The Google File System

King.Zevin@ADSLab

Outline

- Motivation
- Assumptions
- Questions
- Measurements
- Benefits/Limitations
- Conclusion

Motivation

- Need for a scalable DFS
- Large distributed data-intensive applications
- High data processing needs
- Performance, Reliability, Scalability and Availability
- More than traditional DFS

Assumptions

- Assumptions – Environment
- Assumptions – Applications

Assumptions – Environment

- Commodity Hardware
  - inexpensive
- TBs of Space
  - must support TBs of space
- Component Failure
  - the norm rather than the exception

Assumptions – Applications

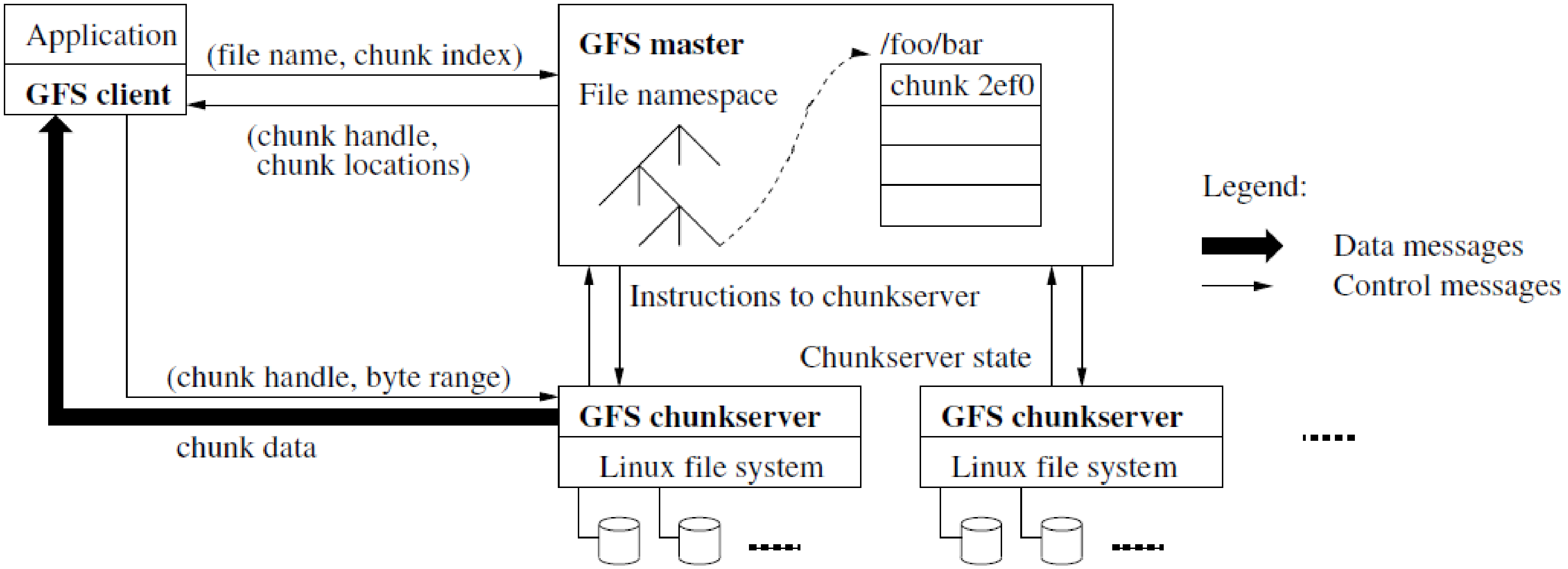
- **Multi-GB files**
  - Common
- **Workloads**
  - Large streaming reads
  - Small random reads
  - Large, sequential writes that append data to file
  - Multiple clients concurrently append to one file
- **High sustained bandwidth**
  - More important than low latency

Question 0 –
How to design – isolation

- **large data and small control infos**
  - Separate metadata from data

- **Then we get the initial architecture model**
  - Client – where user code runs
  - Master – metadata for the chunks
  - Chunkservers – large data

Architecture model



Question 0 –
How to design – isolation

- **large data and small control infos**
  - Separate metadata from data
- **Then we get the architecture model**
  - Client – where user code runs
  - Master – metadata for the chunks
  - Chunkservers – large data
- **Then some questions will be brought up**

More questions – single master

- **What if the single master fails?**
  - Single master is not single – we make replicas

- **Network bottleneck?**
  - Reduce the interaction with master
  - **Let the clients hold sth to interact with chunkservers more**
  - **Cache – clients cache the chunk infos**
  - **Let the chunkservers can interact more with each other**
  - **Primary and secondary**

- **Too much metadata to hold**
- **No disks! All in memory!**

Question 1 –
Too much metadata

- Too much metadata to hold and we need
- No disks! All in memory!
- Traditional block is too small
- Let the chunk size be 64MB

Benefits – chunk size : 64MB

- Larger chunk, less metadata
  - Master maintains less than 64B of metadata for 64MB chunk
  - 1 TB chunks : < 1 MB metadata
  - 1 PB chunks : < 1 GB metadata
- Less interactions between client and master
- One continuous TCP connection during one chunk

- One disadvantage
  - little files with one chunk
    might make the chunkserver hotspot

So we get "chunk"

- Files are divided into *chunks*
- Replicated over *chunkservers,* called *replicas*
- Unique 64-bit *chunk handles*
- Chunks as Linux files

Question 2 –
What to hold in master

- **Metadata :**
  - The file and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk's replicas
- **All in memory**
- **first two kept persistent by operation log**
- **Get locations when startup and with heartbeat**
  - **Change too often**

Question 2.1 – Why operation log?

- We need reliability and ability to recovery
- Master might need restart, might fail
- Critical historical logs must be kept
  - In disks
  - By multiple servers
- In addition to the logs, we need to
- Checkpoint
- When checkpointing, create a new operation log and start a new checkpointing thread

Question 3 – What master does

- Chunk creation
- Re-replication
- Garbage collection

Question 3 – What master does

- **Chunk creation**
  - Low disk usage
  - Limit recent creations
  - Different racks
- **Re-replication**
- **Garbage collection**

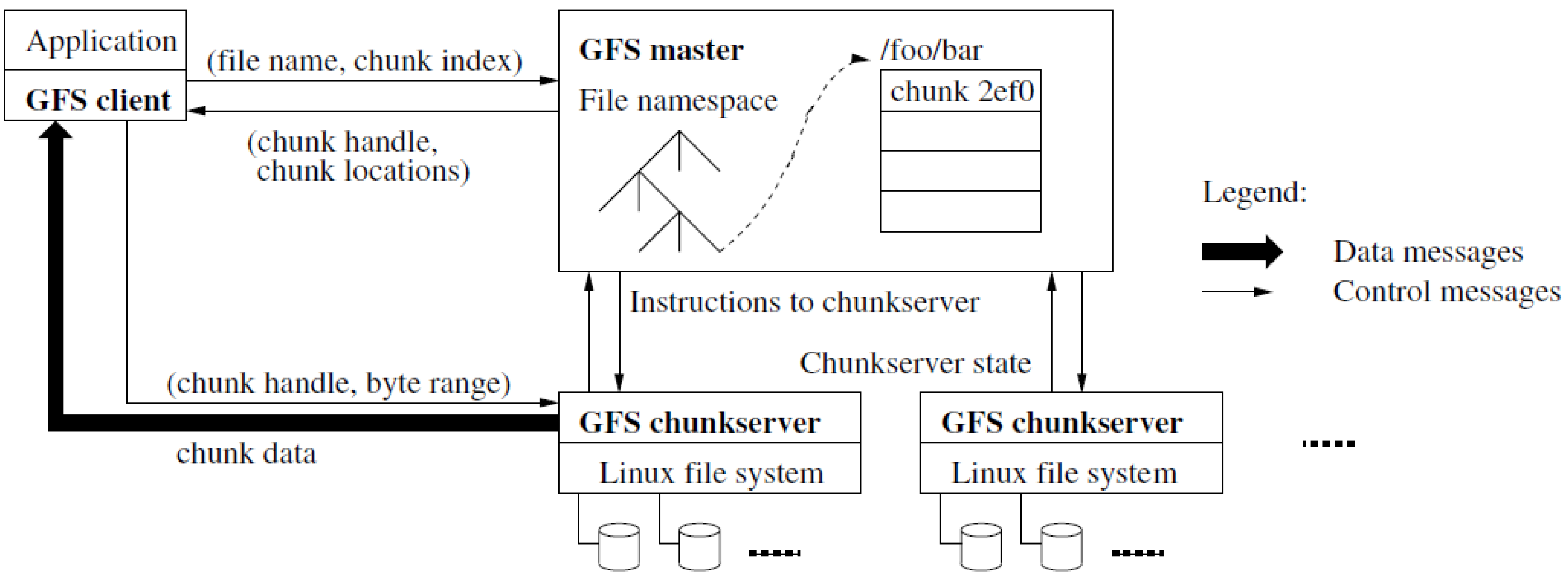
Question 3 –
What master does

- **Chunk creation**
- **Re-replication**
  - Priority
  - 2 failure > 1 failure
  - Live > deleted
  - Blocking > not blocking
- **Garbage collection**

Question 3 – What master does

- **Chunk creation**
- **Re-replication**
- **Garbage collection**
  - Does not delete immediately unless deleted twice
  - Why?
  - Simple and amortized(background activity, done only when free)
  - How?
  - After deleted, file is renamed to a hidden name
  - Deleted after 3 days
  - Can be undeleted and read

Architecture model



Question 4 – Namespace Management and Locking

- We need concurrency, so we need locking

- No per-directory data structure.
- No hard or symbolic links

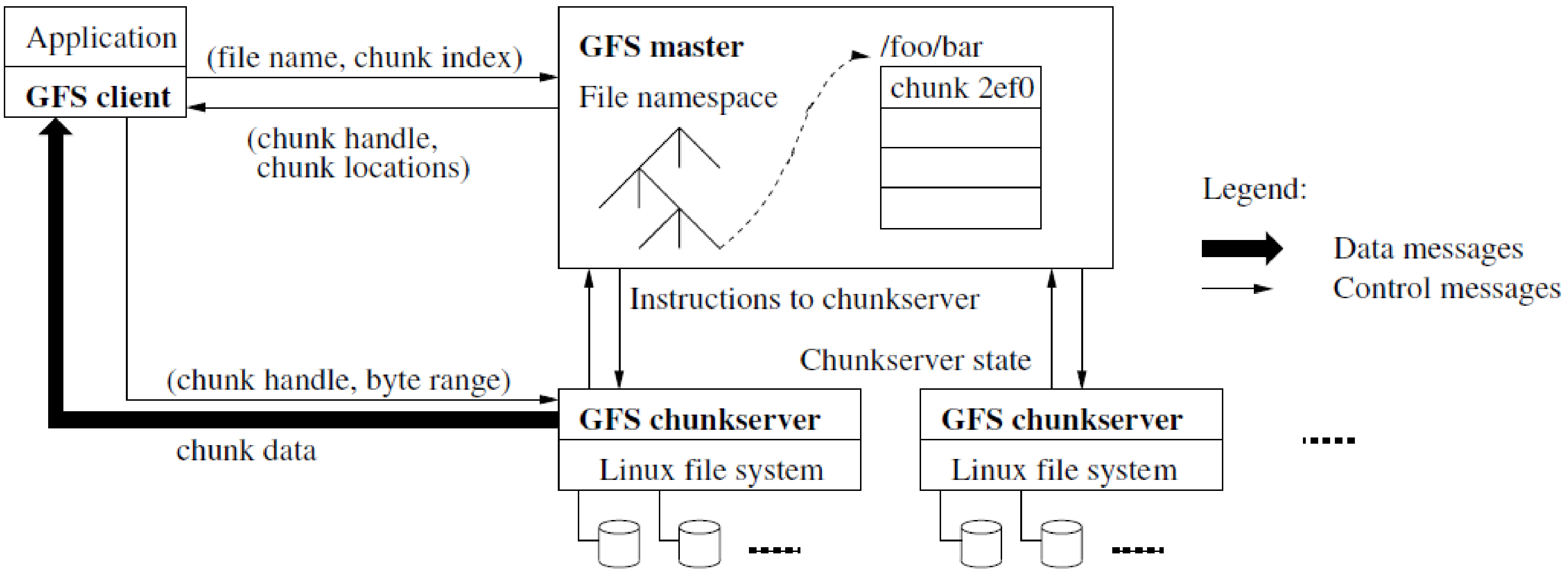
- Namespaces are represented as a lookup table mapping full pathnames to metadata
  - Prefix compression

- Each master operation acquires a set of locks before it runs

Question 4 – Example of Locking Mechanism

- Preventing /home/user/foo from being created while /home/user is being snapshotted to /save/user
  - Snapshot operation
    - Read locks on /home and /save
    - Write locks on /home/user and /save/user
  - File creation
    - read locks on /home and /home/user
    - write locks on /home/user/foo
  - Conflict locks on /home/user

Architecture model



Question 5 –
How to minimize master interactions

- Master – too much involvement!

- Between master and client, we use cache.

- Then what about master and chunkservers?
- Then give more power to chunkservers!

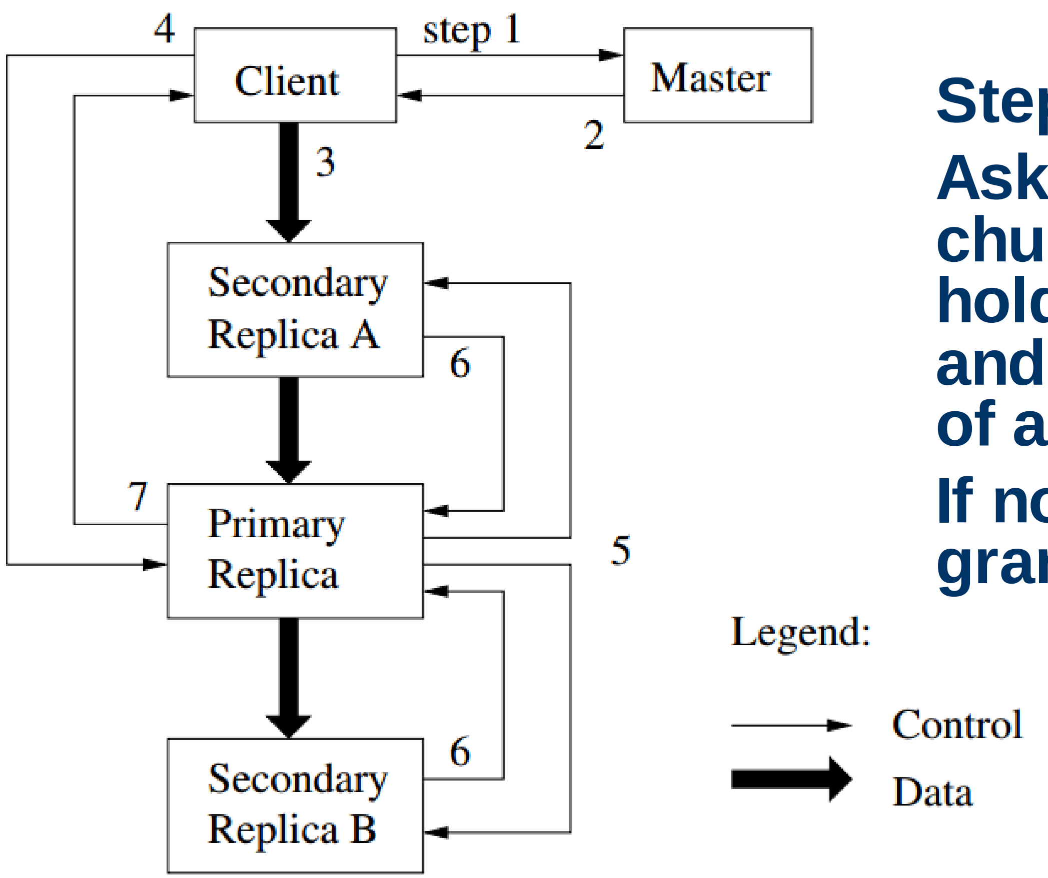
- Lease!

Question 5 –
Lease: to minimize master interactions

- Master grants a lease to a chunk – primary chunk
- Primary chunk picks the mutation order, which secondary chunks will follow.
- A lease is about 60s, extension requests can be piggybacked on HeartBeat message.

- Then we get the more detailed picture for the architecture model.

Question 6 – How the components interact

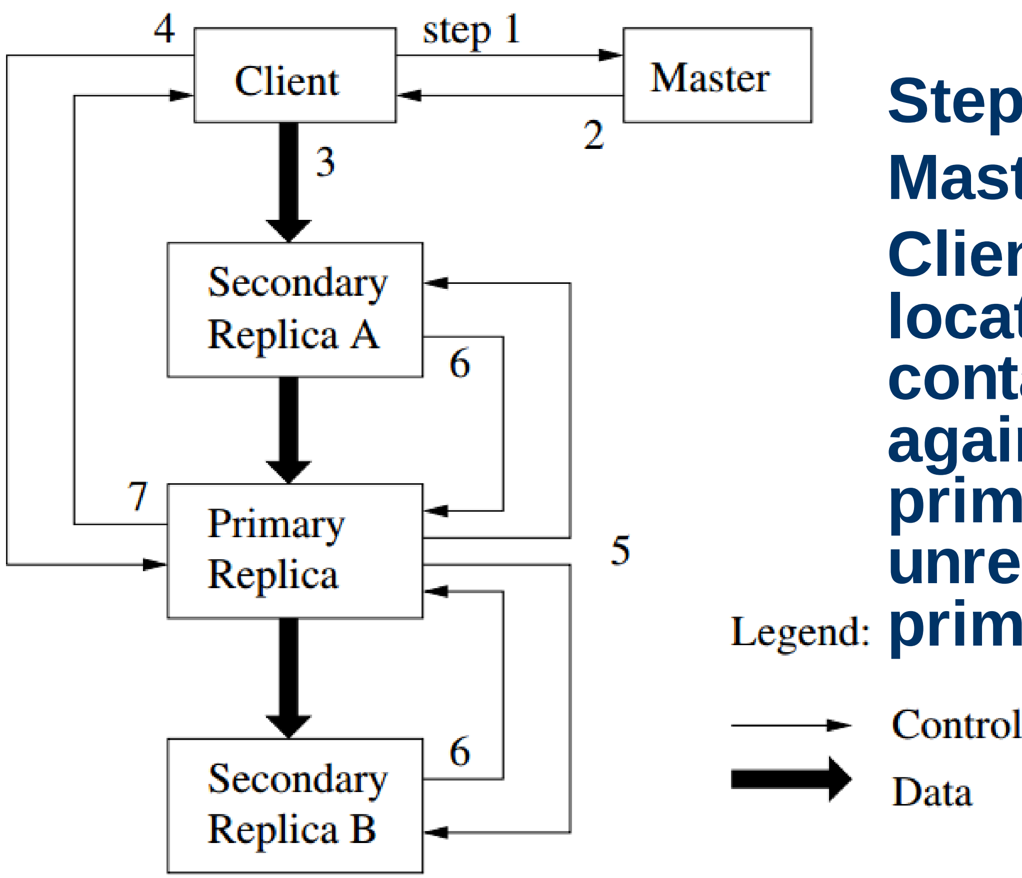


Step1:

Asks which chunkserver holds lease and locations of all replicas.

If no lease, grant one.

Question 6 –
How the components interact

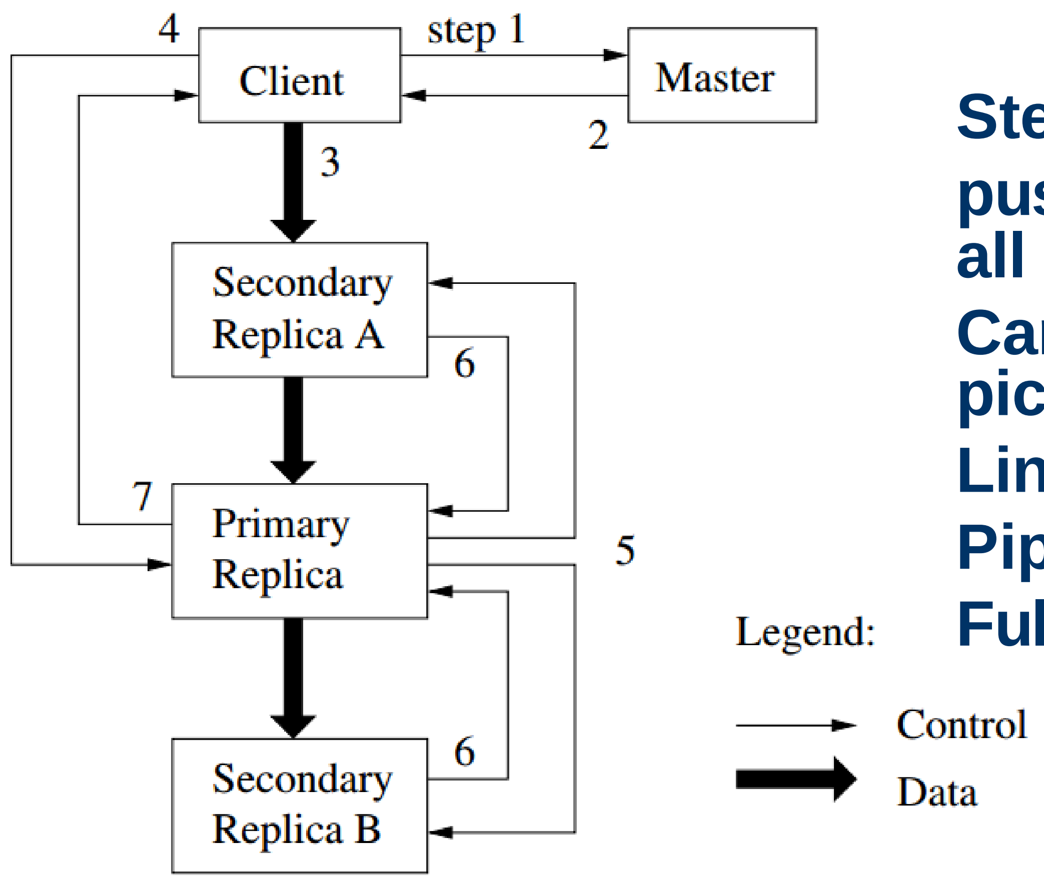


Step2:

Master replies.

Client caches locations, contact master again only when prim unreachable or prim is not prim.

Question 6 –
How the components interact



Step3:

push data to all replicas.

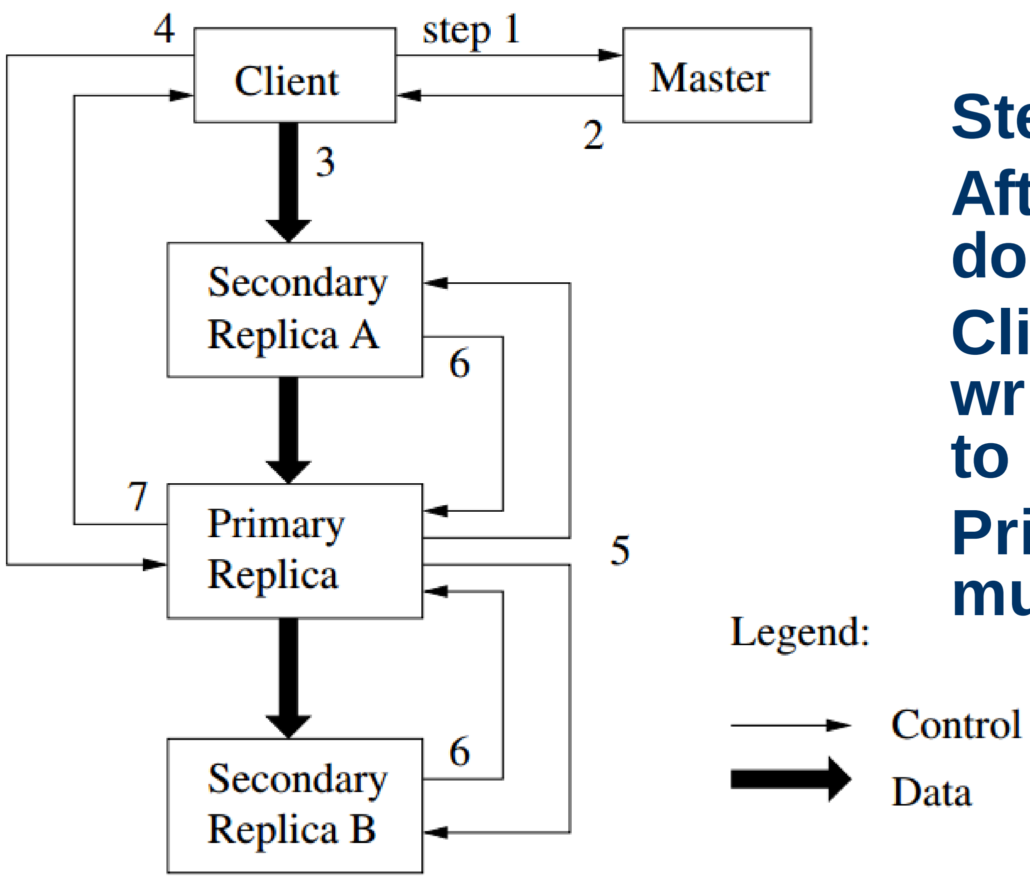
Carefully picked chain.

Linearly,

Pipelining.

Full duplex

Question 6 –
How the components interact



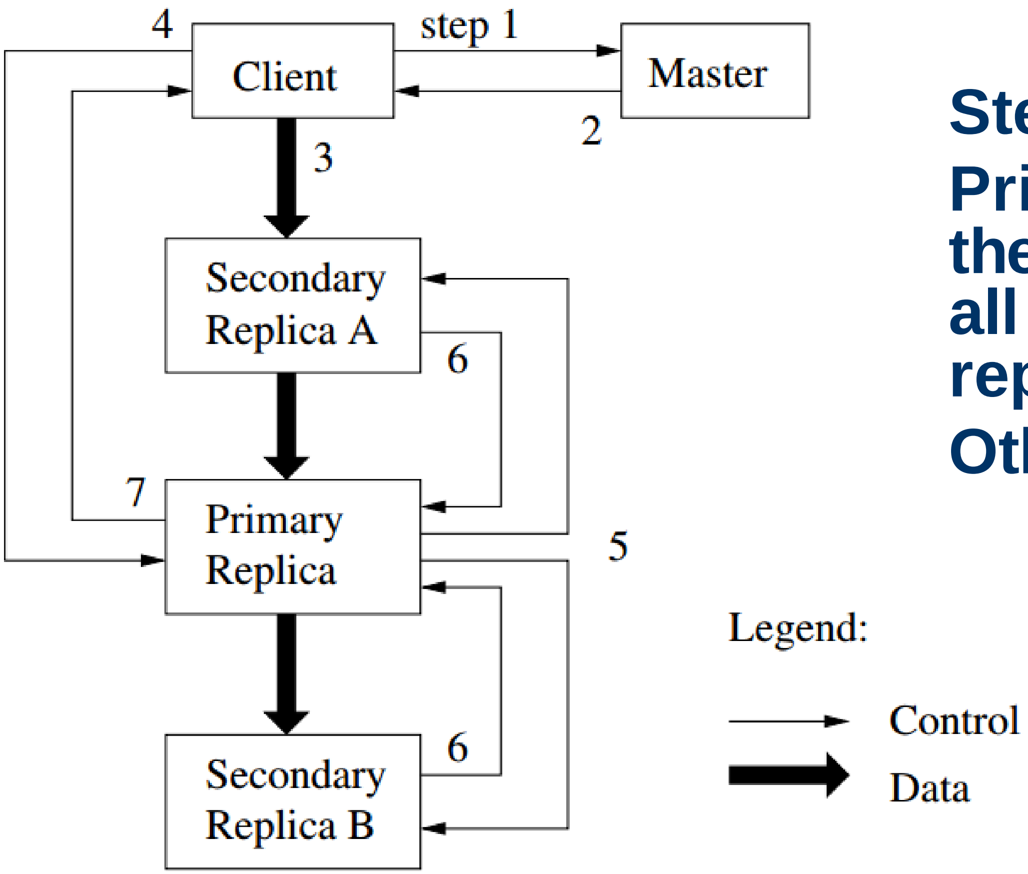
Step4:

After 3 totally done.

Client sends write request to prim.

Prim applies mutations.

Question 6 –
How the components interact

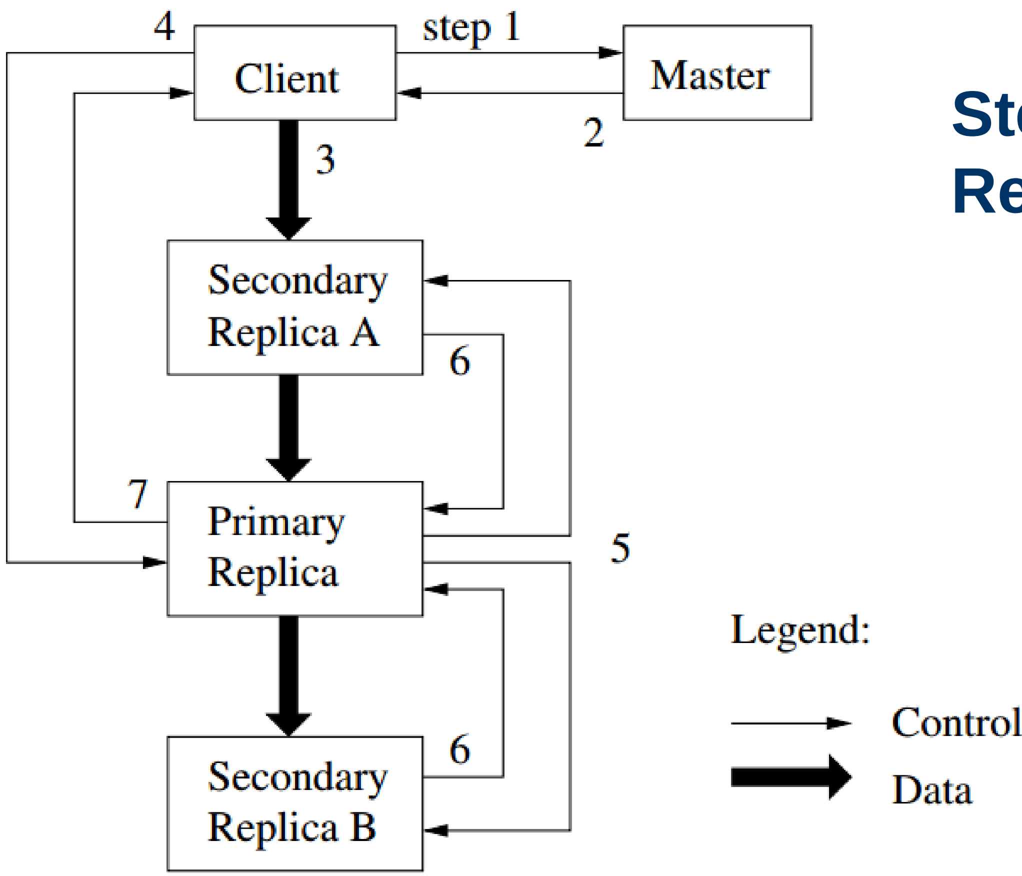


Step5:

Prim forward the request to all other replicas.

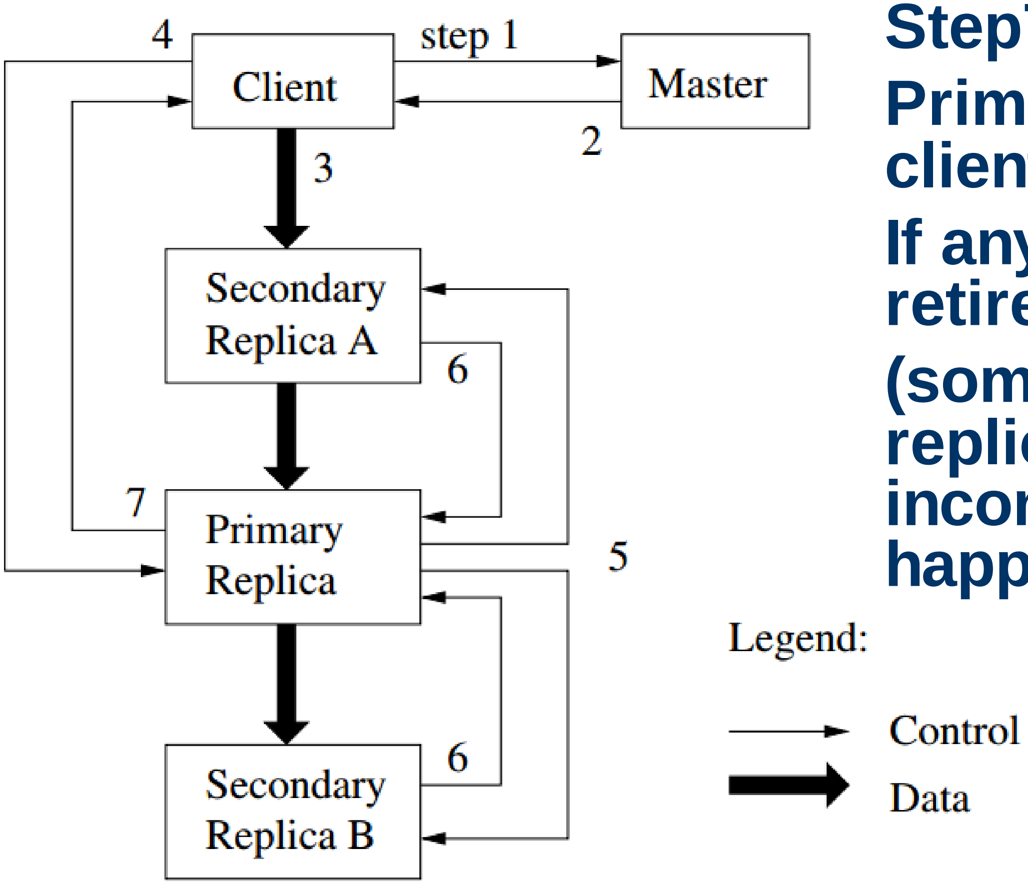
Others apply.

Question 6 –
How the components interact



Step6:
Reply to prim

Question 6 –
How the components interact



Step7:

Prim replies to client.

If any error, client retires.

(some secondary replica might fail, inconsistency happens.)

Some detailed questions to discuss

Question 7 –
How is the consistency model?

- **Relaxed consistency model**
- Two types of mutations
  - **Writes**
    - Cause data to be written at an application-specified file offset
  - **Record appends**
    - Operations that append data to a file
    - Cause data to be appended atomically at least once
    - Offset chosen by GFS, not by the client

<table>
<tr><th></th><th>Write</th><th>Record Append</th></tr>
<tr><td>Serial success</td><td>defined</td><td rowspan="2">defined interspersed with inconsistent</td></tr>
<tr><td>Concurrent successes</td><td>consistent but undefined</td></tr>
<tr><td>Failure</td><td colspan="2">inconsistent</td></tr>
</table>

Question 7.1 –
How to append records atomically?

- **Be similar to the preceding graph**
- But checks if the chunk will exceed 64MB
  - If so, it pads and let the client to append again to a new chunk
  - Limited within ¼ size once
- **What if one secondary replica fails?**
  - Client retries. Who failed pads and begin at the same offset.
  - As a result, replicas of the same chunk may contain different data, possibly including duplicates of the same record.
  - But GFS guarantees the data is written at least once and at the same offset.

Question 7.2 –
Then how to explain the Table 1?

- States of a file region after a mutation
  - *Consistent*
    - All clients see the same data, regardless which replicas they read from
  - *Defined*
    - *consistent* + all clients see what the mutation writes in its entirety
  - *Undefined*
    - *consistent* +but it may not reflect what any one mutation has written
  - *Inconsistent*
    - Clients see different data at different times

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Question 8 –
What about snapshot?

- Goals
  - To quickly create branch copies of huge data sets
  - Or to easily checkpoint the current state

- First revoke leases on chunks involved
- Then write operation will ask master and master will

- Copy-on-write
- locally

Question 9 – How to know which replica is stale

- **Version**!

- Replica version updated when:
  - Master grant a new lease
  - Each mutation is applied

Question 10 – Fault Tolerance and Diagnosis

- Fast Recovery
  - Operation log
  - Checkpointing
- Chunk replication
  - Each chunk is replicated on multiple chunkservers on different racks
- Master replication
  - Operation log and check points are replicated on multiple machines
- Data integrity
  - Checksumming to detect corruption of stored data
  - Each chunkserver independently verifies the integrity
- Diagnostic logs
  - Chunkservers going up and down
  - RPC requests and replies

Background – for measurements

- Two clusters within Google
  - Cluster A: Research & Development
    - Read and analyze data, write result back to cluster
    - Much human interaction
    - Short tasks
  - Cluster B: Production data processing
    - Long tasks with multi-TB data
    - Seldom human interaction

Measurements

- Read rates much higher than write rates
- Both clusters in heavy read activity
- Cluster A supports up to 750MB/read, B: 1300 MB/s
- Master was not a bottle neck

Cluster	**A**	**B**
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Measurements

- Recovery time (one chunkserver killed)
  - 15,000 chunks containing 600GB are restored in 23.2 minutes (replication rate ≅ 400MB/s)
- Recovery time (two chunkserver killed)
  - Each containing 16,000 chunks and 660GB
  - 266 chunks – single replica
  - Higher priority – 2 minutes
  - (Total time not told)

Benefits and Limitations

- Simple design with single master
- Fault tolerance
- Custom designed
- Only viable in a specific environment

Conclusion

- Different than previous file systems
- Satisfies needs of the application
- Fault tolerance

Thanks!

- Welcome to ask and discuss!