Google Cluster Computing Faculty Training Workshop

Module VII: Other Google Technologies

Overview

- BigTable
- Chubby

BigTable

A Conventional Database…

- Data structure:
  - arbitrary ## of rows
  - Fixed number and type of columns
- Supports search based on values in all cells
- Supports synthesis of output reports based on multiple tables (relational operators)

Google's Needs

- Data reliability
- High speed retrieval
- Storage of huge numbers of records (several TB of data)
- (Multiple) past versions of records should be available

Assumptions

- Many times more reads than writes
- Individual component failures common
- Disks are cheap
- If they control database design as well as application design, the interface need not be standard

Reasonable Questions

- Are structured queries necessary?
- Can data be organized such that related data is physically close by nature?
- What is the minimum coordination required to retrieve data?
- Can existing components be leveraged to provide reliability and abstraction?

From Needs to Constraints

- Simplified data retrieval mechanism
  - (row, col, timestamp) • value lookup, only
  - No relational operators
- Atomic updates only possible at row level

But Some Additional Flexibility…

- Arbitrary number of columns per row
- Arbitrary data type for each column
    - New constraint: data validation must be performed by application layer!

Logical Data Representation

- Rows & columns identified by arbitrary strings
- Multiple versions of a (row, col) cell can be accessed through timestamps
    - Application controls version tracking policy
- Columns grouped into column families

Column Families

- Related columns stored in fixed number of *families*
  - Family name is a prefix on column name
  - e.g., "fileattr:owning_group", "fileattr: owning_user", etc.
- Permissions can be applied at family level to grant read/write access to different applications
- Members of a family compressed together

No Data Validation

- Any number of columns can be stored in a row within the pre-defined families
  - Database will not enforce existence of any minimum set of columns
- Any type of data can be stored in any column
  - Bigtable sees only byte strings of arbitrary length

Consistency

- Multiple operations on a single row can be grouped together and applied atomically
  - No multi-row mutation operators available
- User can specify timestamp to apply to data or allow Bigtable to use ‘now()’ function

Version Control

- Cell versions stored most-recent first for faster access to more recent data
- Two version expiration policies available:
  - Retain last *n* copies
  - Retain data for *n* time units

Data Access

- Straight (row, col, ts) lookup
- Also supports (row, col, MOST_RECENT)
- Filtered iterators within row with regex over column names or additional constraints on timestamps
- Streaming access of large amounts of data to and from MapReduce

Implementation

- Uses several other Google components:
  - GFS provides reliable low-level storage for table files, metadata, and logs
  - Chubby provides distributed synchronization
  - Designed to easily interface with MapReduce

Physical Data Representation

- *SSTable* file provides immutable key•value map with an index over all keys mapping key•disk block
    - Index stored in RAM; value lookup involves only one disk seek to disk block

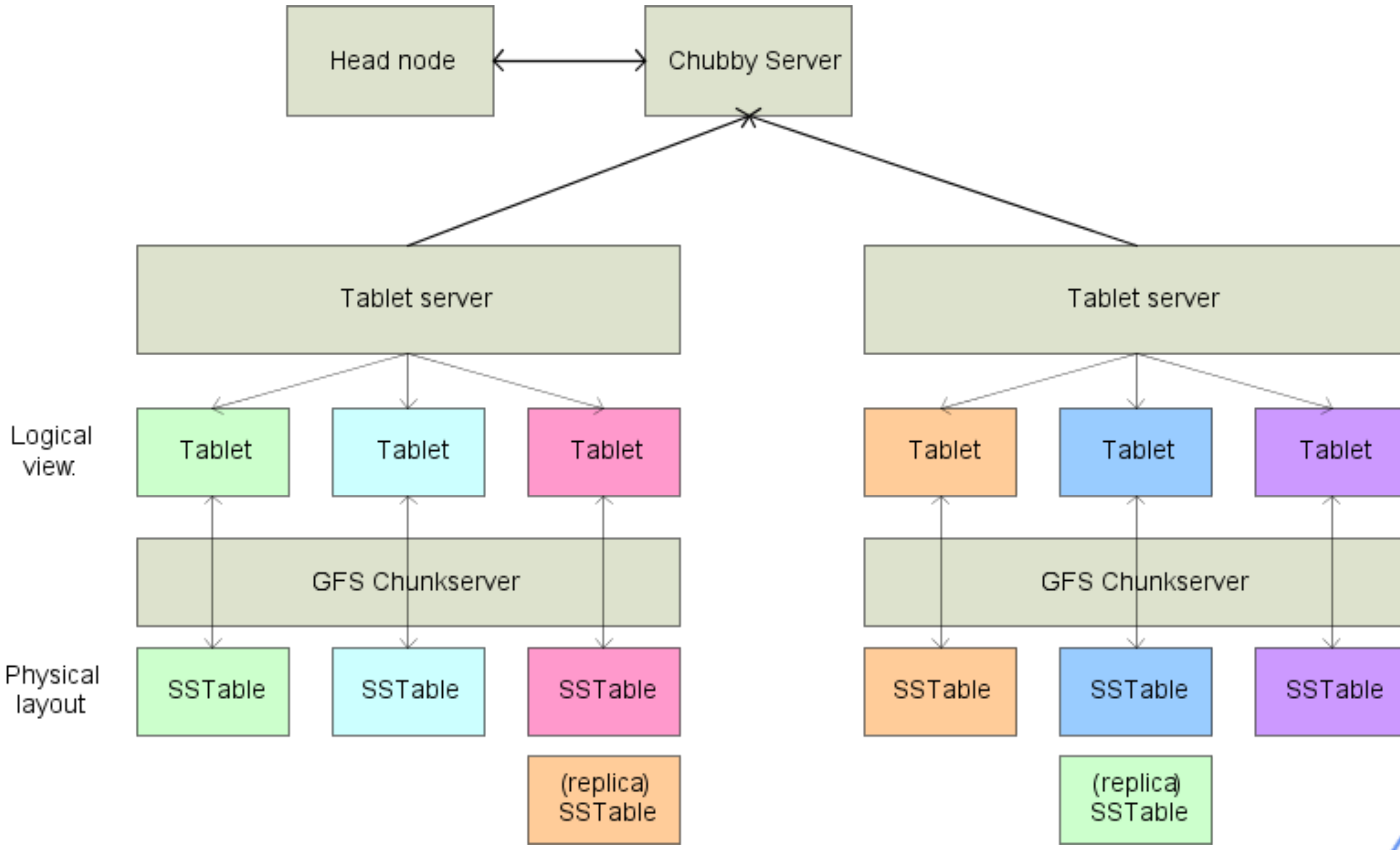
Physical Representation (2)

- A logical "table" is divided into multiple *tablets*
    - Each tablet is one or more SSTable files
- Each tablet stores an interval of table rows
    - If a tablet grows beyond a certain size, it is split into two new tablets

Network Interface

- One master server
  - Communicates only with tablet servers
- Several tablet servers
  - Perform actual client accesses
- "Chubby" lock server provides coordination and mutual exclusion
- GFS servers provide underlying storage

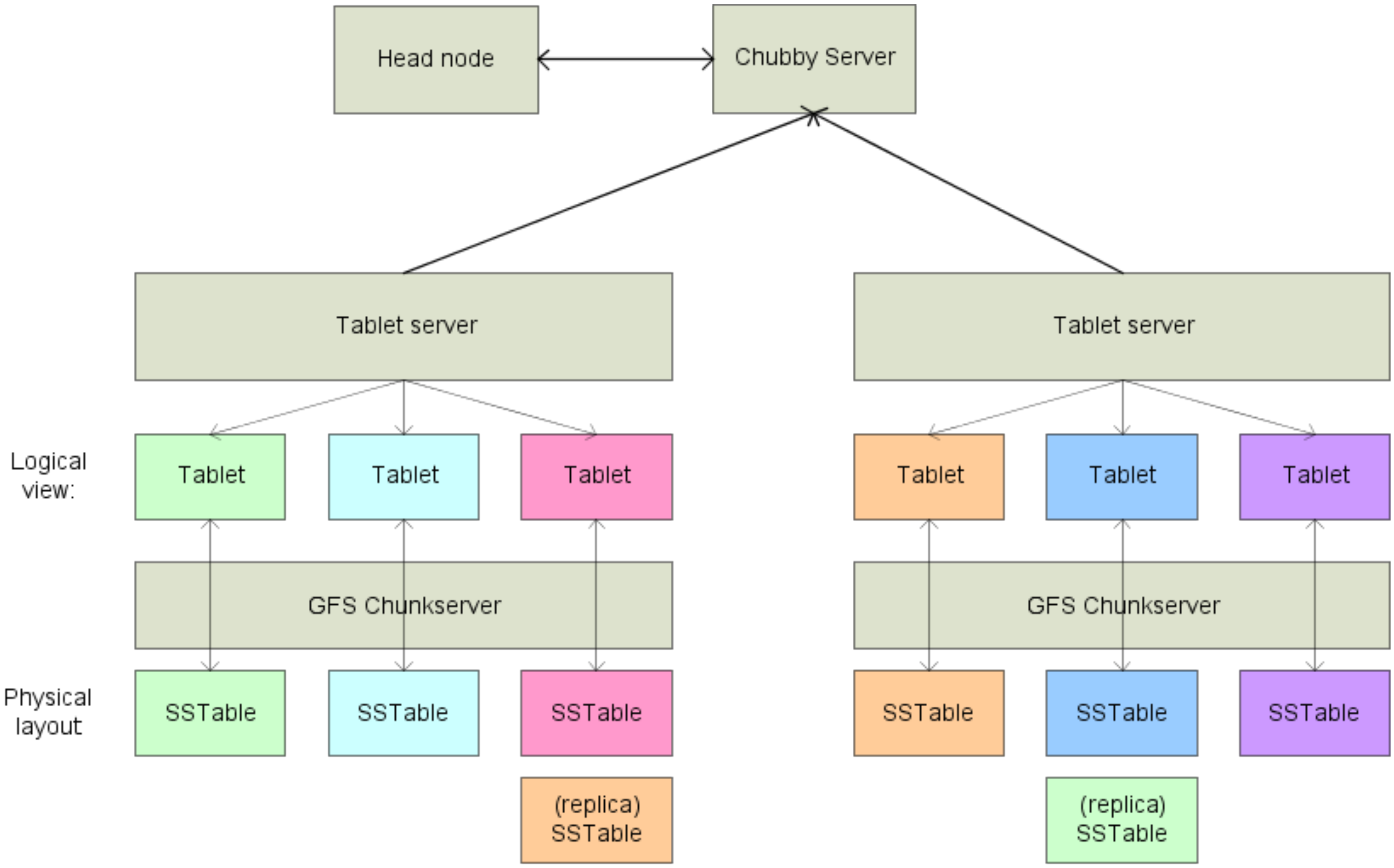
Bigtable Architecture



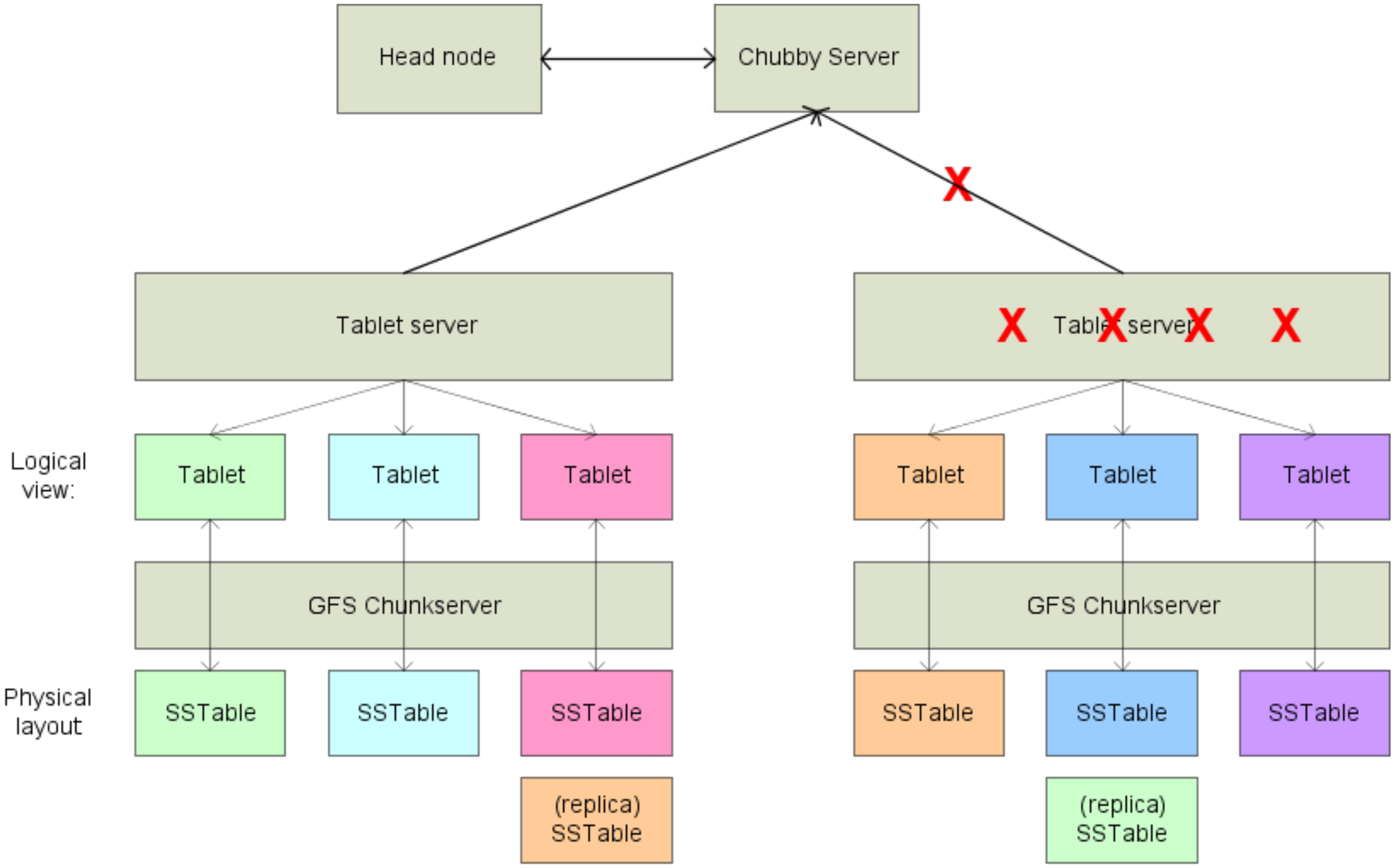
Master Responsibilities

- Determine which tablet server should hold a given (new) tablet
- Interface with GFS to garbage collect stale SSTable files
- Detect tablet server failures/resumption and load balance accordingly

Tablet Server Failure



Tablet Server Failure



Tablet Server Failure

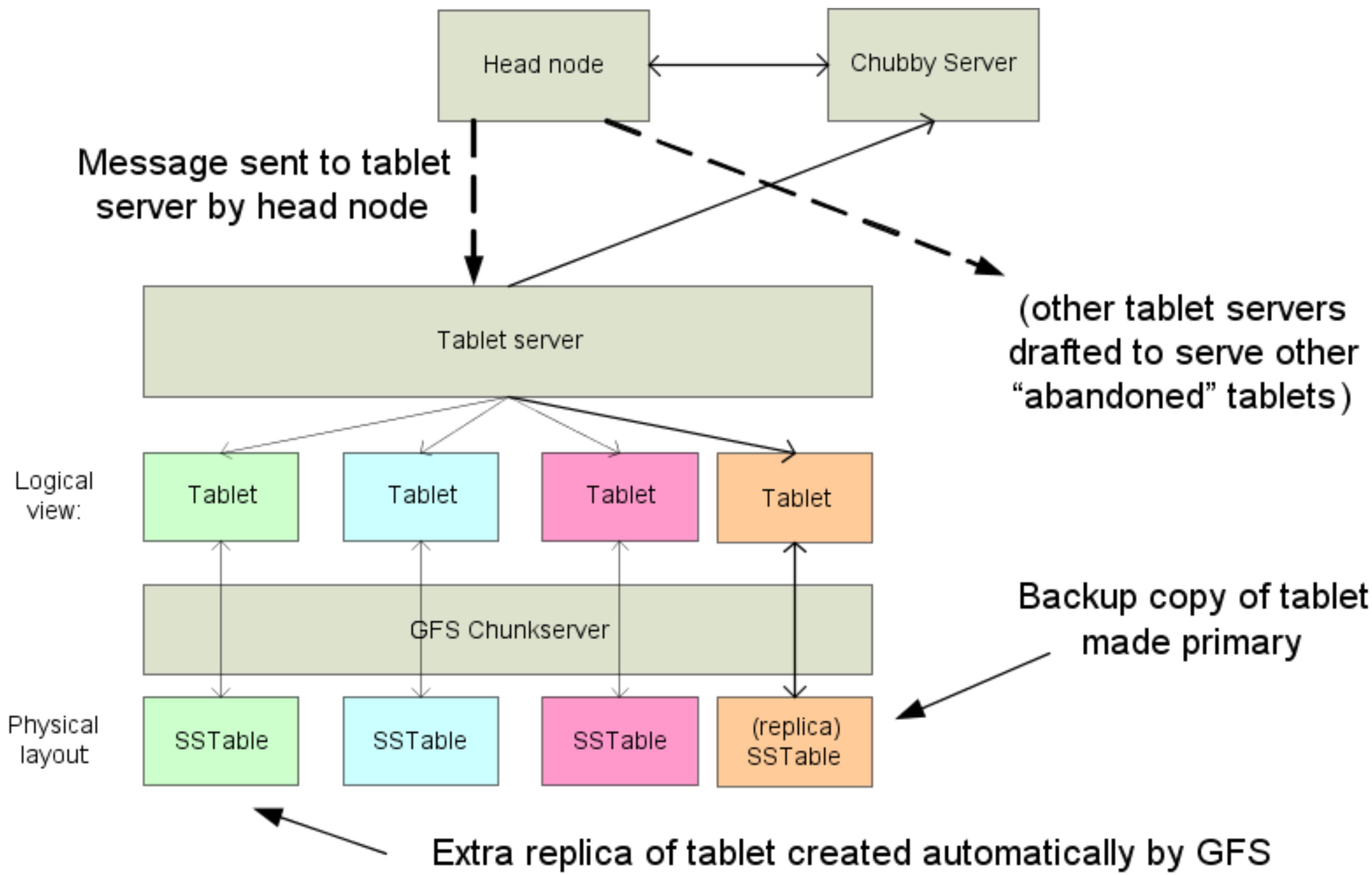
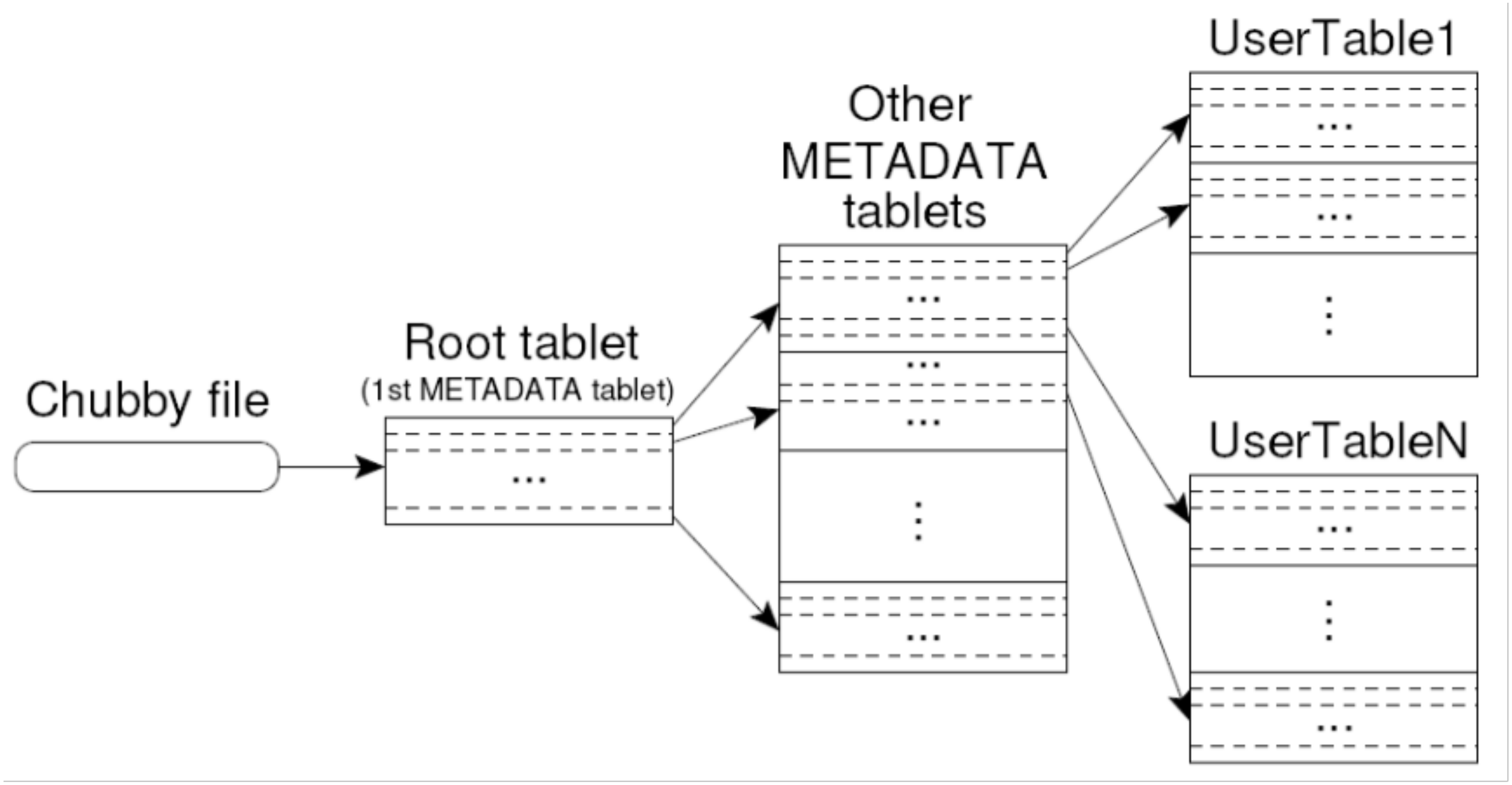


Table Access Structure



Write Procedure

- Writes to a tablet are recorded in a GFS-enabled commit log
- New data is then stored in memory on tablet server
    - supercedes underlying SSTable files

Minor Compactions

- Old data is stored in SSTable files
- Newer values are stored in memory in a *memtable*
- When a memtable exceeds a certain size, it is converted to an SSTable and written to disk
    - …Thus a tablet may be multiple SSTables underneath!

Merging Compactions

- Multiple SSTable files are now involved in a single lookup operation – slow!
- *Merging compactions* read multiple SSTables and create a new SSTable containing the most recent data
  - Old SSTable files are discarded
  - If only one SSTable remains for a tablet, called a *major compaction*

Commit Logs & Server Failure

- Diagram from earlier is not entirely accurate:
  - Contents of memtable are lost on tablet server failure
  - When a new tablet server takes over, it replays the commit log for the tablet first
  - Compactions discard unneeded commit log entries

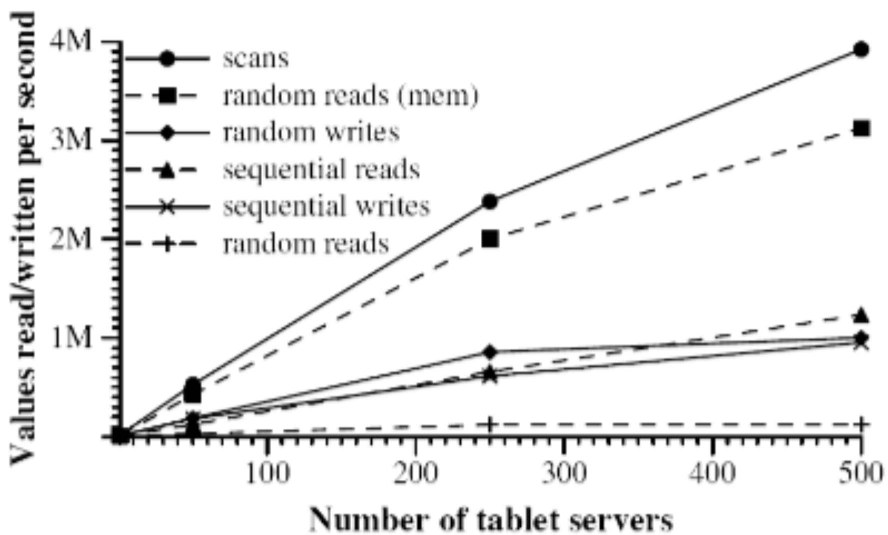
Further Optimizations

- Locality groups
  - Multiple column families can be declared as "related"; stored in same SSTable
  - Fast compression algorithms conserve space in SSTable by compressing related data
- Bloom filters
  - If multiple SSTables comprise a tablet, bloom filters allow quick discarding of irrelevant SSTables from a lookup operation

Performance

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Number of 1KB reads/writes per second, per server



Conclusions

- Simple data schemas work
    - Provided you design clients ground-up for this ahead of time
- Layered application building simplifies protocols & improves reliability
- Very high data transfer rates possible for simple data maps, lots of parallelism available

Chubby

What is it?

- A *coarse-grained lock service*
  - Other distributed systems can use this to synchronize access to shared resources
- Intended for use by “loosely-coupled distributed systems”

Design Goals

- High availability
- Reliability

- Anti-goals:
  - High performance
  - Throughput
  - Storage capacity

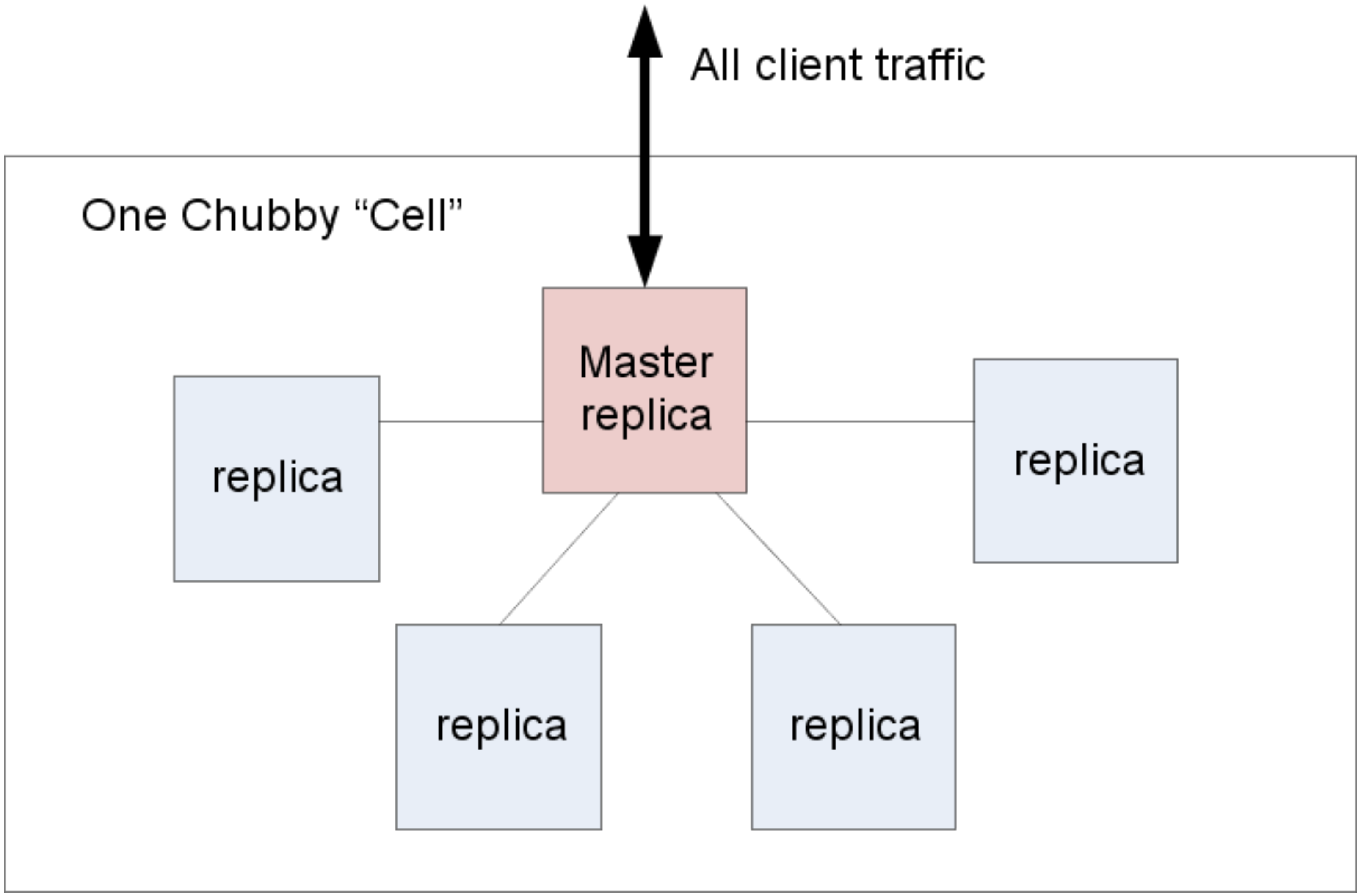
Intended Use Cases

- GFS: Elect a master
- BigTable: master election, client discovery, table service locking
- Well-known location to bootstrap larger systems
- Partition workloads
- Locks should be **coarse**: held for hours or days – build your own fast locks on top

External Interface

- Presents a simple distributed file system
- Clients can open/close/read/write files
    - Reads and writes are *whole-file*
    - Also supports *advisory* reader/writer locks
    - Clients can register for notification of file update

Topology


All client traffic
One Chubby "Cell"
Master replica
replica
replica
replica
replica

Master election

- Master election is simple: all replicas try to acquire a write lock on designated file. The one who gets the lock is the master.
  - Master can then write its address to file; other replicas can read this file to discover the chosen master name.
  - Chubby doubles as a *name service*

Distributed Consensus

- Chubby cell is usually 5 replicas
  - 3 must be alive for cell to be viable
- How do replicas in Chubby agree on their own master, official lock values?
  - PAXOS algorithm

PAXOS

- Paxos is a family of algorithms (by Leslie Lamport) designed to provide *distributed consensus* in a **network** of several **processors**.

Processor Assumptions

- Operate at arbitrary speed
- Independent, random failures
- Procs with stable storage may rejoin protocol after failure
- Do not lie, collude, or attempt to maliciously subvert the protocol

Network Assumptions

- All processors can communicate with ("see") one another
- Messages are sent asynchronously and may take arbitrarily long to deliver
- Order of messages is not guaranteed: they may be lost, reordered, or duplicated
- Messages, if delivered, are not corrupted in the process

A Fault Tolerant Memory of Facts

- Paxos provides a memory for individual “facts” in the network.
- A **fact** is a binding from a variable to a value.
- Paxos between 2F+1 processors is reliable and can make progress if up to F of them fail.

Roles

- Proposer – An agent that proposes a fact
- Leader – the authoritative proposer
- Acceptor – holds agreed-upon facts in its memory
- Learner – May retrieve a fact from the system

Safety Guarantees

- Nontriviality: Only *proposed* values can be learned
- Consistency: Only at most one value can be learned
- Liveness: If at least one value V has been proposed, eventually any learner L will get *some* value

Key Idea

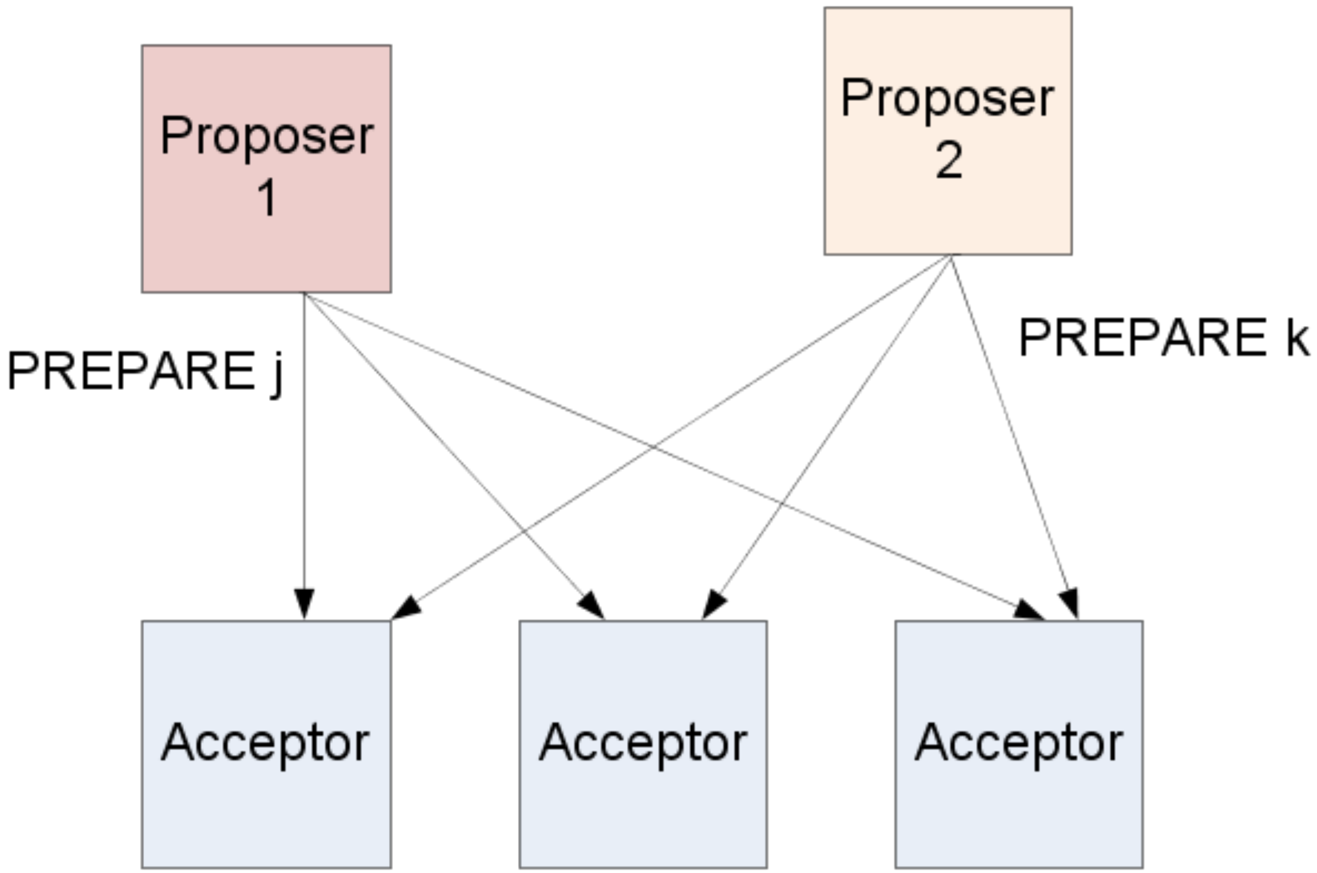
- Acceptors do not act unilaterally. For a fact to be learned, a **quorum** of acceptors must agree upon the fact
- A quorum is any majority of acceptors
- Given acceptors {A, B, C, D}, Q = {{A, B, C}, {A, B, D}, {B, C, D}, {A, C, D}}

Basic Paxos

- Determines the authoritative value for a single variable

- Several proposers offer a value V_n to set the variable to.
- The system converges on a single agreed-upon V to be the fact.

Step 1: Prepare

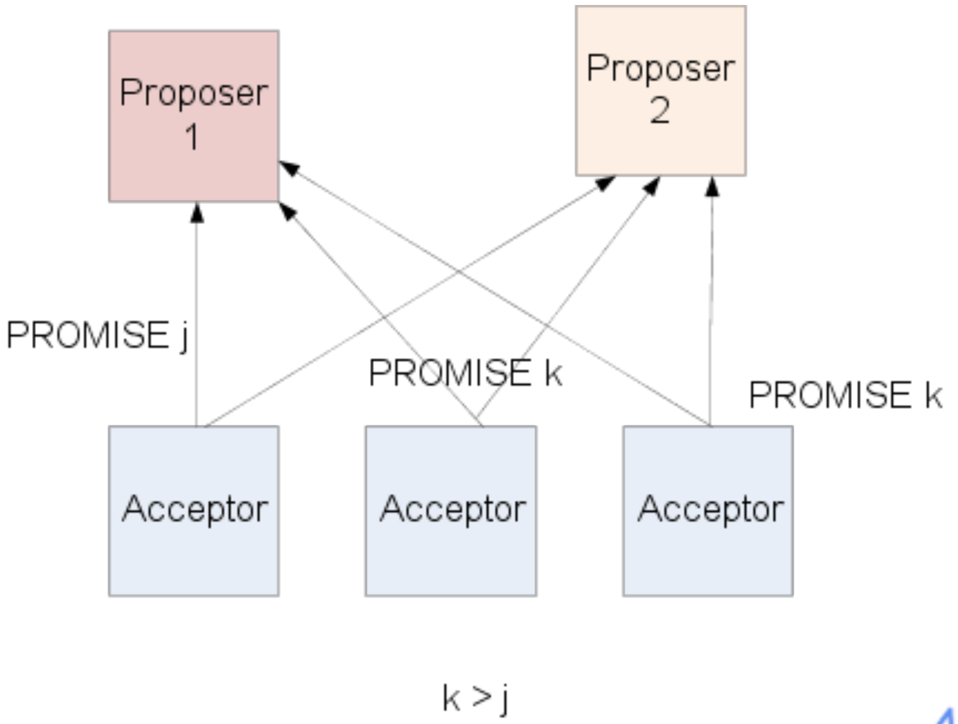


k > j

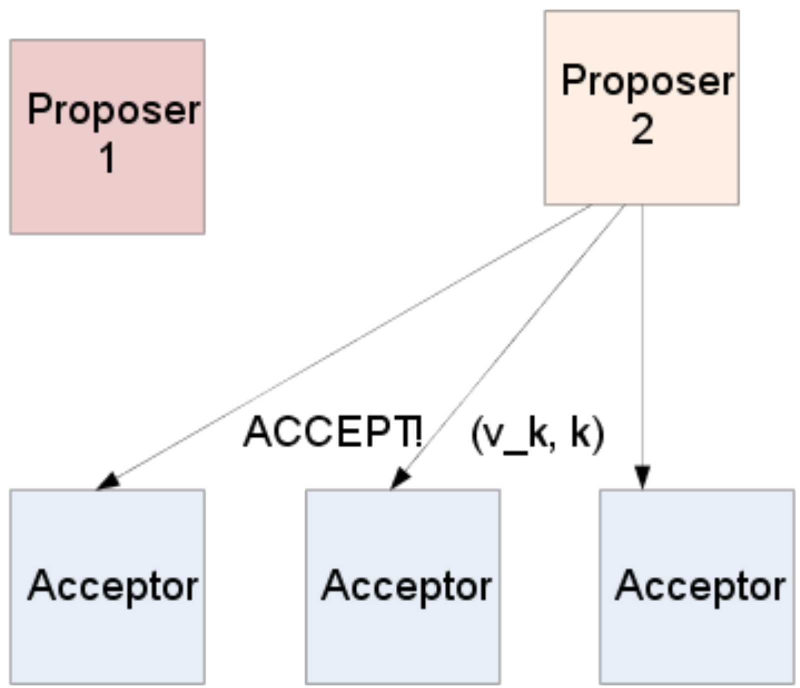
Step 2: Promise

- PROMISE x – Acceptor will accept proposals only numbered *x* or higher

- Proposer 1 is *ineligible* because a quorum has voted for a higher number than *j*

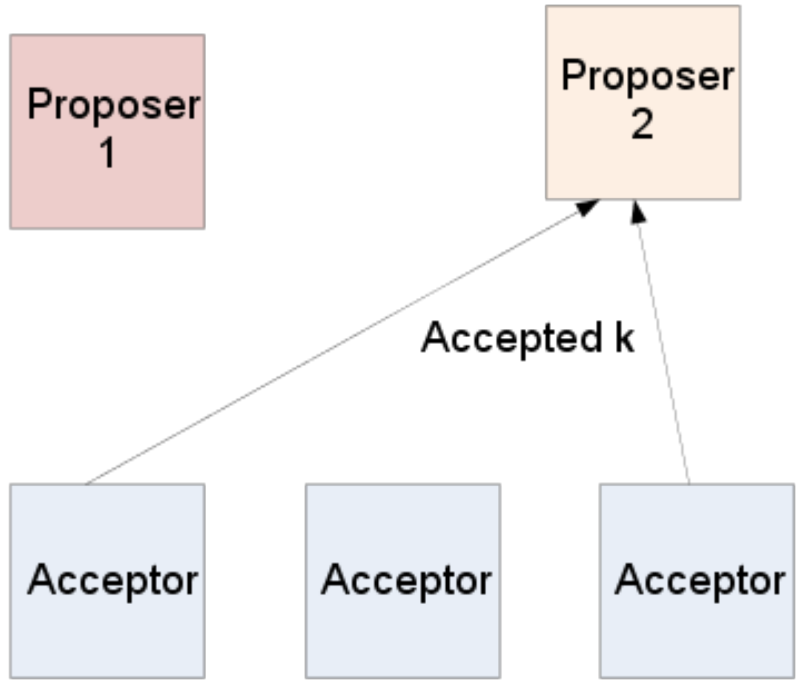


Step 3: Accept!



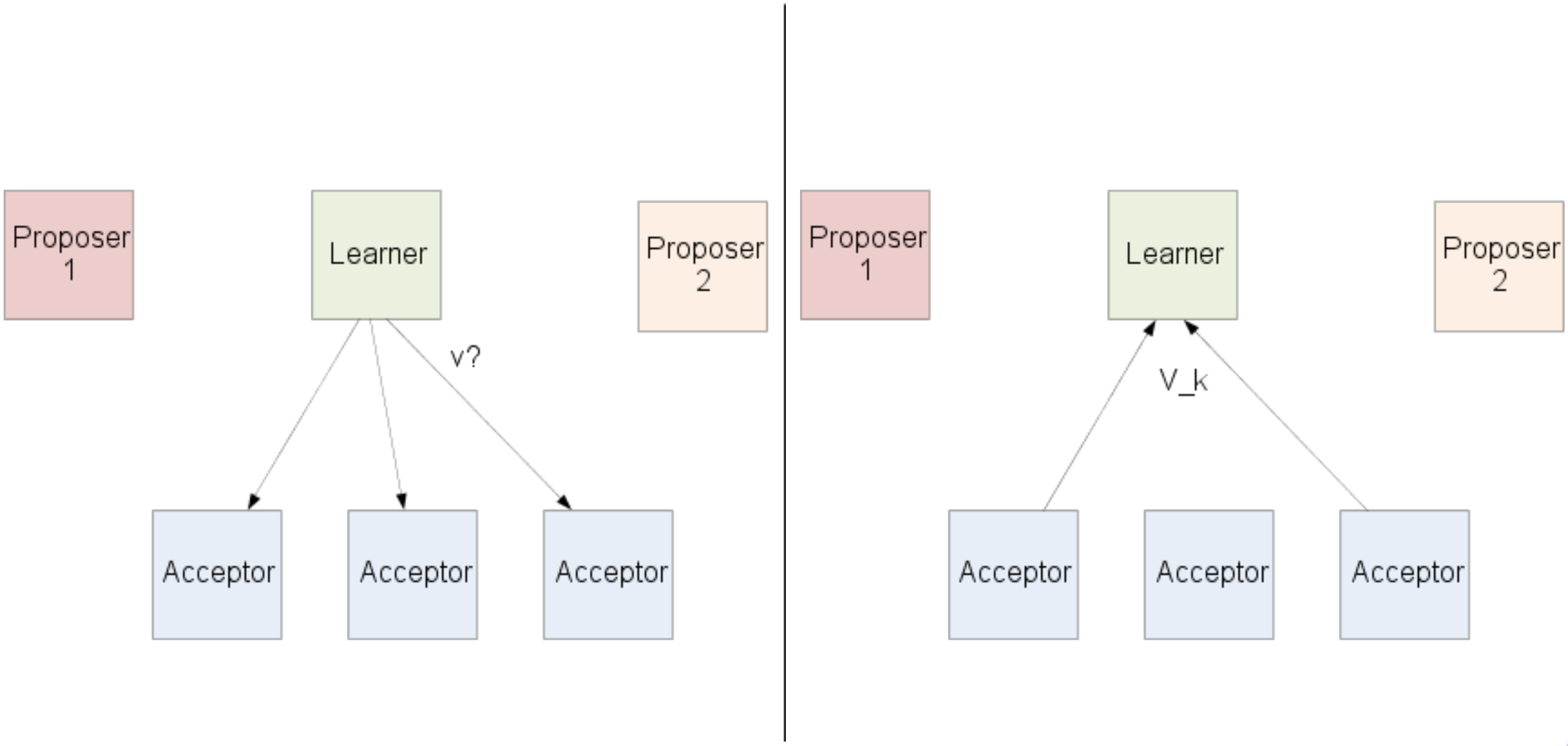
Proposer 1 is disqualified; Proposer 2 offers a value

Step 4: Accepted



A quorum has accepted value v_k; it is now a fact

Learning values



If a learner interrogates the system, a quorum will respond with fact V_k

Basic Paxos…

- Proposer 1 is free to try again with a proposal number > k; can take over leadership and write in a new authoritative value
  - Official fact will change "atomically" on all acceptors from perspective of learners
  - If a leader dies mid-negotiation, value just drops, another leader tries with higher proposal

More Paxos Algorithms

- Not whole story
- MultiPaxos: steps 1—2 done once, 3—4 repeated multiple times by same leader
- Also: cheap Paxos, fast Paxos, generalized Paxos, Byzantine Paxos…

Paxos in Chubby

- Replicas in a cell initially use Paxos to establish the leader.
- Majority of replicas must agree
- Replicas promise not to try to elect new master for at least a few seconds (“master lease”)
- Master lease is periodically renewed

Client Updates

- All client updates go through master
- Master updates official database; sends copy of update to replicas
    - Majority of replicas must acknowledge receipt of update before master writes its own value
- Clients find master through DNS
    - Contacting replica causes redirect to master

Chubby File System

- Looks like simple UNIX FS: /ls/foo/wombat
  - All filenames start with ‘/ls’ (“lockservice”)
  - Second component is cell (“foo”)
  - Rest of the path is anything you want
- No inter-directory move operation
- Permissions use ACLs, non-inherited
- No symlinks/hardlinks

Files

- Files have version numbers attached
- Opening a file receives handle to file
  - Clients cache all file data including file-not-found
  - Locks are *advisory* – not required to open file

Why Not Mandatory Locks?

- Locks represent client-controlled resources; how can Chubby enforce this?
- Mandatory locks imply shutting down client apps entirely to do debugging
  - Shutting down distributed applications much trickier than in single-machine case

Callbacks

- Master notifies clients if files modified, created, deleted, lock status changes
- Push-style notifications decrease bandwidth from constant polling

Cache Consistency

- Clients cache all file content
- Must send respond to Keep-Alive message from server at frequent interval
- KA messages include invalidation requests
  - Responding to KA implies acknowledgement of cache invalidation
- Modification only continues after all caches invalidated or KA time out

Client Sessions

- **Sessions** maintained between client and server
  - Keep-alive messages required to maintain session every few seconds
- If session is lost, server releases any client-held handles.
- What if master is late with next keep-alive?
  - Client has its own (longer) timeout to detect server failure

Master Failure

- If client does not hear back about keep-alive in *local lease timeout*, session is **in jeopardy**
  - Clear local cache
  - Wait for “grace period” (about 45 seconds)
  - Continue attempt to contact master
- Successful attempt => ok; jeopardy over
- Failed attempt => session assumed lost

Master Failure (2)

- If replicas lose contact with master, they wait for grace period (shorter: 4—6 secs)
- On timeout, hold new election

Reliability

- Started out using replicated Berkeley DB
- Now uses custom write-thru logging DB
- Entire database periodically sent to GFS
  - In a different data center
- Chubby replicas span multiple racks

Scalability

- 90K+ clients communicate with a single Chubby master (2 CPUs)
- System increases lease times from 12 sec up to 60 secs under heavy load
- Clients cache virtually everything
- Data is small – all held in RAM (as well as disk)

Conclusion

- Simple protocols win again
- Piggybacking data on Keep-alive is a simple, reliable coherency protocol