Google File System

CS 202

From paper by Ghemawat, Gobioff & Leung

UCR
UNIVERSITY OF CALIFORNIA, RIVERSIDE

The Need

- Component failures normal
  - Due to clustered computing
- Files are huge
  - By traditional standards (many TB)
- Most mutations are mutations
  - Not random access overwrite
- Co-Designing apps & file system

- Typical: 1000 nodes & 300 TB

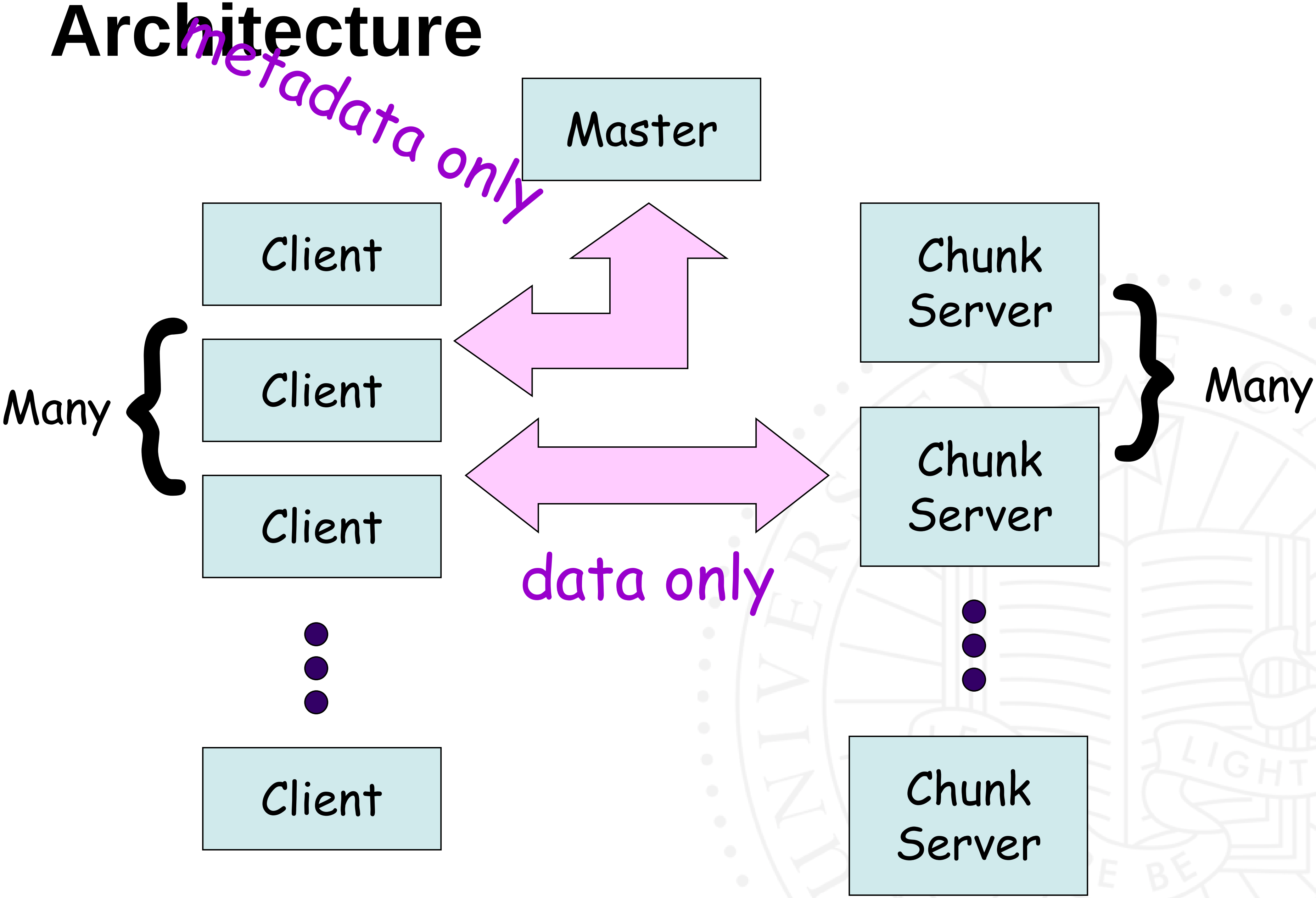
Desiderata

- Must monitor & recover from comp failures
- Modest number of large files
- Workload
  - Large streaming reads + small random reads
  - Many large sequential writes
    - Random access overwrites don’t need to be efficient
- Need semantics for concurrent appends
- High sustained bandwidth
  - More important than low latency

Interface

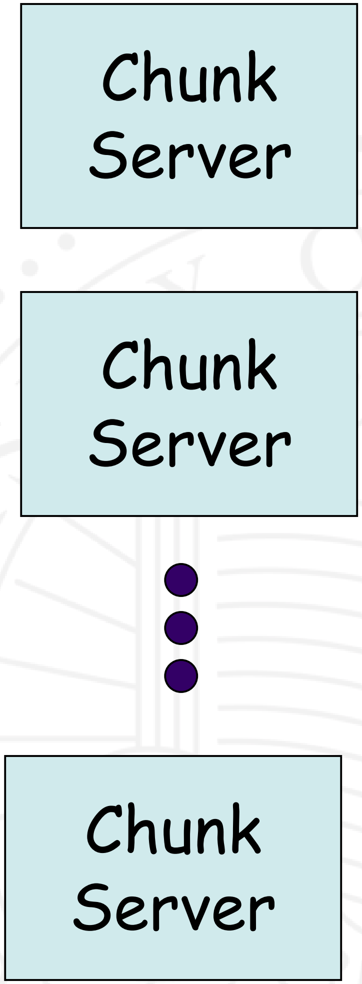
- Familiar
  - Create, delete, open, close, read, write
- Novel
  - Snapshot
    - Low cost
  - Record append
    - Atomicity with multiple concurrent writes

Architecture



Architecture

- Store all files
  - In fixed-size chucks
    - 64 MB
    - 64 bit unique handle
- Triple redundancy

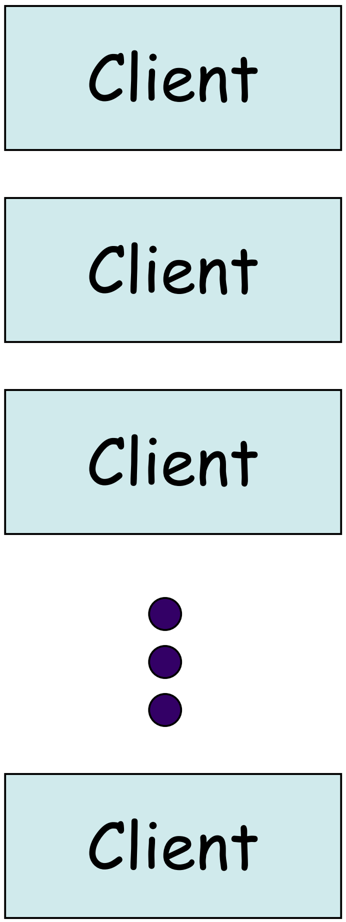


Architecture

Master

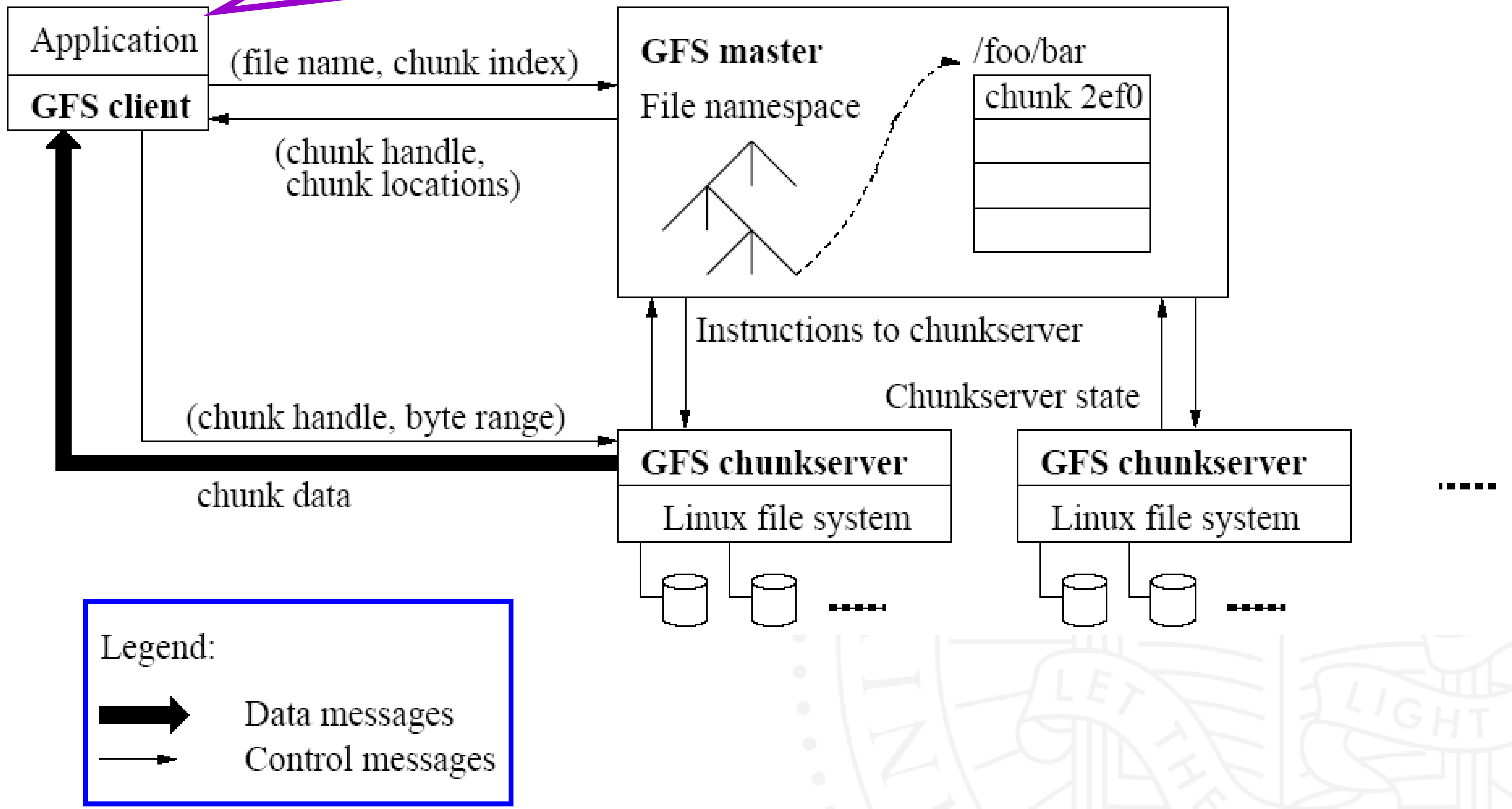
- Stores all metadata
  - Namespace
  - Access-control information
  - Chunk locations
  - 'Lease' management
- Heartbeats
- Having one master → global knowledge
  - Allows better placement / replication
  - Simplifies design

Architecture

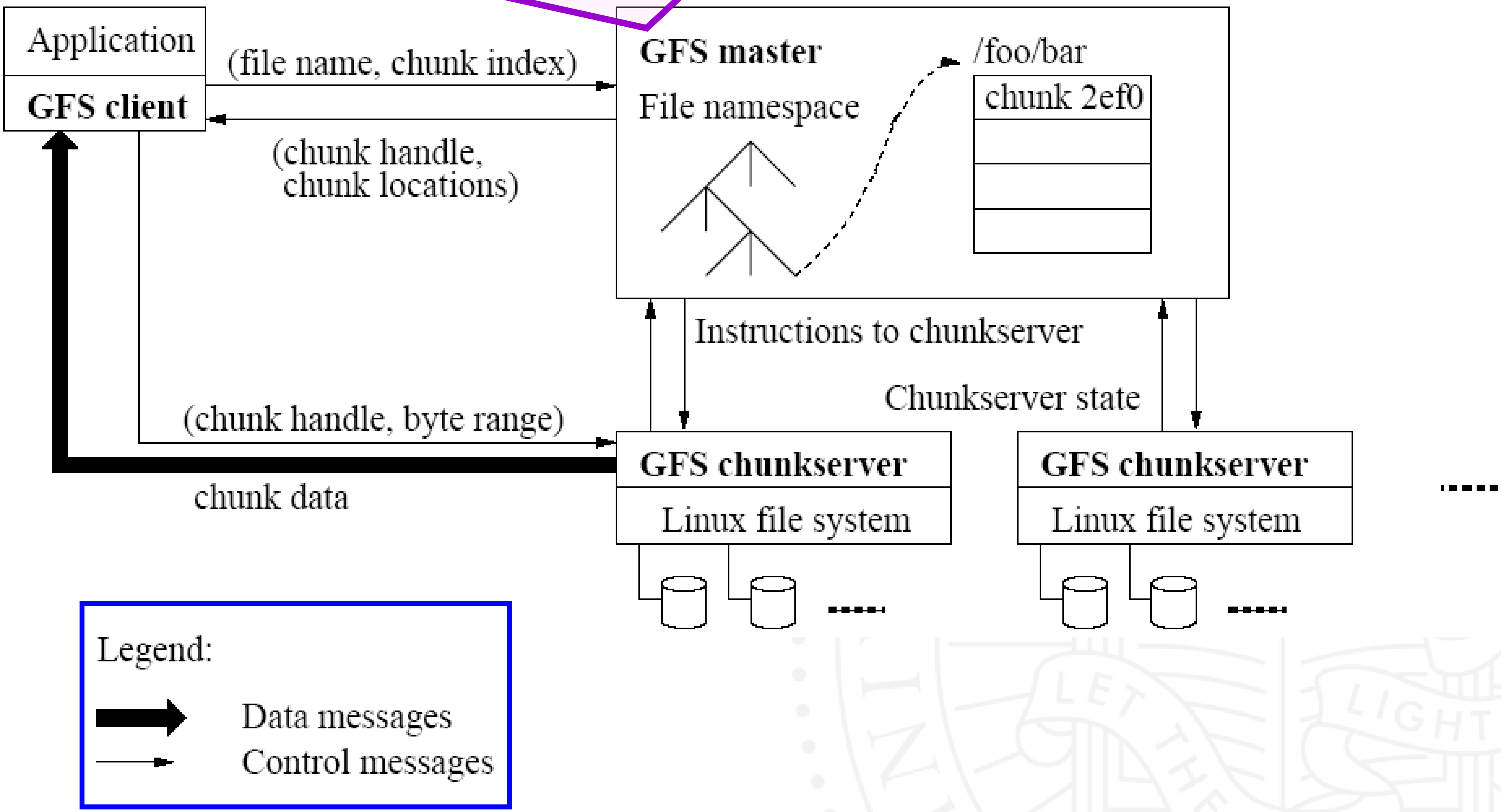


- GFS code implements API
- Cache only metadata

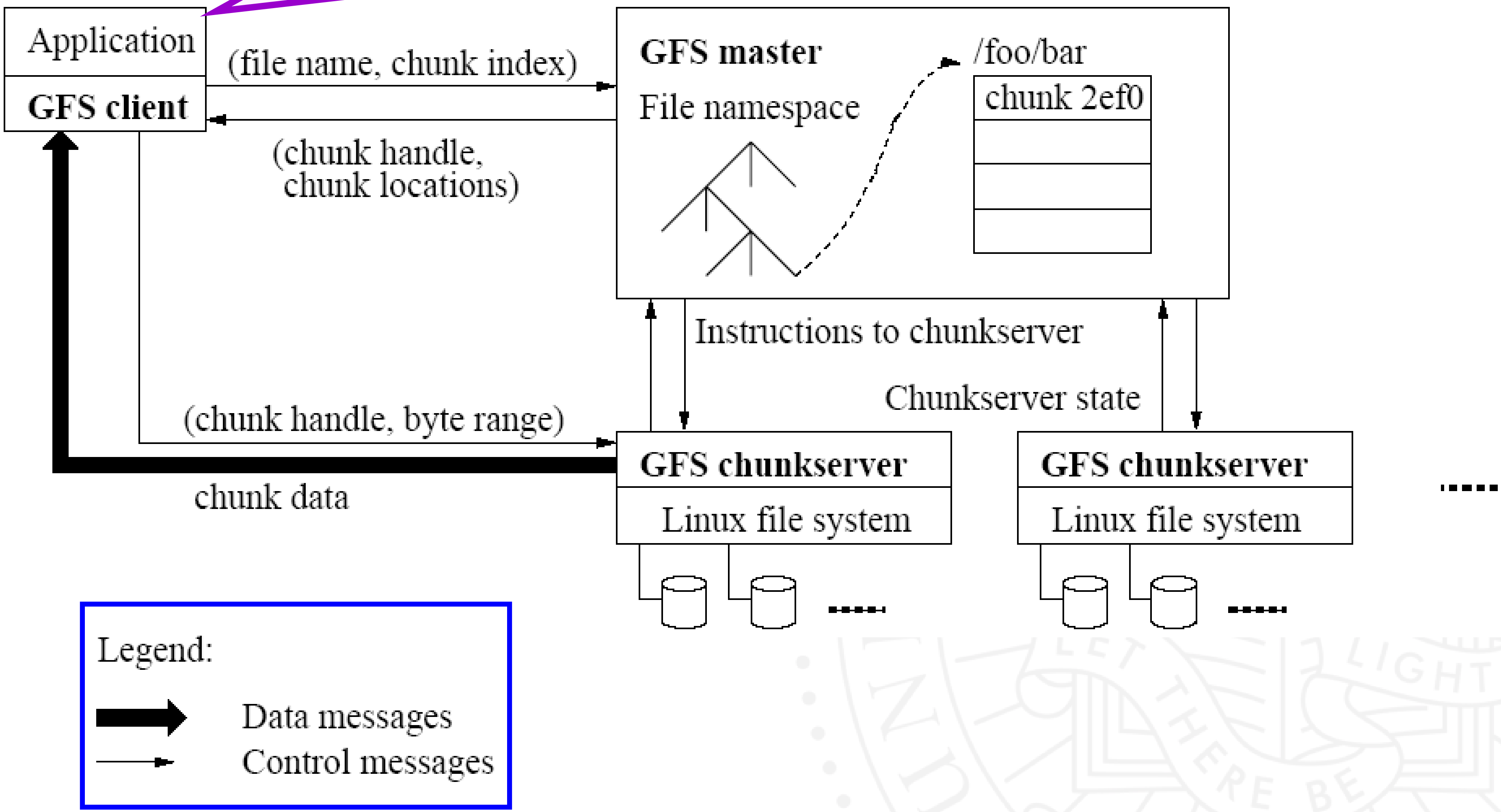
Using fixed chunk size, translate filename & byte offset to chunk index.
Send request to master

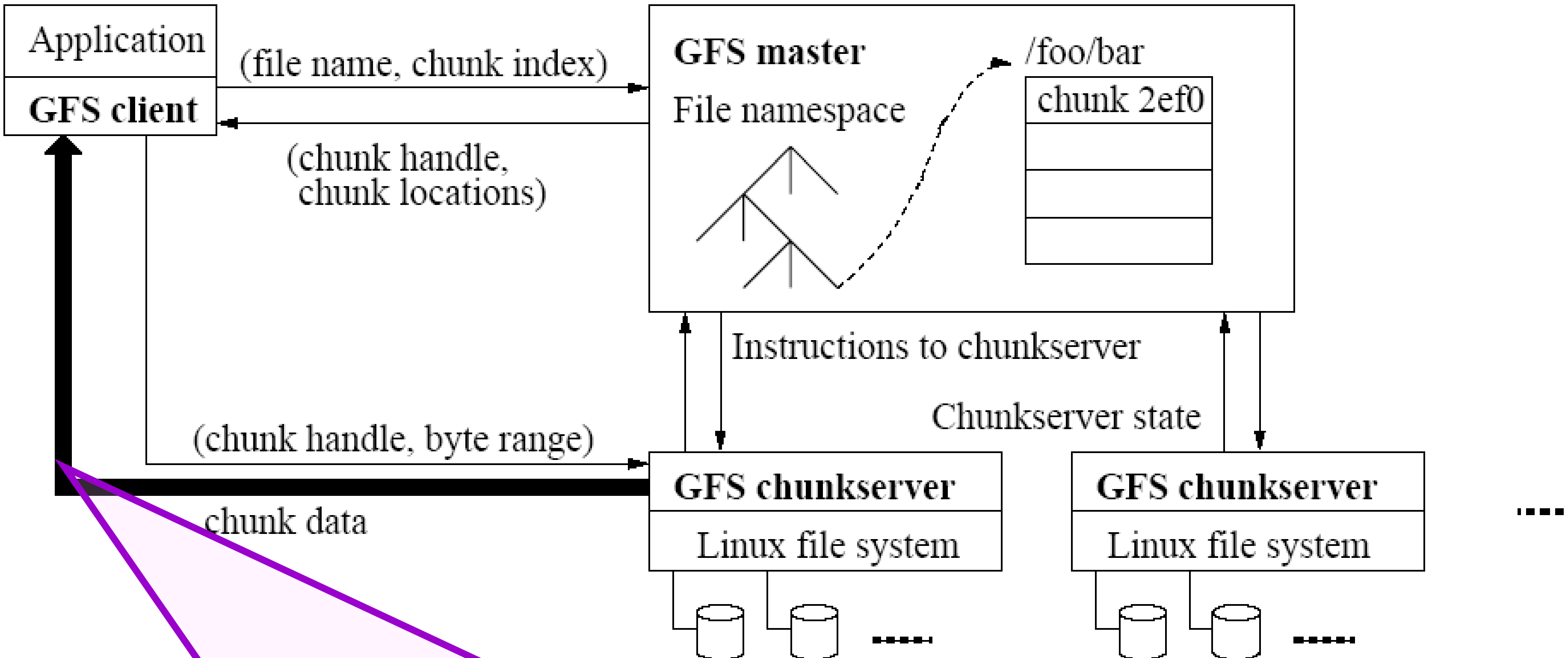


Replies with chunk handle & location of chunkserver replicas (including which is 'primary')



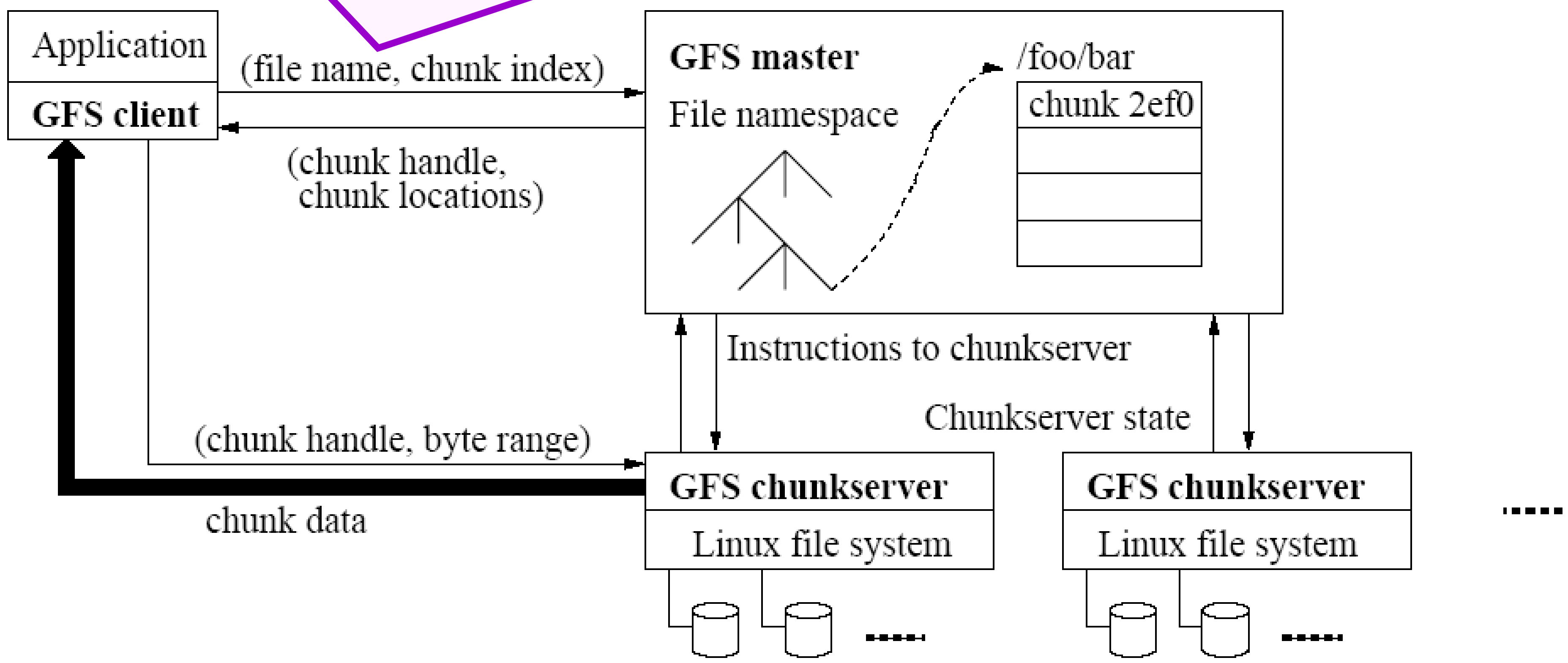
Cache info
using filename & chunk index as key



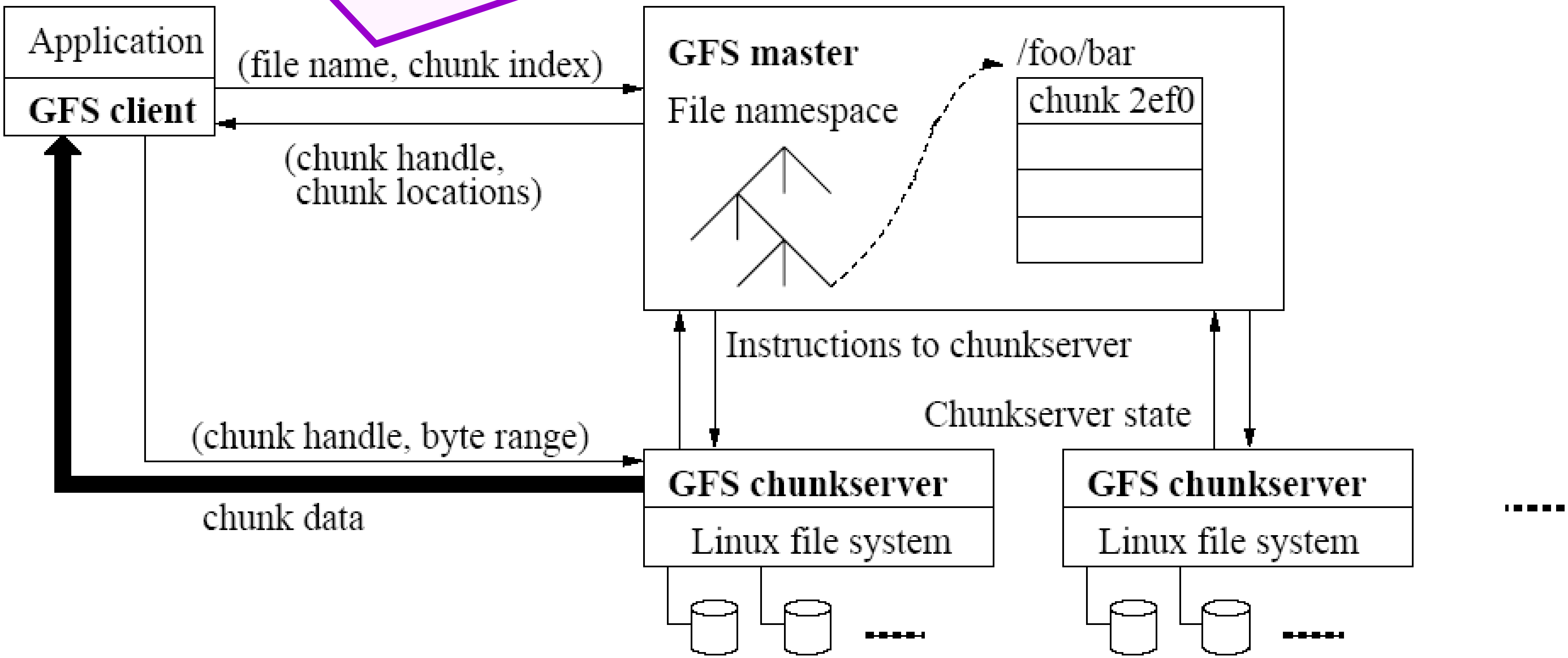


Request data from nearest chunkserver
"chunkhandle & index into chunk"

No need to talk more
About this 64MB chunk
Until cached info expires or file reopened



Often initial request asks about
Sequence of chunks



Metadata

- Master stores three types
  - File & chunk namespaces
  - Mapping from files → chunks
  - Location of chunk replicas
- Stored in memory
- Kept persistent thru logging

Consistency Model

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

Consistent = all clients see same data

Consistency Model

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*co[illegible]sistent* [illegible] *undefined*	
Failure	*inconsistent*	

Defined = consistent + clients see full effect of mutation
Key: all replicas must process chunk-mutation requests in ***same order***

Consistency Model

<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>

Different clients may see different data

Implications

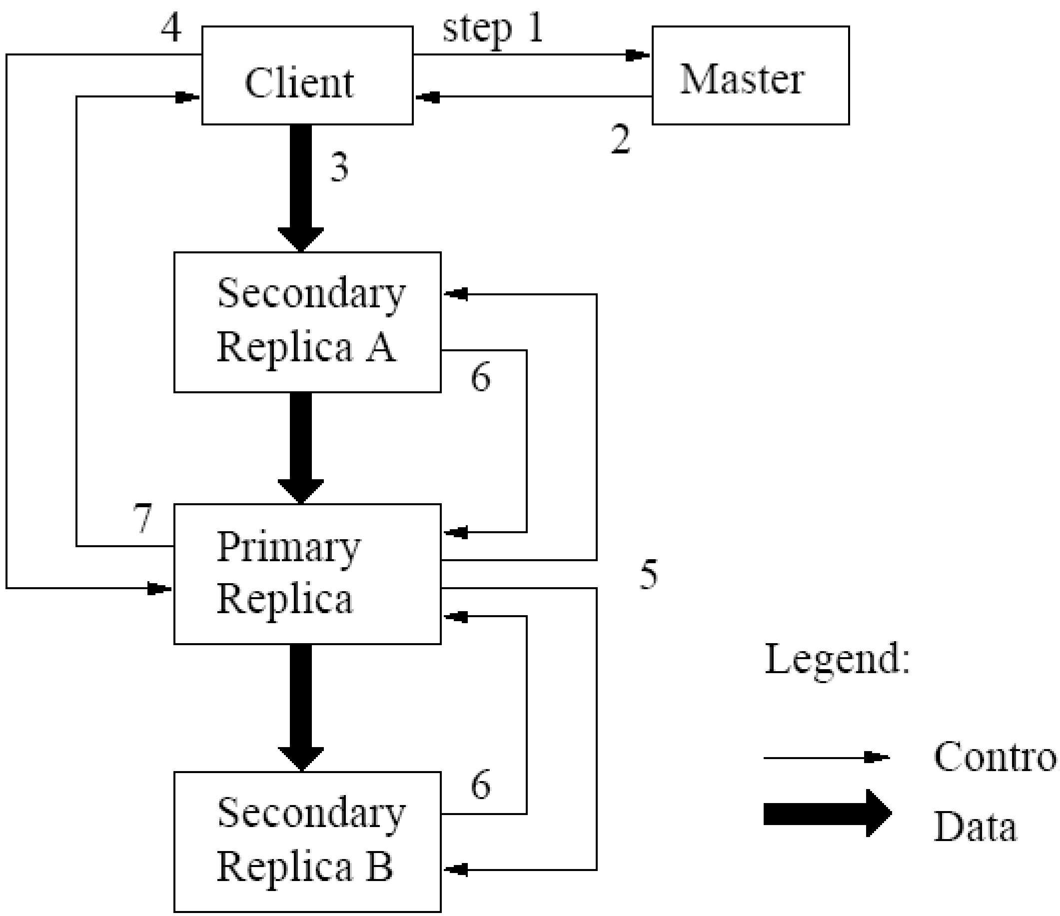
- Apps must rely on appends, not overwrites
- Must write records that
  - Self-validate
  - Self-identify
- Typical uses
  - Single writer writes file from beginning to end, then renames file (or checkpoints along way)
  - Many writers concurrently append
    - At-least-once semantics ok
    - Reader deal with padding & duplicates

Leases & Mutation Order

- Objective
  - Ensure data consistent & defined
  - Minimize load on master

- Master grants 'lease' to one replica
  - Called '**primary**' chunkserver
- Primary serializes all mutation requests
  - Communicates order to replicas

Write Control & Dataflow



Atomic Appends

- As in last slide, but…
- Primary also checks to see if append spills over into new chunk
  - If so, pads ***old*** chunk to full extent
  - Tells secondary chunk-servers to do the same
  - Tells client to try append again on ***next*** chunk

- Usually works because
  - max(append-size) < ¼ chunk-size [API rule]
  - (meanwhile other clients may be appending)

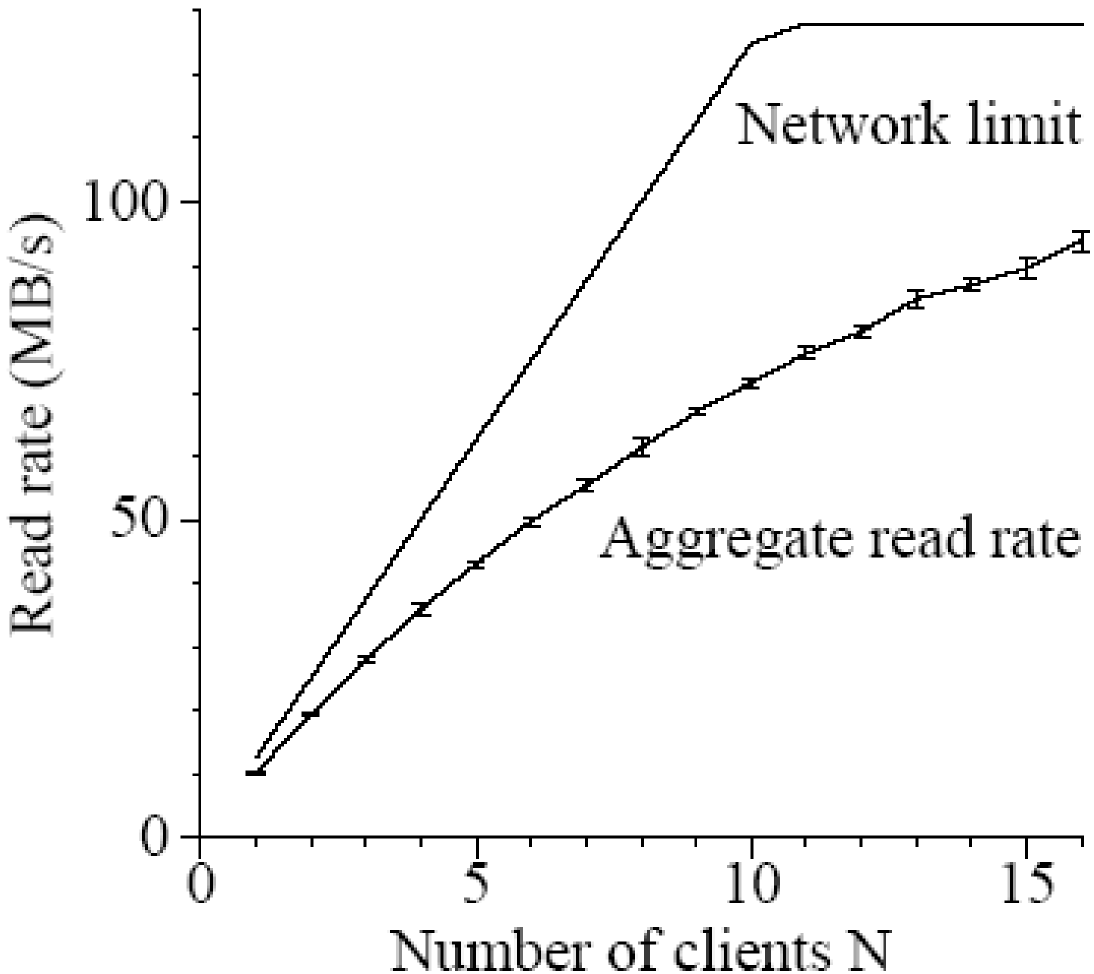
Other Issues

- Fast snapshot
- Master operation
  - Namespace management & locking
  - Replica placement & rebalancing
  - Garbage collection (deleted / stale files)
  - Detecting stale replicas

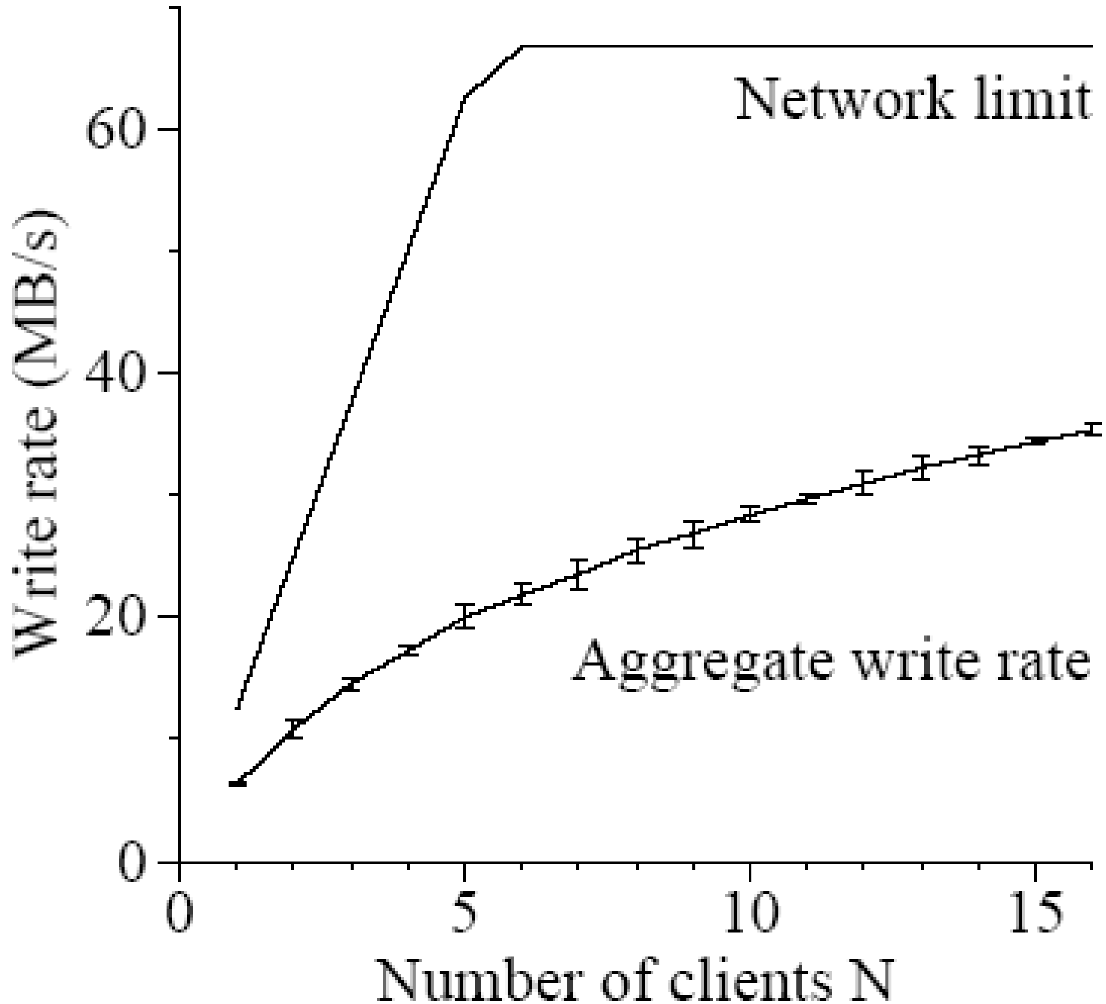
Master Replication

- Master log & checkpoints replicated
- Outside monitor watches master livelihood
  - Starts new master process as needed
- Shadow masters
  - Provide read-access when primary is down
  - Lag state of true master

Read Performance



Write Performance



Record-Append Performance