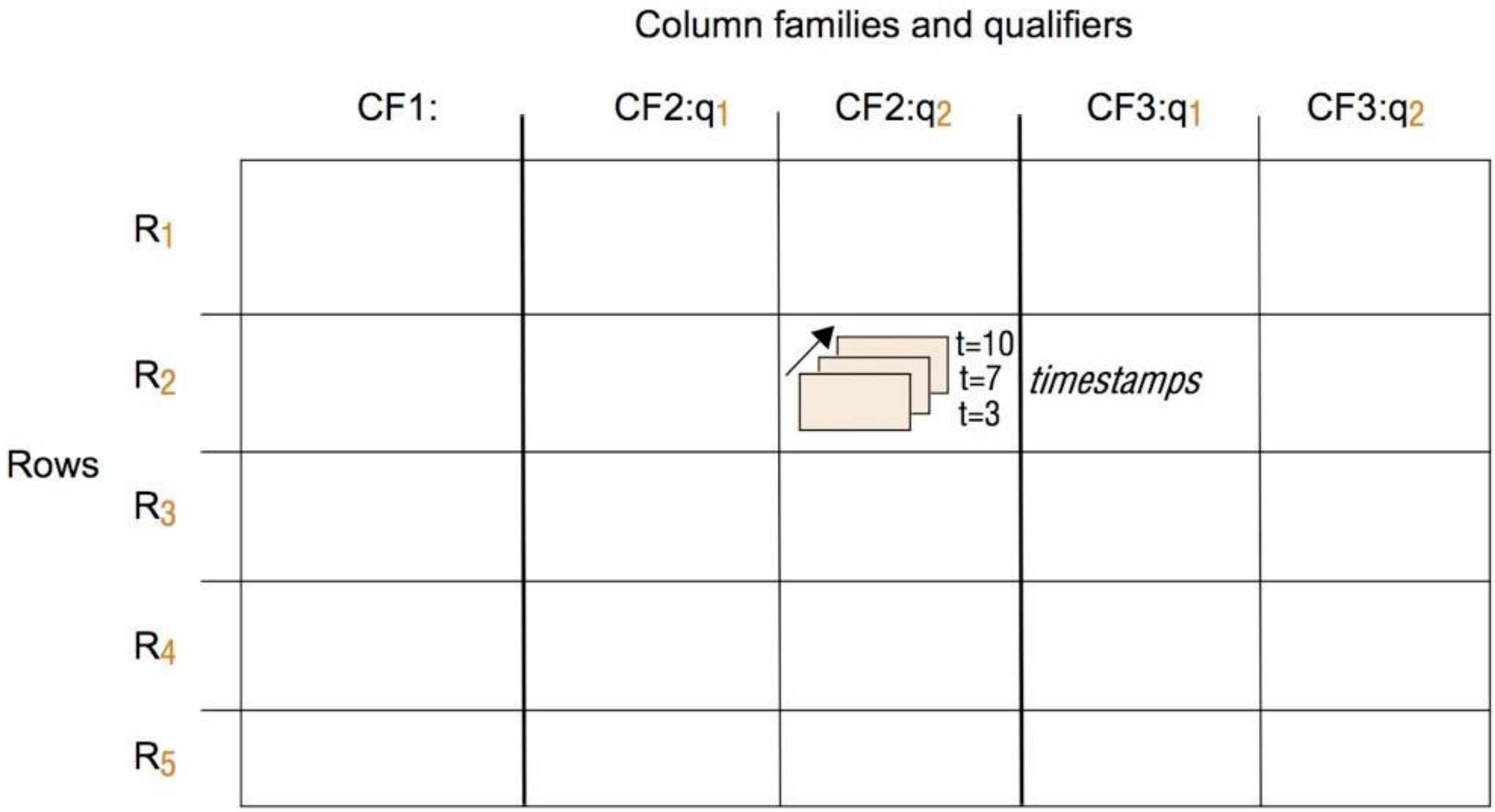
Bigtable

- GFS and Chubby solve file storage and lock systems
  - ➔ What about large indexed datasets?
  - ➔ Relational DBs are too slow and too complex
- The Google way:
  - ➔ Strip away all performance-deteriorating features of RDBMS
  - ➔ Offer a minimal interface to keep things simple
- Bigtable!
  - ➔ A **very large-scale distributed table** built on top of GFS and Chubby
  - ➔ Supports **semi-structured / structured data**
    - without full relational operators, e.g. join
  - ➔ Able to to scale to billions of rows and thousands of columns
  - ➔ Used by over 60 Google products including Search, Maps, Analytics

Bigtable

- Table is indexed by **row, column and timestamp**
- Table split into fixed size **tablets** optimised for GFS
  - Subsequences of rows map onto tablets, which are the unit of distribution and placement
- Same **client, leader, worker pattern** as with GFS (workers manage tablets) but also with select (client) caching for tablet location
- ‘Cheap’ optimisations:
  - ➔ Rows in a table are lexicographically ordered
  - ➔ Columns are accessed using "family:qualifier"
  - ➔ Considerably simpler than RDBMS: No support for complex relational operations such as joins, unions, intersections, etc.
- Often used with MapReduce for computations on large-scale datasets
  - ➔ e.g. Web search:
    - row_key=URL, columns=webpage_attributes, timestamp=when_captured

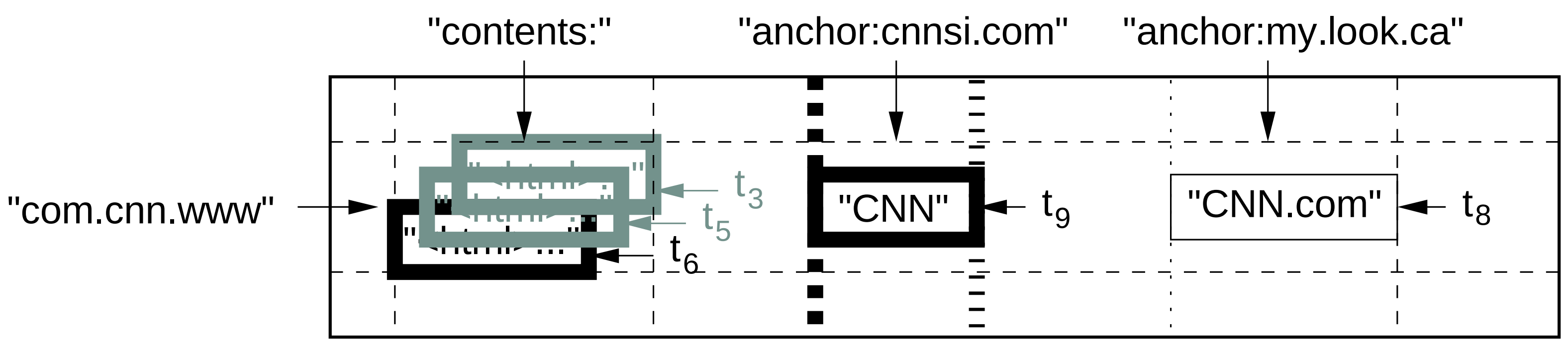
Bigtable (continued)



Bigtable maintains a lexicographic ordering of a given table by row key

Example use case: Storing Web pages

- It is common within Google to process information about web pages
- The row name is a reversed URL
- The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page

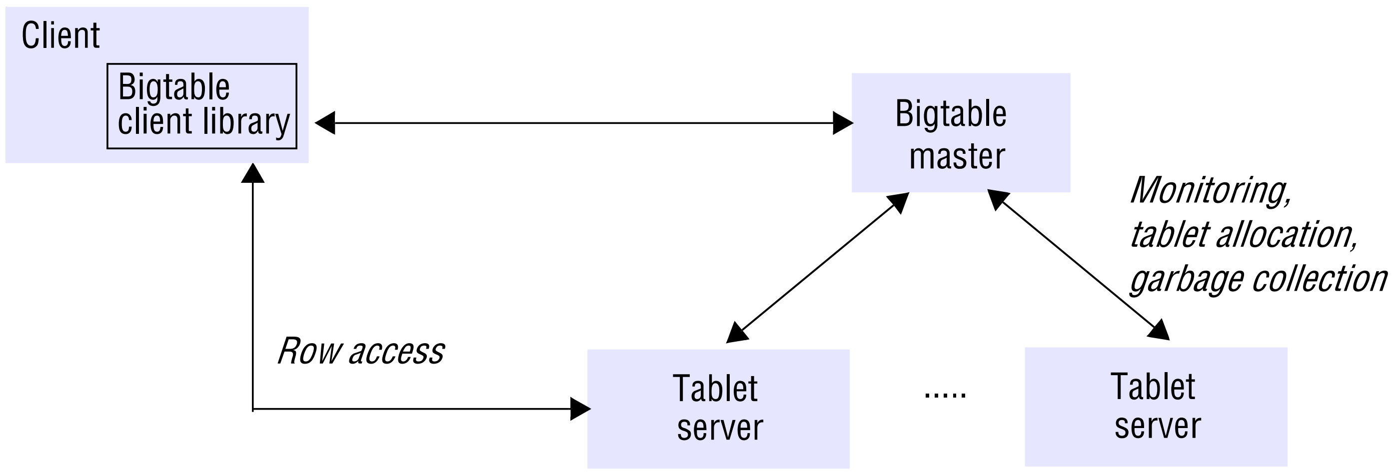


BigTable API

Bigtable supports an API that provides a wide range of operations, including:

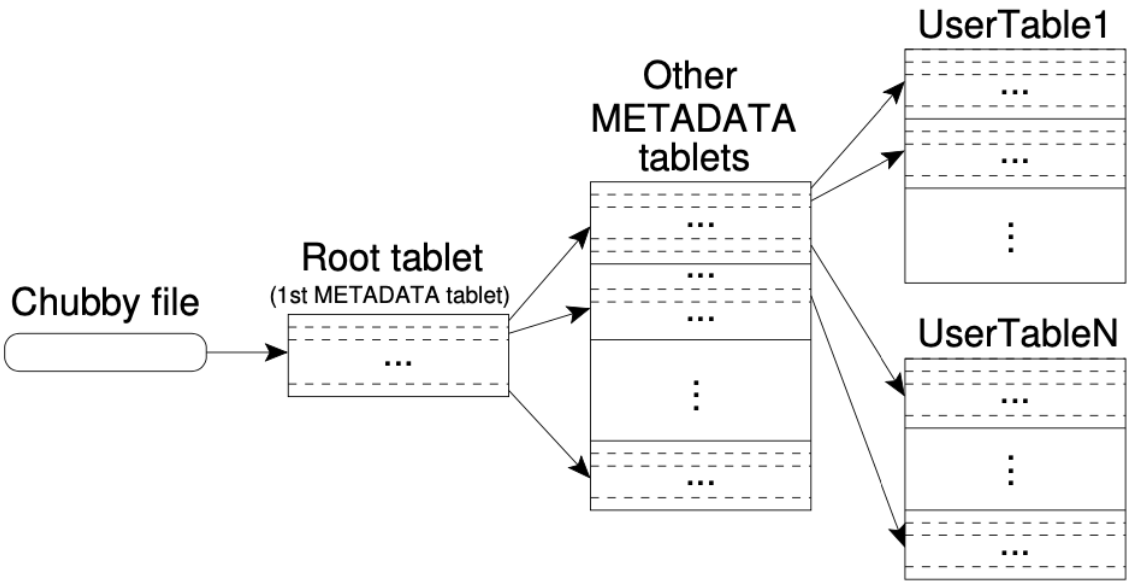
- The creation and deletion of tables
- The creation and deletion of column families within tables;
- Accessing data from given rows;
- Writing or deleting cell values;
- Carrying out **atomic row mutations** including data accesses and associated write and delete operations (more global, cross-row transactions are not supported);
- Iterating over different column families, including the use of regular expressions to identify column ranges;
- Associating metadata such as access control information with tables and column families.

BigTable Architecture



- A master server and a potentially large number of tablet servers.
- A master mainly does two things:
  - Monitoring the status of tablet servers
  - Ensuring effective load balancing
- Clients communicate directly with the tablet servers (each holding a tablet, i.e., a range of row keys) for reads and writes.

Tablet Index



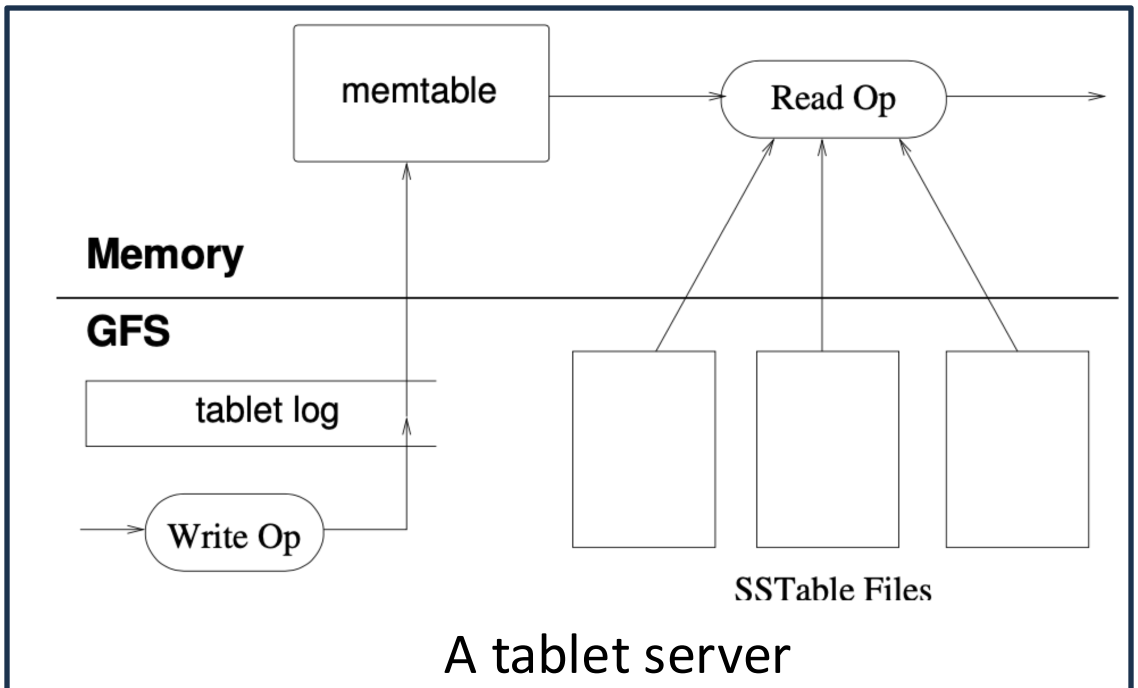
- Root tablet contains pointers to METADATA tablets, and each METADATA tablet, in turn, stores the location of user tablets by encoding their start and end row keys.
- Clients first consult a Root metadata table to find out which tablet server stores the metadata for the row key range they need.
- Once the client knows which tablet server holds the desired tablet, it sends requests directly to that server (a similar design to GFS).

SSTable(1)

- The tablets are stored as one or more ***SSTables*** (Sorted Strings Table) in GFS: An ordered, **immutable** map from keys to values.

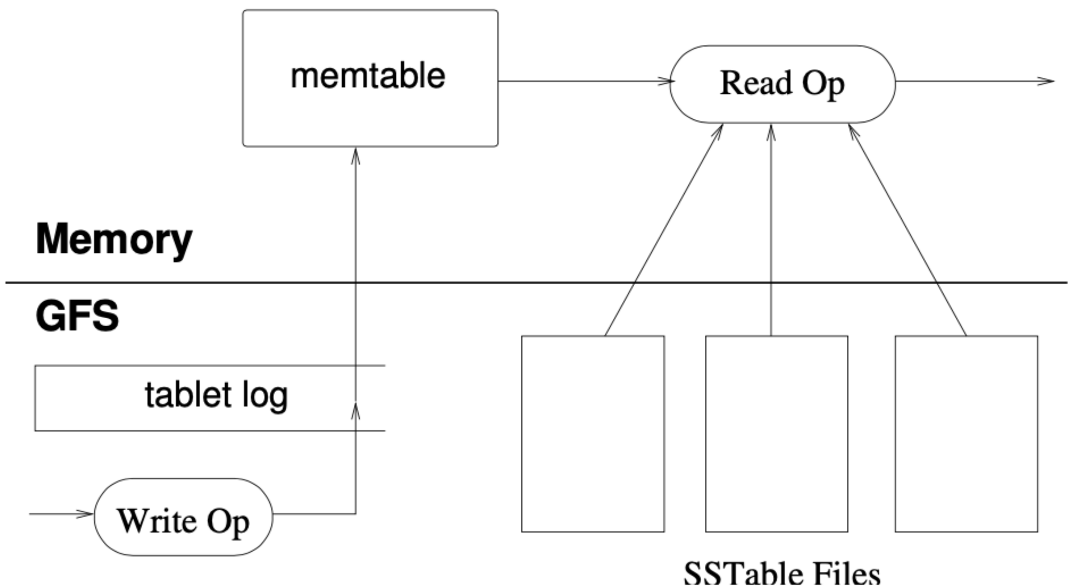
Updates to tablets are batched and applied periodically:

- **1. Tablet Log**: Updates are first recorded in a **commit log** (write-ahead log) in GFS to ensure recoverability.



SSTable(2)

- **2. Memtable**: The **most recent updates** are stored in the **memtable:** an in-memory sorted structure.
  - The memtable stores the latest committed data to enable quick access to recent changes before they are written to GFS.



A tablet server

BigTable – Compactions

- Memtable & Minor Compaction
  - Updates stored in memtable; grows with writes.
  - When full, memtable is frozen and converted to an SSTable in GFS.
  - Reduces memory use and minimizes commit log data needed for recovery.
  - Reads/writes continue during compaction.
- Merging Compaction
  - Periodically merges multiple SSTables and the current memtable.
  - Reduces the number of SSTables to optimize read performance.

BigTable - Monitoring

- BigTable uses Chubby for monitoring in a rather interesting way
- Maintains a directory in Chubby containing files representing each of the available tablet servers.
- When a new tablet server comes along, it creates a new file in this directory and more importantly, it obtains an **exclusive lock** on this file.
- Masters periodically check the status of the tablet server locks and if a lock is lost by a server, then master can infer a problem with the server.
- The tablet server can also surrender its lock if the machine is needed (reassigned) for a different purpose.

Summary of Design Choices

Element	*Design choice*	*Rationale*	*Trade-offs*
Chubby	Combined lock and file abstraction	Multipurpose, for example supporting elections	Need to understand and differentiate between different facets
	Whole-file reading and writing	Very efficient for small files	Inappropriate for large files
	Client caching with strict consistency	Deterministic semantics	Overhead of maintaining strict consistency
Bigtable	The use of a table abstraction	Supports structured data efficiently	Less expressive than a relational database
	The use of a centralized master	As above, master has a global view; simpler to implement	Single point of failure; possible bottleneck
	Separation of control and data flows	High-performance data access with minimal master involvement	-
	Emphasis on monitoring and load balancing	Ability to support very large numbers of parallel clients	Overhead associated with maintaining global states

Some Cloud Reflection

- Google is much more than just a search engine.
- Similar to Amazon, the cloud was a side-effect of building applications on a massive scale.
- This “side-effect” infrastructure has allowed them to build more applications that were previously not possible.
- This also opened the door for others (as cloud customers).

Additional Reading

- CDKB: Chapter 21
- Lots of material at Google Labs:
  - ➔ The Anatomy of a Large-Scale Hypertextual Web Search Engine
  - ➔ The Google File System
  - ➔ The Chubby Lock Service for Loosely-Coupled Distributed Systems
  - ➔ Paxos Made Live – An Engineering Perspective
  - ➔ MapReduce: Simplified Data Processing on Large Clusters
  - ➔ Bigtable: A Distributed Storage System for Structured Data
  - ➔ Interpreting the Data: Parallel Analysis with Sawzall

  http://research.google.com/pubs/papers.html

Expected Learning Outcomes

At the end of these sessions:

- You should have an appreciation of the challenges of web search and the provision of cloud services
- You should understand the problem of scalability as it applies to the above problems and the trade-offs with other dimensions (reliability, performance, openness)
- You should understand the architecture of Google Infrastructure and how such a middleware solution can support the business
- You should also be aware of the specific techniques used within Google relating to:
  - ➔ ProtocolBuffers
  - ➔ GFS & MapReduce
  - ➔ Chubby
  - ➔ BigTable

  (and why they are designed the way they are and how this contributes to delivering the goals of Google)