Staggeringly LARGE File Systems

Shrutarshi Basu
Advanced Systems

Motivations

- **LOTS** of data to store
- Storage must be reliable and available
- Lots of cheap distributed storage
- High bandwidth data links

Distribute

LARGE amounts of data

Highly Connected

Low Cost

Error Prone

- Pond: the OceanStore Prototype
    - Internet-scale untrusted storage
    - Distributed storage, distributed control
- The Google Filesystem
    - Google's trusted, managed Datacenters
    - Distributed storage, centralized control

GFS vs OceanStore

	GFS	**OceanStore**
Scale	Google	Internet
Architecture	Master + chunkservers	Primary + Secondary Replica
Control and Data	Separate	Combined
Target	Datacenters	Wide-area, distributed networks
Trust	Trust Everything	Untrusted nodes

Pond

The OceanStore Prototype

Sean Rhea, Patrick Eaton, Dennis Geels,
Hakim Weatherspoon, Ben Zhao,
John Kubiatowicz

Outline

- Problems and Assumptions
- Data Model
- System Architecture
- Pond Prototype
- Evaluation

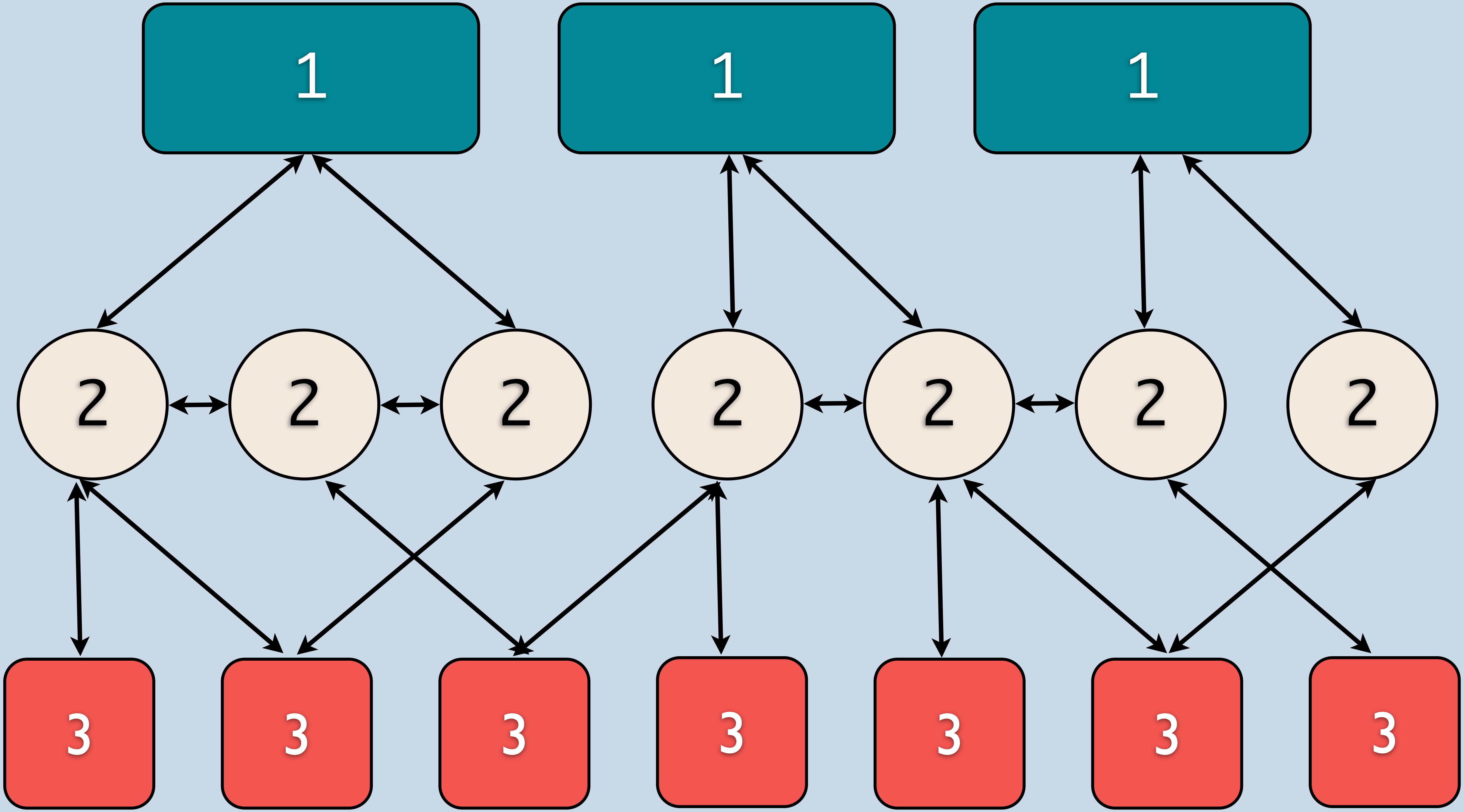
Dominant Cost of Storage

MANAGEMENT

- Health
- Performance
- Durability

Rising disk capacity per unit price

High bandwidth Internet connections


1
1
1
2
2
2
2
2
2
2
3
3
3
3
3
3
3

OceanStore Principles

- The unit of storage is the **data object**
- Information must be universally **accessible**
- **Balance** between the shared and the private
- **Consistency**, **Performance** and **Durability**
- Privacy complements **integrity**

Design A System …

Expressive Storage Interface

untrusted and changing base

OceanStore

Data Model

The model is designed to be general with full ACID semantics

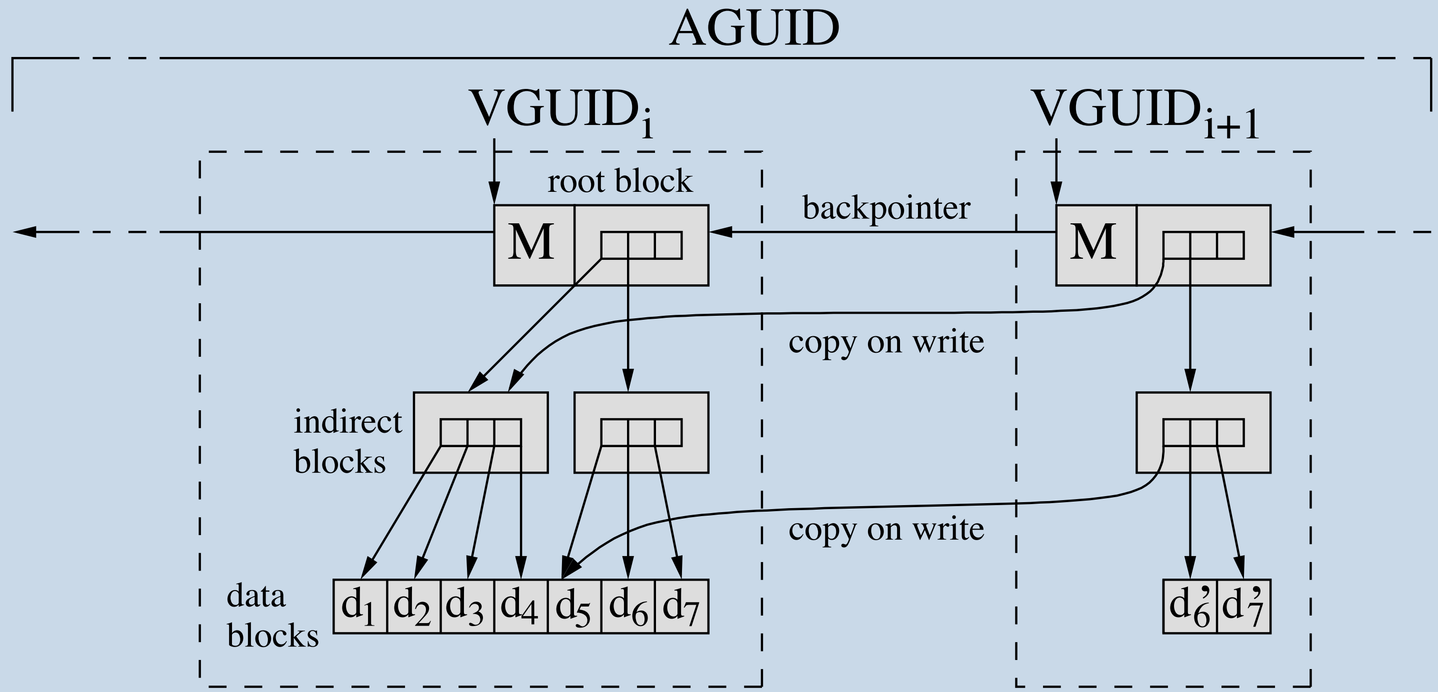


Figure 1: A data object is a sequence of read-only versions, collectively named by an *active* GUID, or AGUID. Each version is a B-tree of read-only blocks; child pointers are secure hashes of the blocks to which they point and are called *block* GUIDs. User data is stored in the leaf blocks. The block GUID of the top block is called the *version* GUID, or VGUID. Here, in version $i + 1$, only data blocks 6 and 7 were changed from version i, so only those two new blocks (and their new parents) are added to the system; all other blocks are simply referenced by the same BGUIDs as in the previous version.

Storage Organization

Application-specific **Consistency**

- An update adds a **version** to the head of an update stream
- Updates are applied **atomically**
- Updates are:
  - an array of potential **actions**
  - each action is guarded by a **predicate**
- Support a variety of consistency semantics
- No support for explicit locks; reliance on atomic update model instead

System Architecture

- Unit of synchronization is the data object
- Changes to different objects are independent

Virtualization through Tapestry

- Resources are identified by a **GUID**
- Not tied to any particular hardware
- Tapestry is a **decentralized** object location and routing system
- Objects addressed via GUID, not IP
- Tapestry routes messages to a **physical host** containing a resource with matching GUID

Replication and Consistency

- Hosts publish BGUIDs of blocks they store
- Primary-copy replication
- Digital Certificates: Heartbeats
- Let’s take a closer look at primary replicas

Primary Replicas

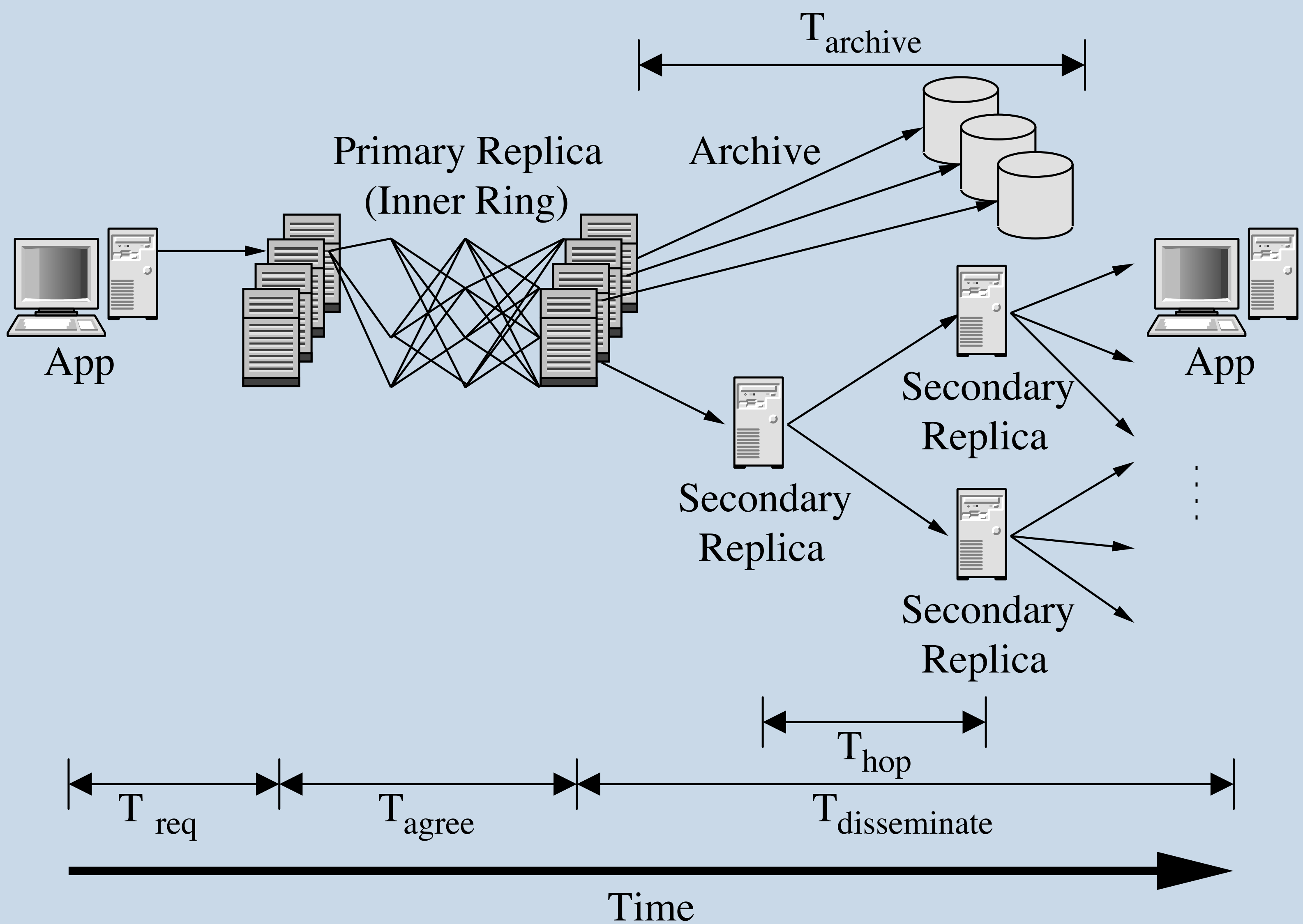
- Primary Replica is a **virtual** resource
- The Inner Ring is a small set of servers
- A **Byzantine** fault-tolerance protocol
- **Push based** update of secondaries
- Application level **multicast** tree

Primary Replicas Continued

- *(3f + 1)* servers, at most *f* may fail
- Use **public key cryptography** to communicate outside the Inner Ring
- Secondaries can **locally** verify authenticity
- Updates without authenticating **individually**
- Proactive threshold signatures and the **responsible parties**

Storage and Caching

- **Durability**:
  - *Erasure codes* achieve higher fault tolerance for the same additional cost
  - New blocks are erasure code and fragments are distributed across Tapestry
- **Performance** (whole block caching):
  - First hosts retrieves and combines fragments
  - First host publishes the cached block
  - Second host finds the cached copy



Full Update Path

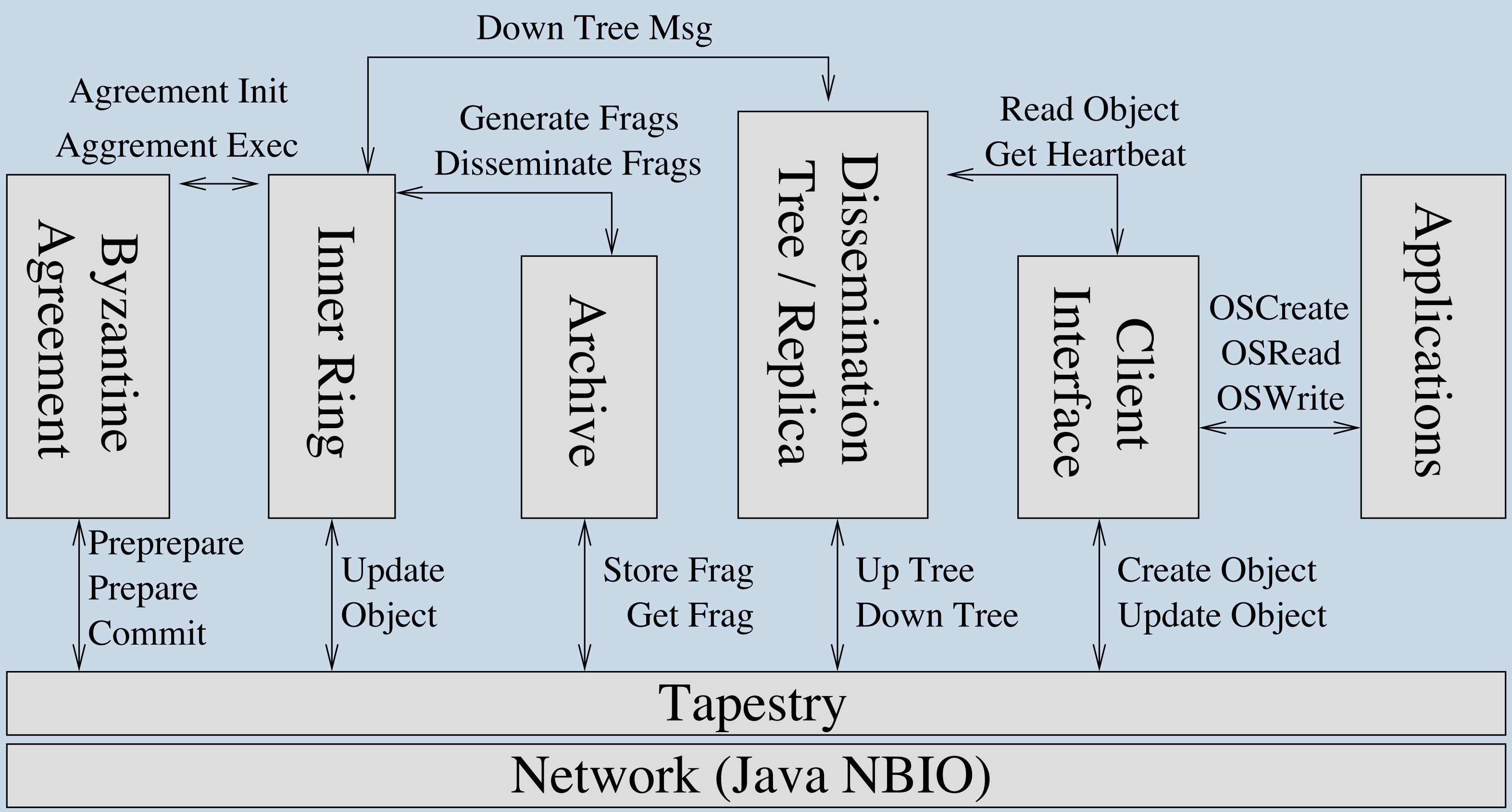


Figure 3: *Prototype Software Architecture*. Pond is built atop SEDA. Components within a single host are implemented as *stages* (shown as boxes) which communicate through events (shown as arrows). Not all stages run on every host; only inner ring hosts run the Byzantine agreement stage, for example.

Pond Prototype

Inner Ring	Client	Avg. Ping	Update Size	Update Latency (ms) 5%	Median	95%
Cluster	Cluster	0.2	4 kB	98	99	100
			2 MB	1098	1150	1448
Cluster	UCSD	27.0	4 kB	125	126	128
			2 MB	2748	2800	3036
Bay Area	UCSD	23.2	4 kB	144	155	166
			2 MB	8763	9626	10231

Wide Area Latency

Takeaways

- Internet-scale persistent data storage
- Incremental scalability, secure sharing and durability
- Byzantine updates, push updates, archival by erasure coding
- Pond prototype supporting multiple applications

Questions and Comments?

Google

File System

Sanjay Ghemawat
Howard Gobioff
Shun-Tak Leung

Outline

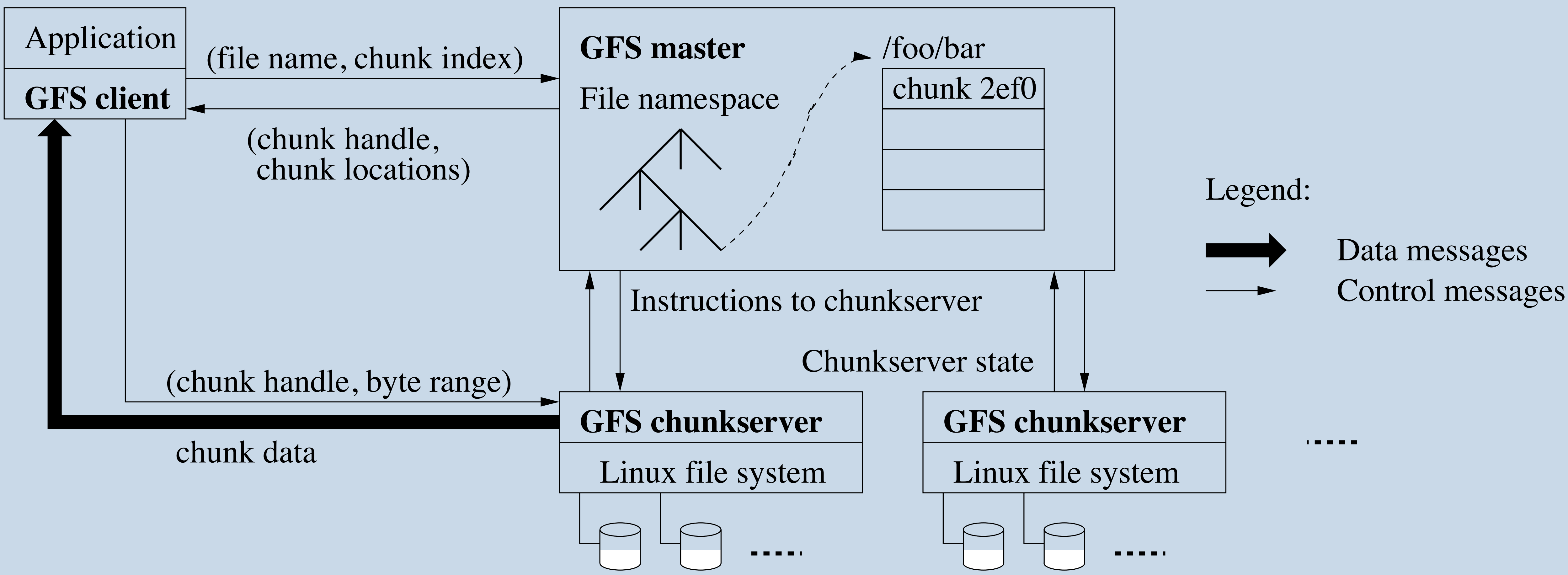
- Problems and Assumptions
- Design Overview
- System Interactions
- Master Operation
- Measurements
- Takeaways

Google-scale Problems

- Component failures are the norm
- Files are huge by traditional standards
- Appending is more common than over-writing
- Benefits of co-designing apps and file system

Assumptions

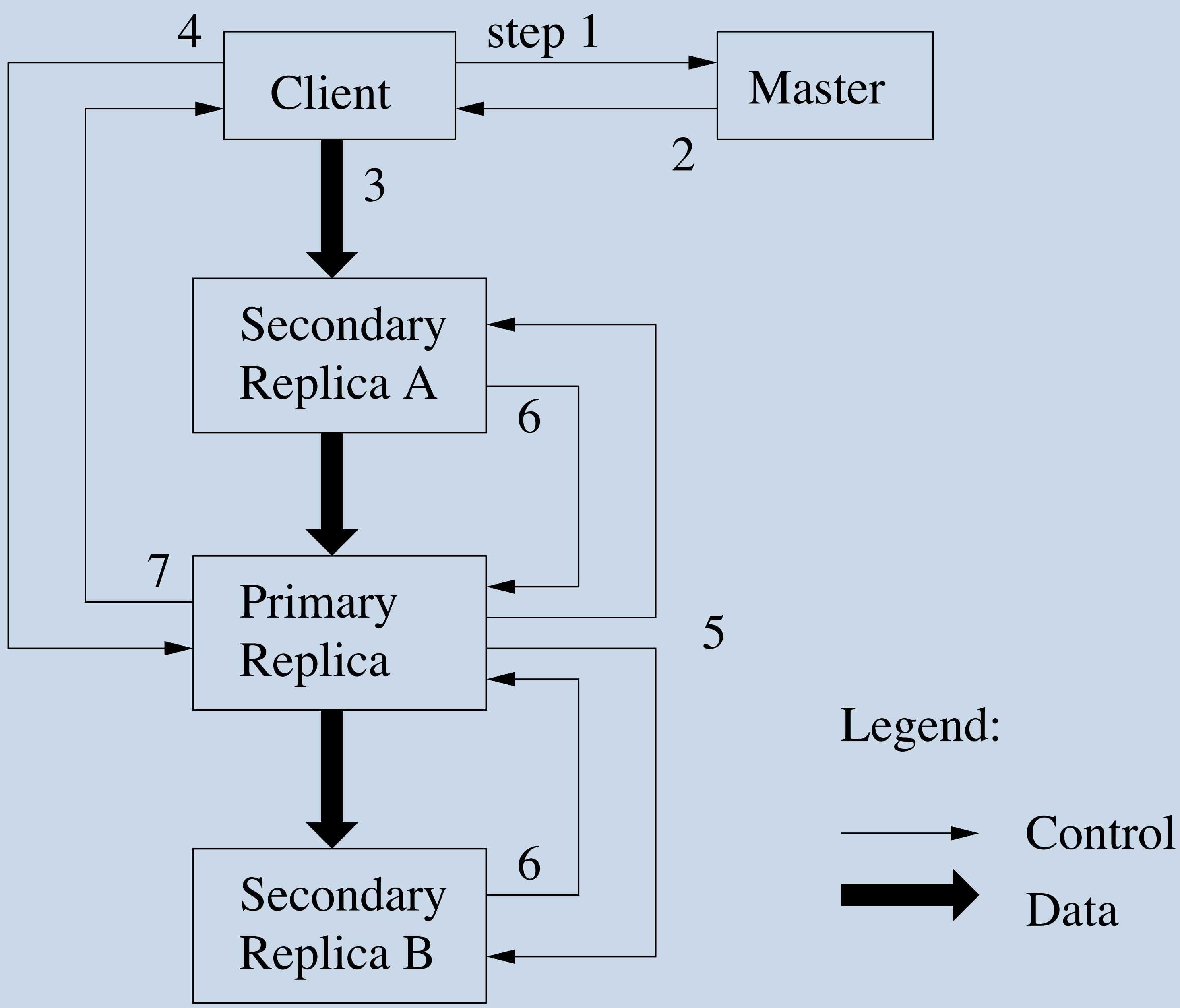
- Targeting Google **Datacenters**
- Cheap commodity components that **fail often**
- System stores a modest number of **large** files
- **Large streaming** and **small random** reads
- Many large, **sequential** writes
- Well-defined semantics for **concurrent appends**
- High sustained **bandwidth** over low latency



Design Overview

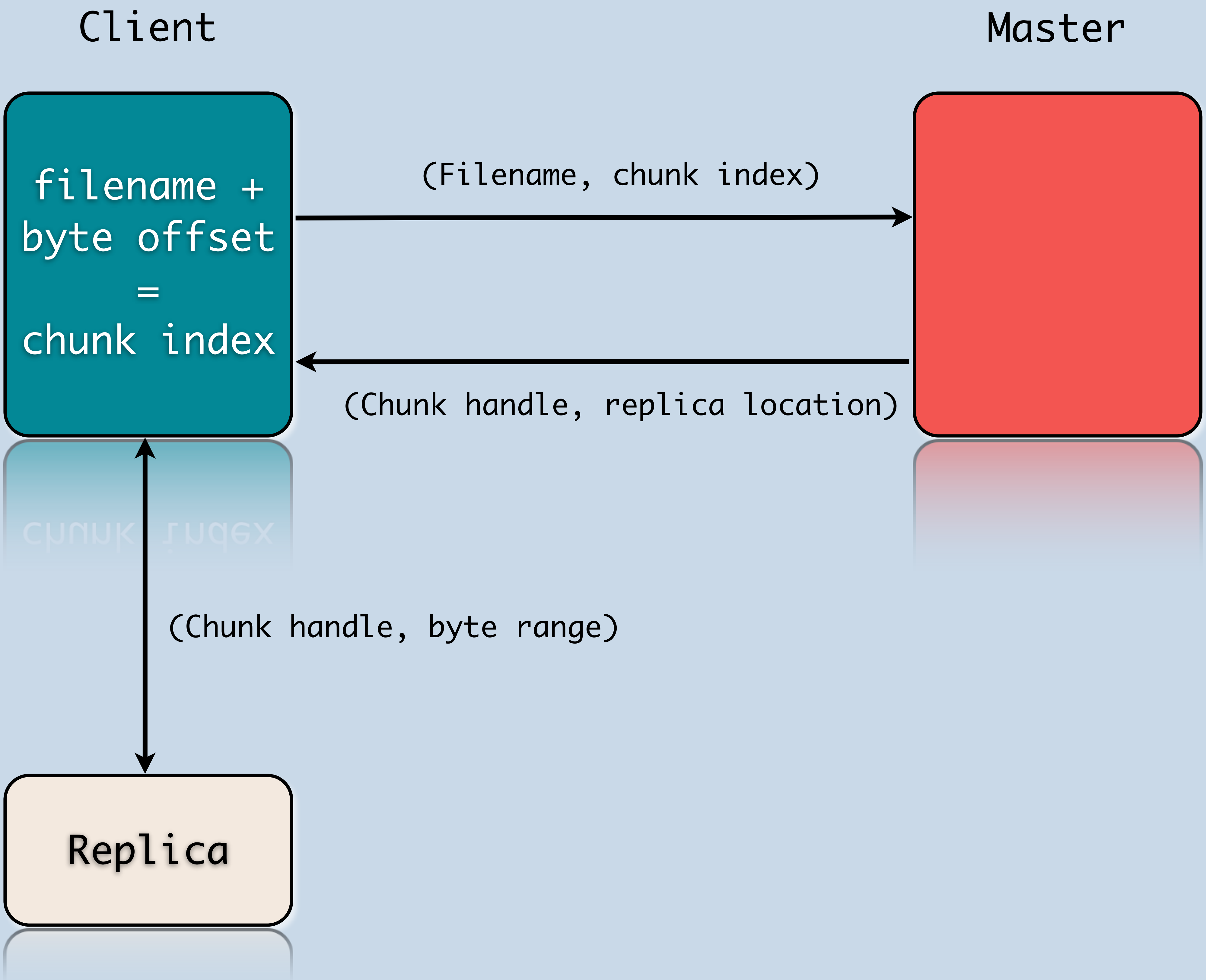
Architecture

- Single master, multiple chunk servers
- 64MB chunk size with 64 bit handle
- Master metadata
  - File and chunk namespaces
  - Mapping from files to chunks
  - Location of chunk replicas (volatile)



System Interaction

A Typical Read



Master Operation

- Namespace Management and Locking
- Replica Placement
- Creation, Re-replication & Rebalancing
- Garbage Collection
- Stale Replica Detection

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>

Consistent: all clients see the same data

Defined: Consistent + clients see complete mutation

Implications for Applications

- Favor appends over writes
- Checkpoints with app-level checksums
- Self-validating, self-identifying records

Takeaways

- Treat **failure** as the norm
- Monitoring, replication and recovery
- High **throughput** for concurrent access
- **Separate** FS control from data transfer

Questions and Comments?

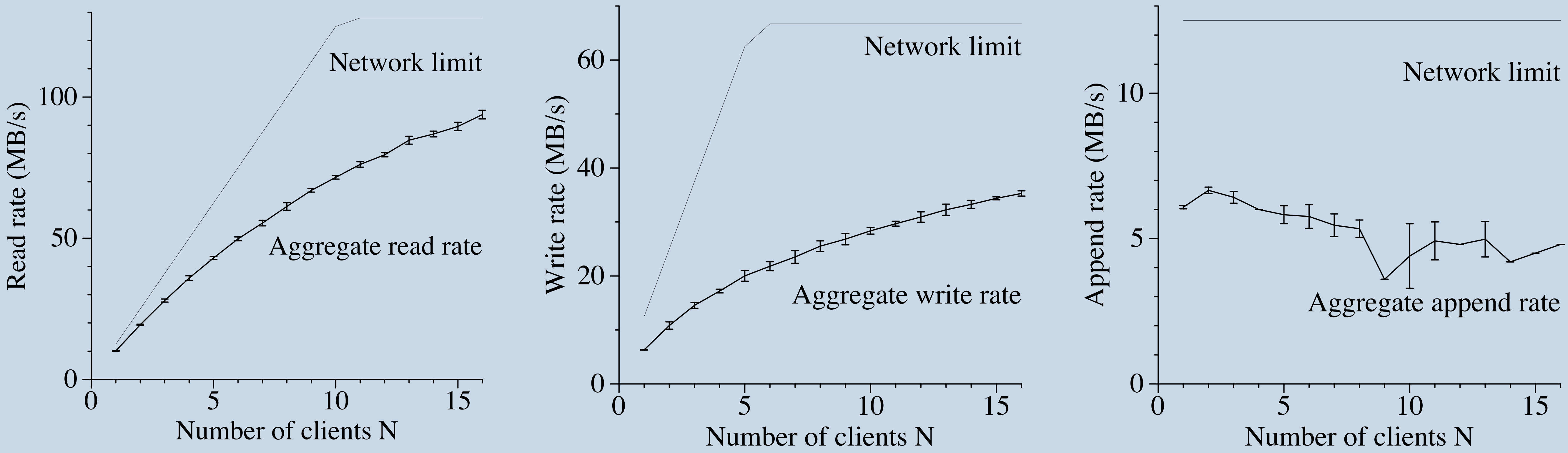
Appendix

GFS Evaluations

- Microbenchmarks (reads, writes, appends)
- Real world clusters

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Cluster Characteristics



(a) Reads (b) Writes (c) Record appends

Microbenchmarks

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

Cluster Performance

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

Table 4: Operations Breakdown by Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested.

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
1B..1K	< .1	< .1	< .1	< .1	< .1	< .1
1K..8K	13.8	3.9	< .1	< .1	< .1	0.1
8K..64K	11.4	9.3	2.4	5.9	2.3	0.3
64K..128K	0.3	0.7	0.3	0.3	22.7	1.2
128K..256K	0.8	0.6	16.5	0.2	< .1	5.8
256K..512K	1.4	0.3	3.4	7.7	< .1	38.4
512K..1M	65.9	55.1	74.1	58.0	.1	46.8
1M..inf	6.4	30.1	3.3	28.0	53.9	7.4

Table 5: Bytes Transferred Breakdown by Operation Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.

Cluster Performance

Pond Evaluations

Overheads, Update and Retrieval Performance, Replication

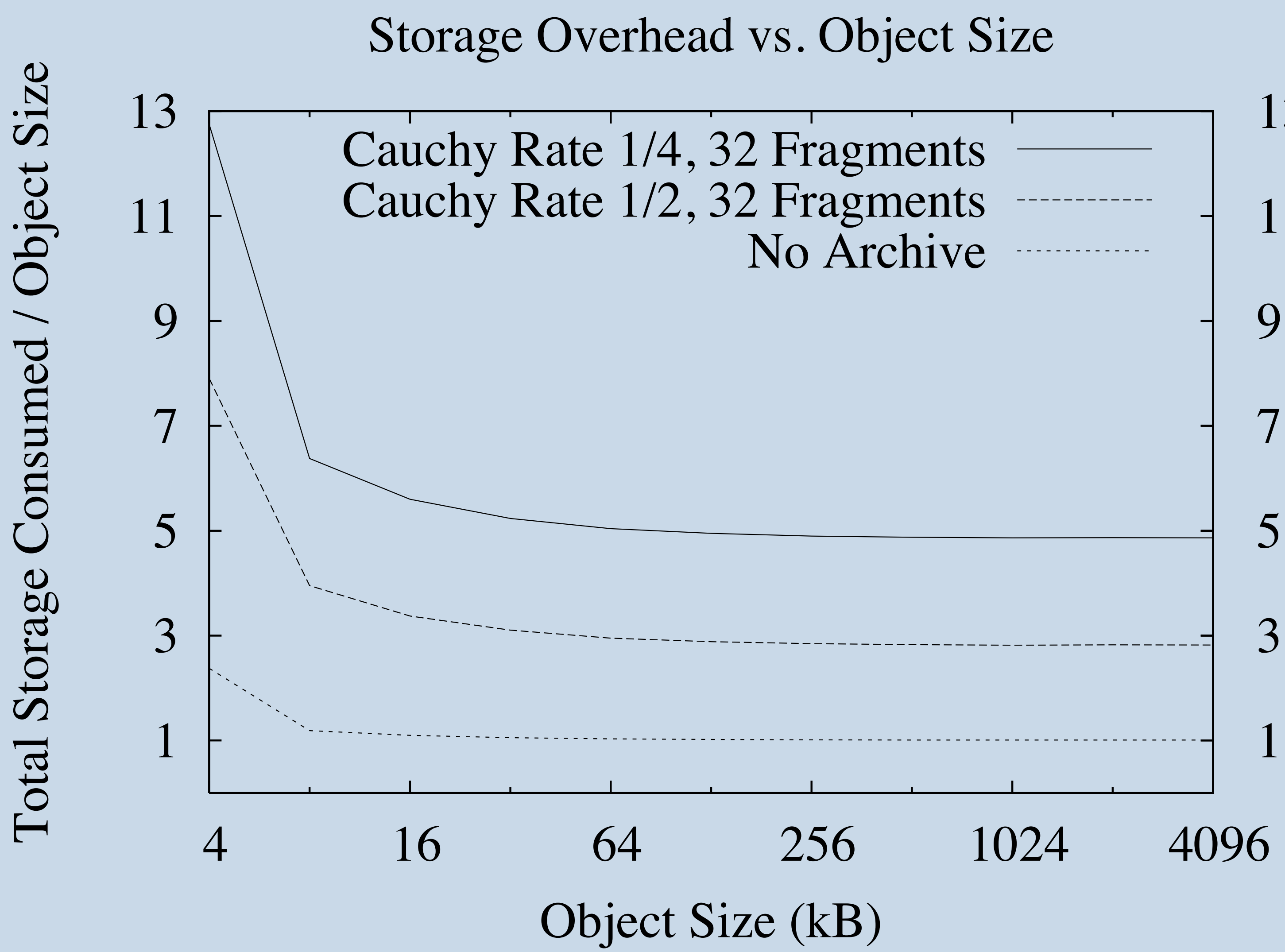
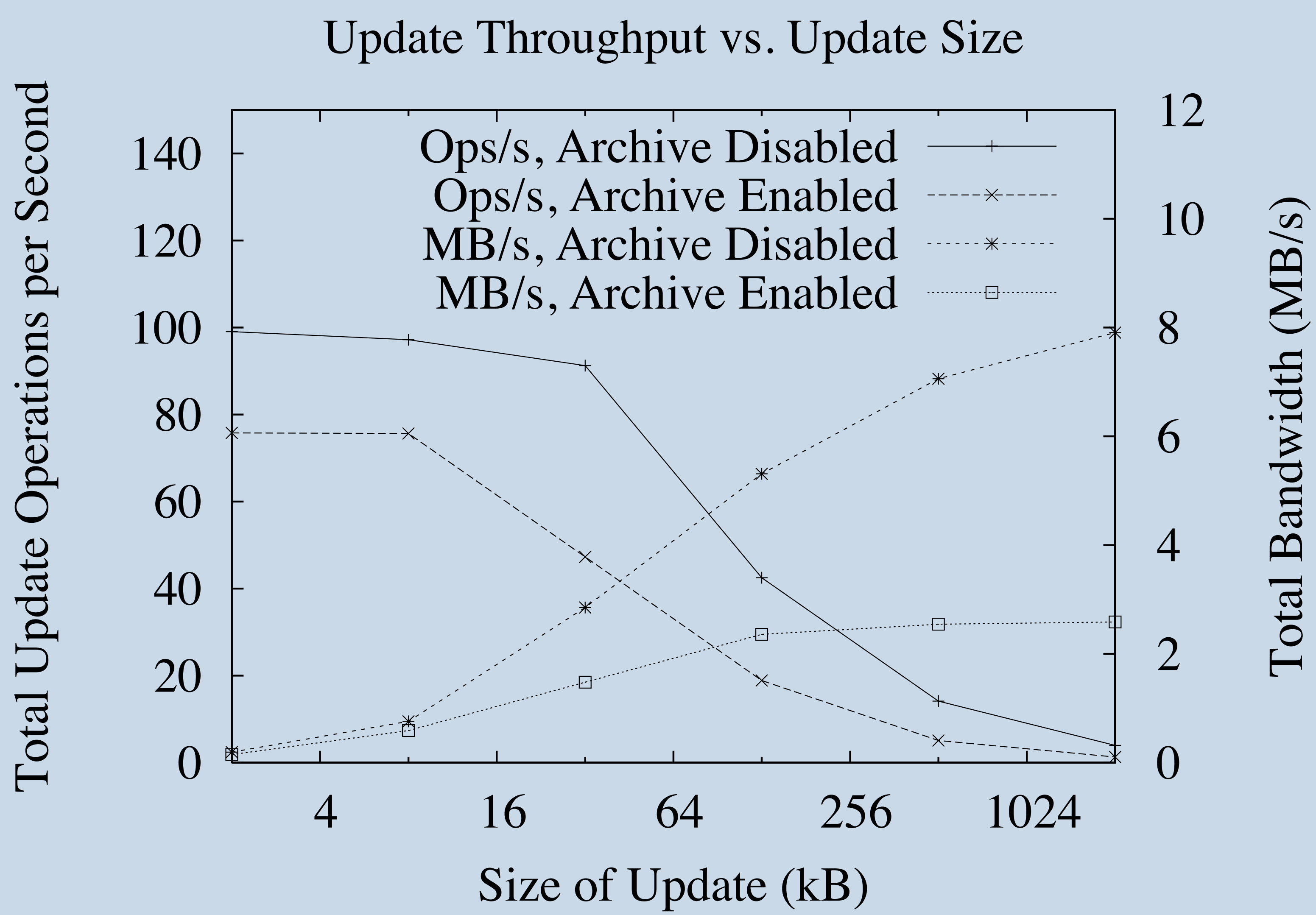
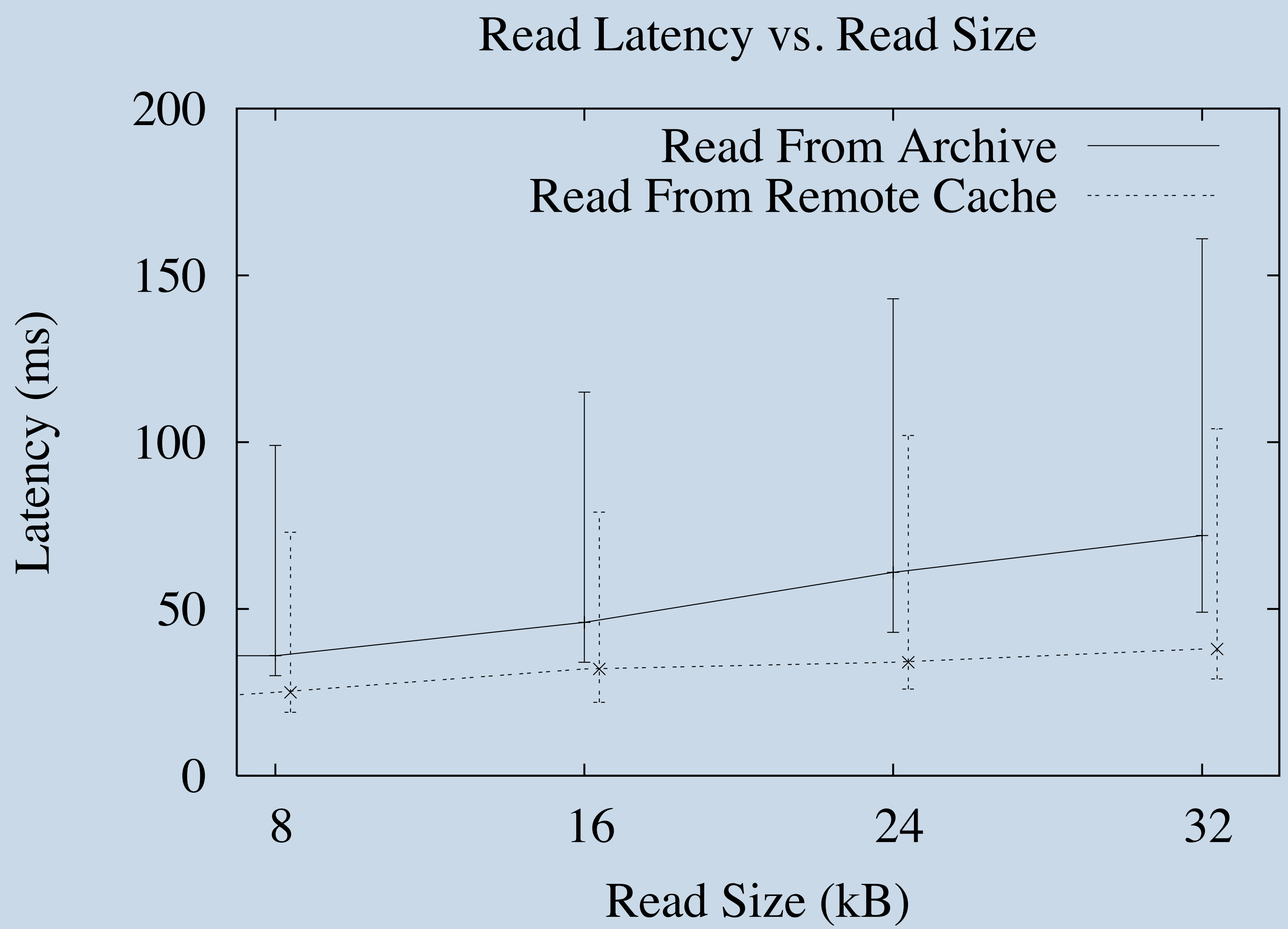


Figure 4: *Storage Overhead*. Objects of size less than the block size of 8 kB still require one block of storage. For sufficiently large objects, the metadata is negligible. The cost added by the archive is a function of the encoding rate. For example, a rate 1/4 code increases the storage cost by a factor of 4.8.

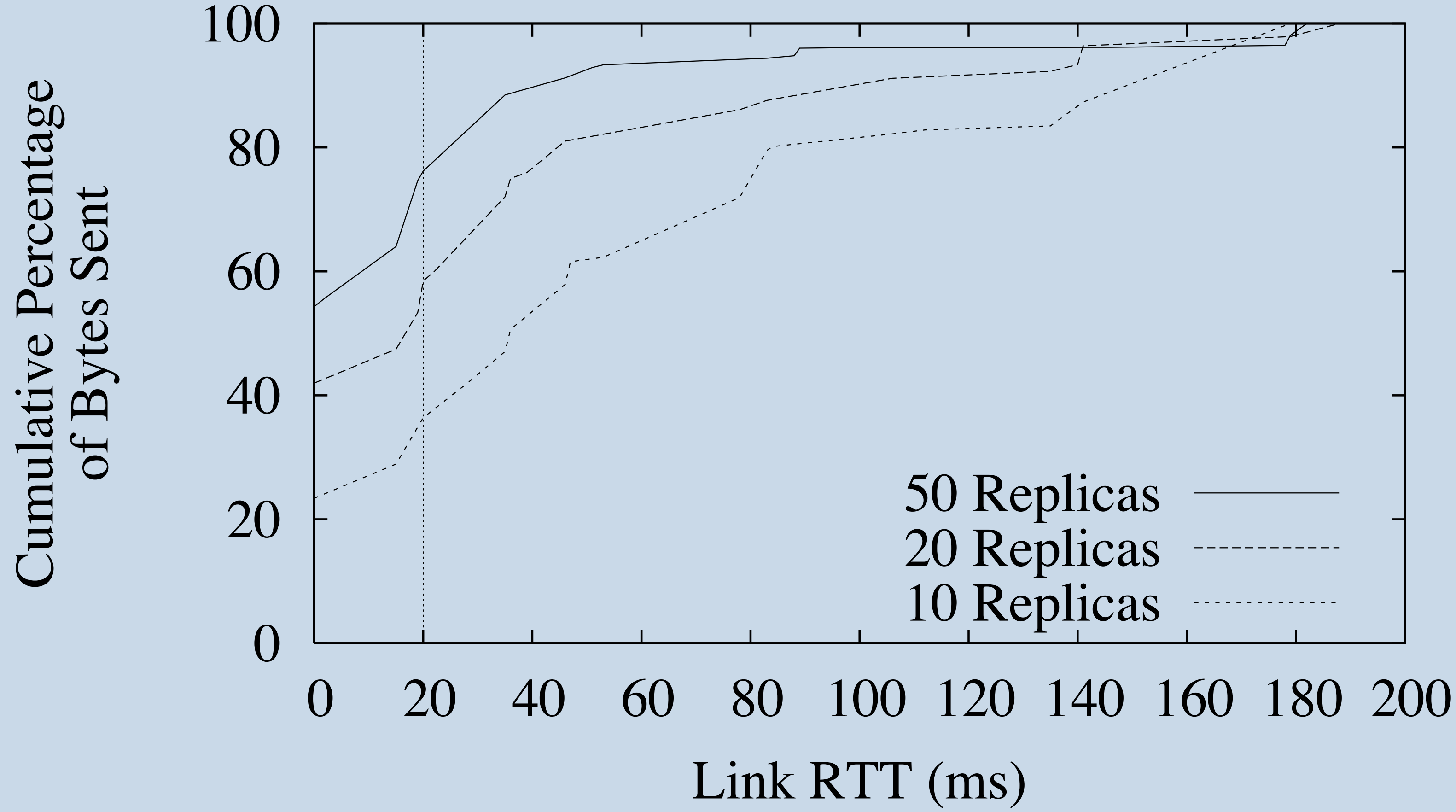
Storage Overhead



Update Throughput



Read Latency



Replication