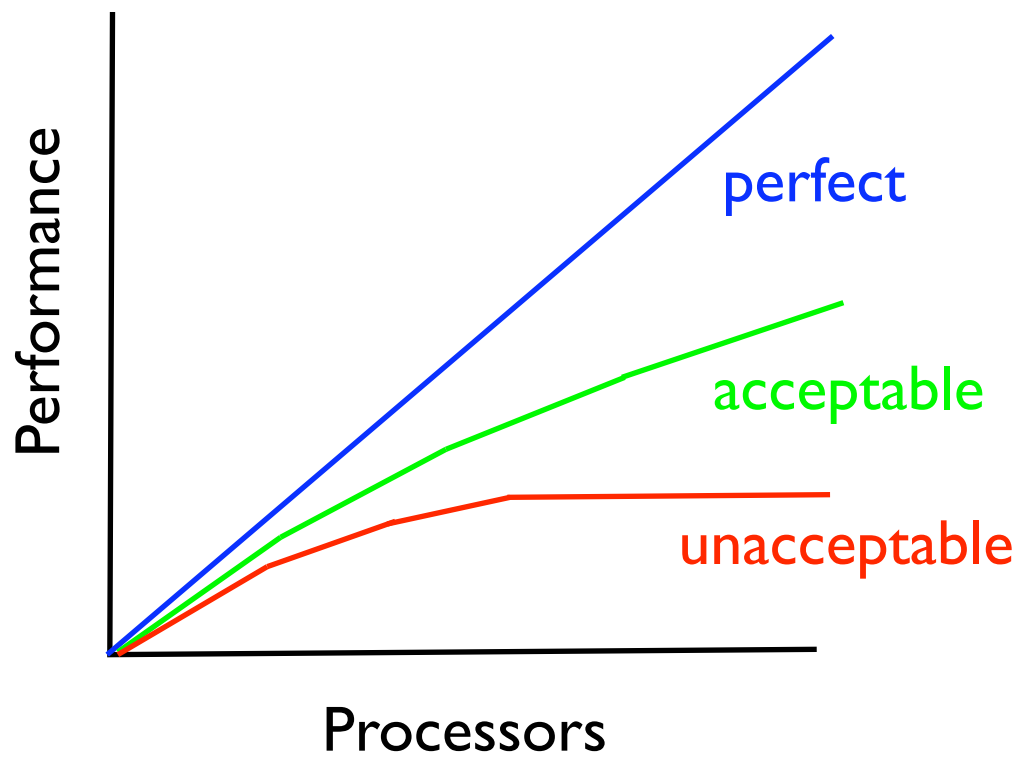
Engineering Goals

- Scalability
- Availability

- Transactional behavior
- Security
- EAI
- ...

Scalability

- How much performance can you get by adding hardware ($)?



Availability

- What fraction of the time is the system up?
- Percent availability:
- MTTF / (MTTF + MTTR) *
- Consider two systems:
- A: crashes every 15 mins, recovers in 10 secs
- B: crashes once per year, down for 4 days
-
- Each is about 99% available ...
- Careful engineering can achieve 99.99% or 99.999% availability.

* this is quite naive!

Replication and Fault-Tolerance

- These terms are used and abused ...
  - replicate hardware
    - clusters
    - RAID
  - replicate data
    - a number of commercial dbms

Replicating Hardware

- Usually the approach is:

  Build a {fast,reliable} system
  from cheap, {slow, unreliable} pieces

- may improve performance
  - if application parallelizes well
- may improve reliability/availability
  - if system tolerates failures

Replicating Data

- Commercial DBMS like MS or Oracle
- P2P systems
- The Google papers

  - may improve performance for read-mostly application or weak consistency
  - may improve availability if the application can fail over

The Google Papers ...

- The Google Cluster Architecture
- The Google File System (GFS)

Two scalable high-performance systems

Highly parallelizable applications
Weak transaction / data consistency model

Build a huge cluster of PCs
engineer for fault tolerance and recovery

Google Search Engine - Scale

- More than 125,000,000 searches / day
  - average well over 1000 searches / sec
- Search accesses > 100 MB of data
- Search uses > 10 billion CPU cycles

Maybe Not So Much ...

- Suppose you do 1,000 searches / sec
- ... but you have 15,000 machines

- ... you get to spend 15 machine seconds per search!

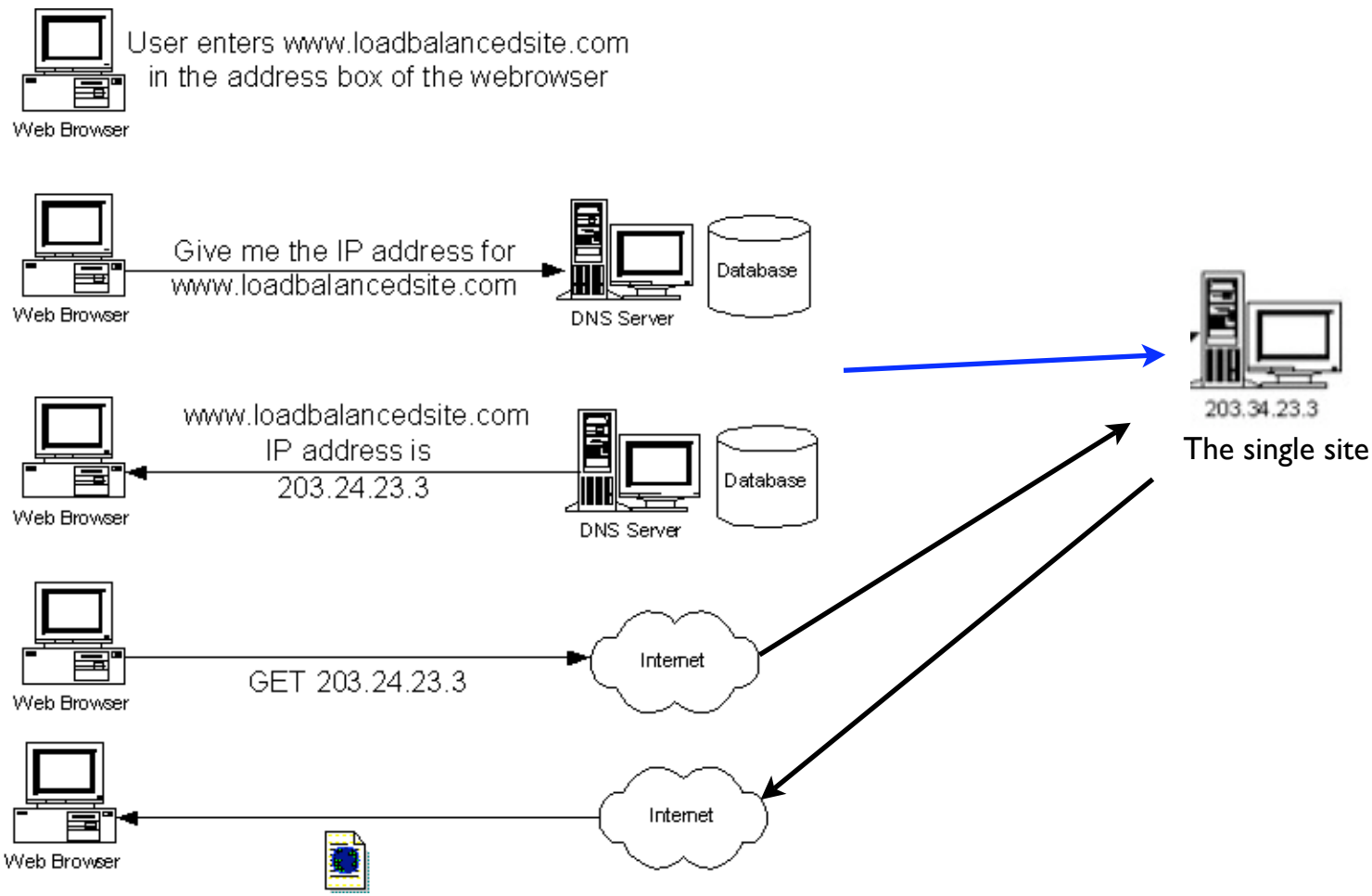
- So the trick is to parallelize effectively.

Top Layer

- Google has multiple clusters of Google Web Servers (GWS's)
- Each cluster is 1000's of machines
- Client requests is routed to a GWS using combination of DNS and local hardware load balancing

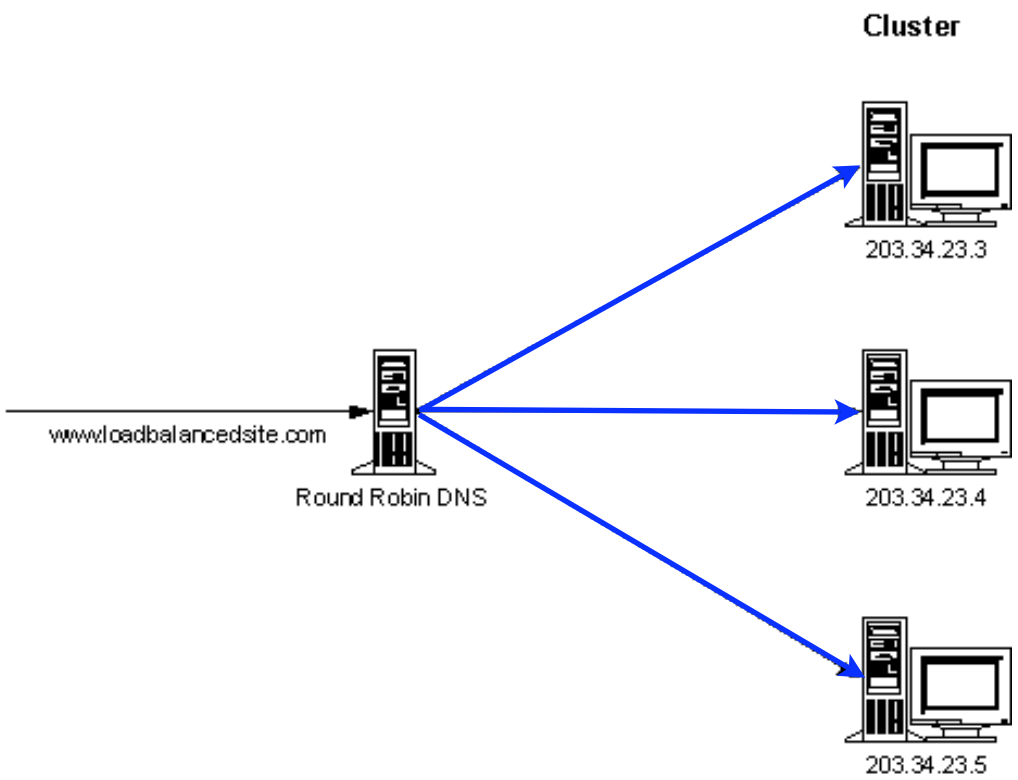
DNS Load Balancing

Ordinary DNS Lookup



DNS Load Balancing

Round-Robin DNS Lookup



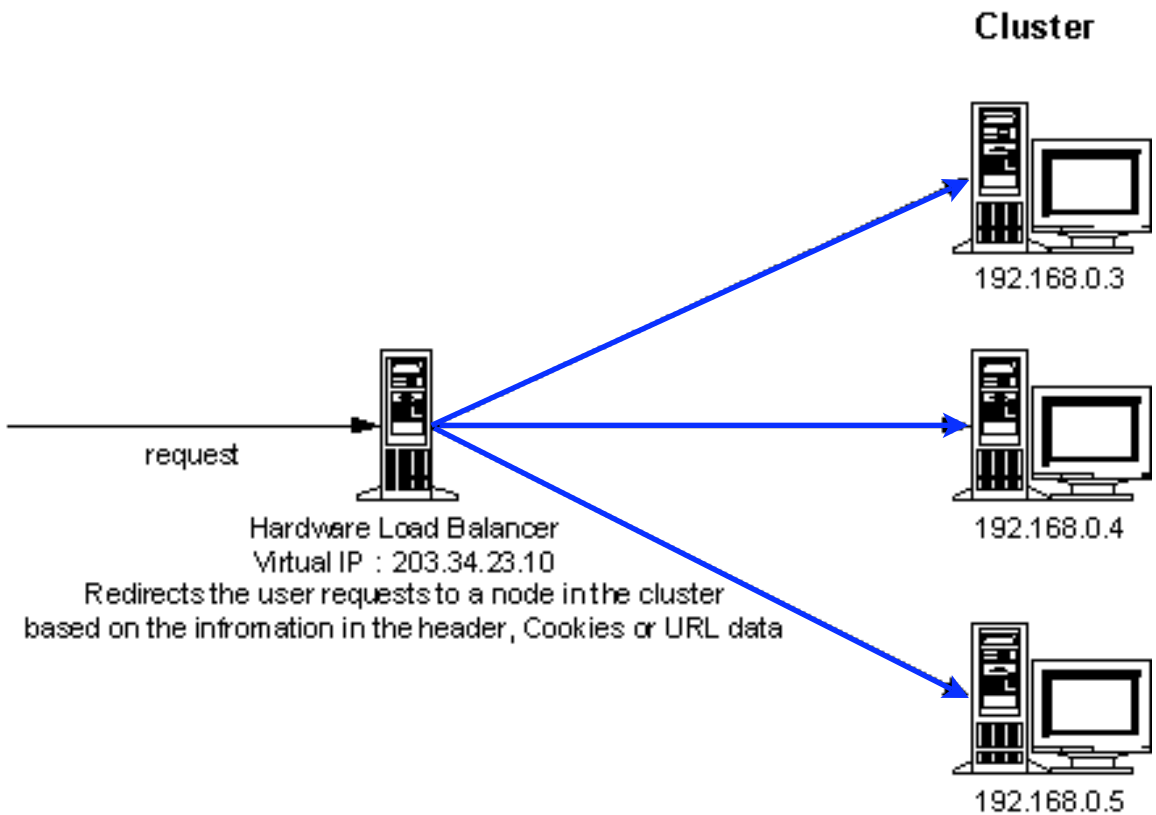
DNS Load Balancing

- In principle, could use many other schemes besides round-robin
  - proximity
  - server load
- Other tricks can be played by overloading the semantics of DNS
  - Akamai
  - send failed lookups to my portal site ...

Hardware Load Balancing

Client always sees same (cluster) address

Load balancer forwards to chosen server within the cluster

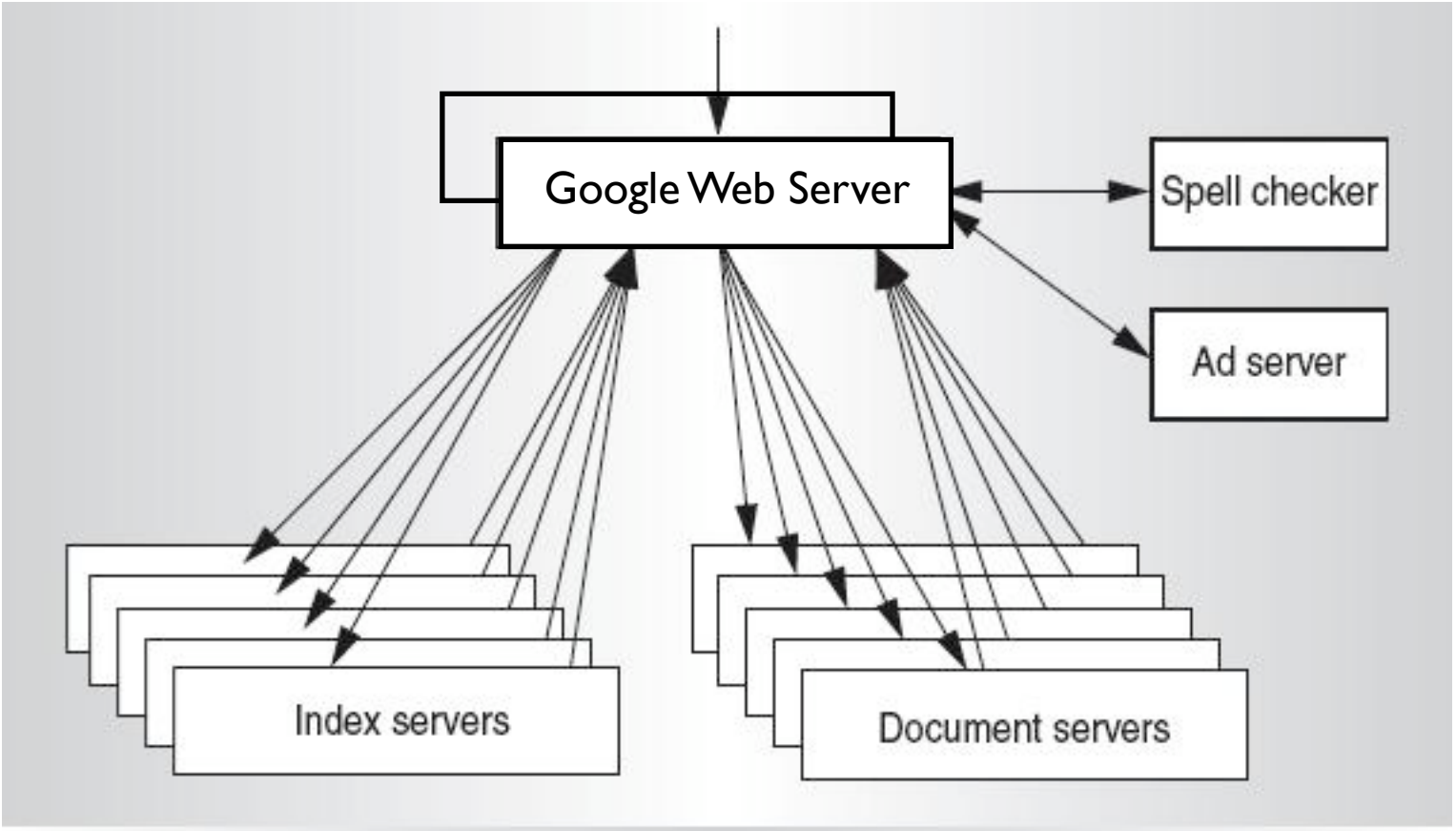


Architecture

Multiple GWS instances

Multiple replicated index servers

Multiple replicated document servers

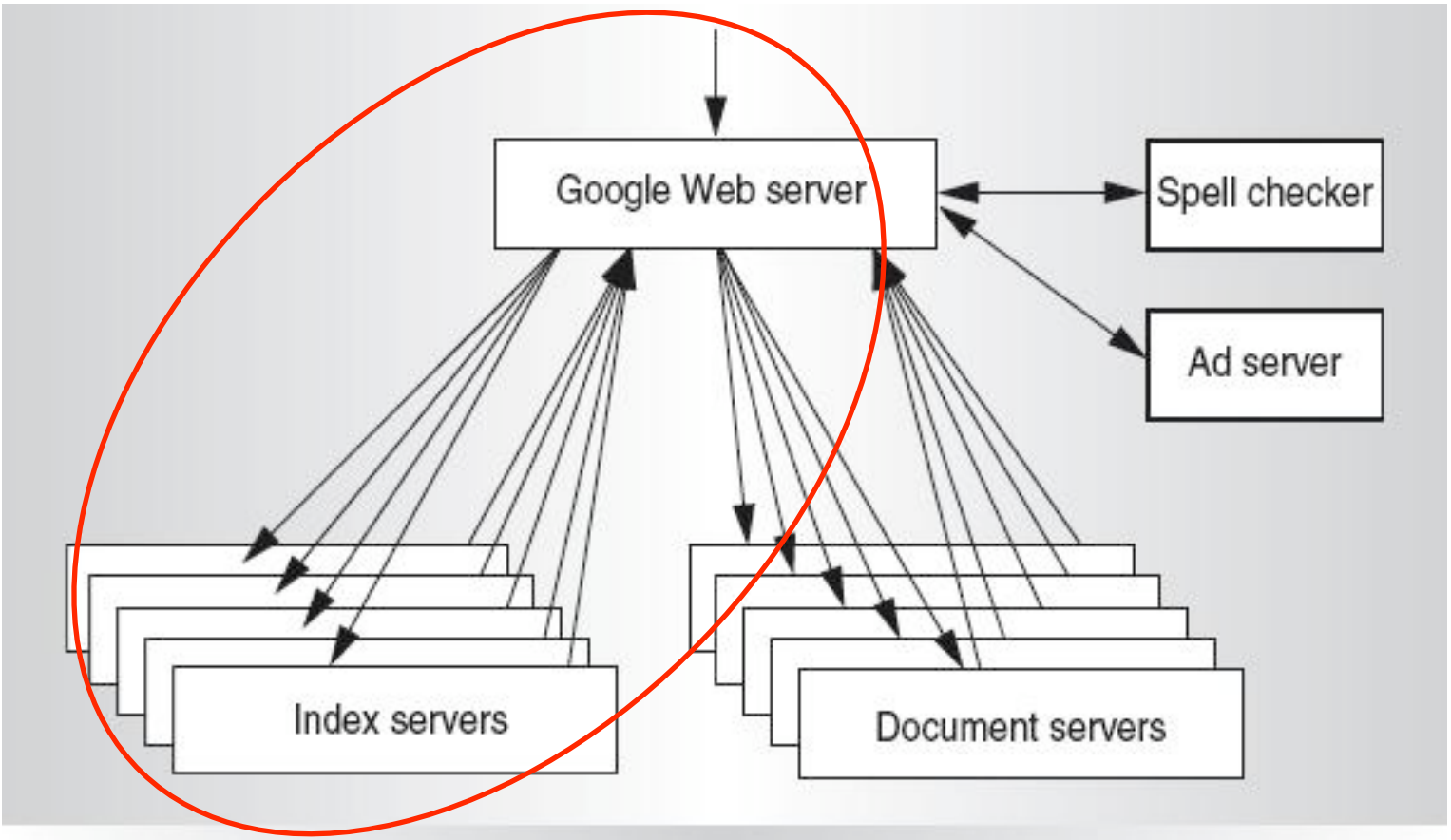


Processing a Query - Phase I

Index server holds *shard* -- random subset

Multiple server replicas for each shard

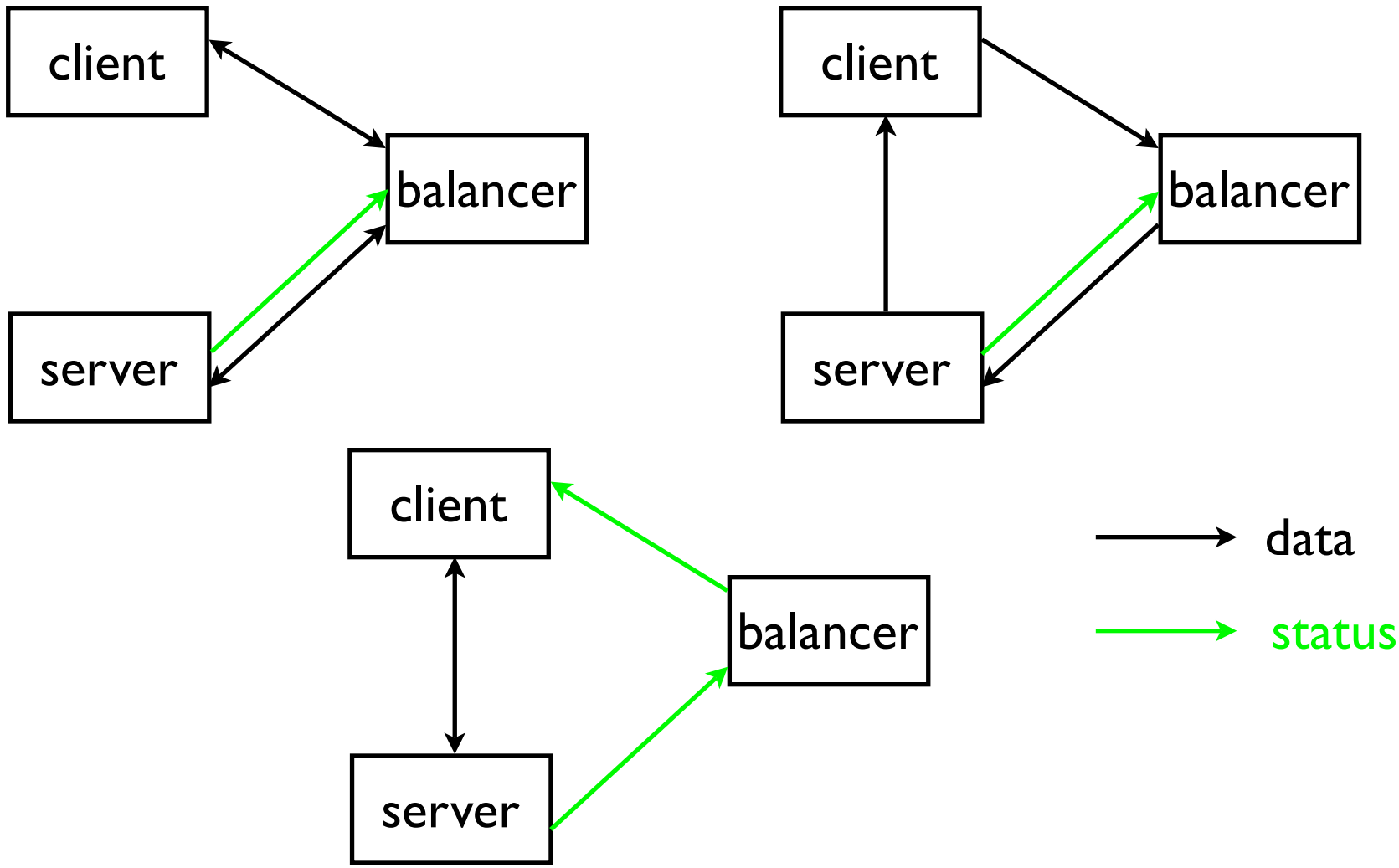
Route requests through load balancer



Phase 1 - More Details

- Shard is (completely) random subset of documents
- Replicated index servers for each shard
- Full set of keywords sent to *one* replica server for *each* shard
- Result is list of *docids* ordered by relevance
  - merge lists from each shard

Load Balancing Options

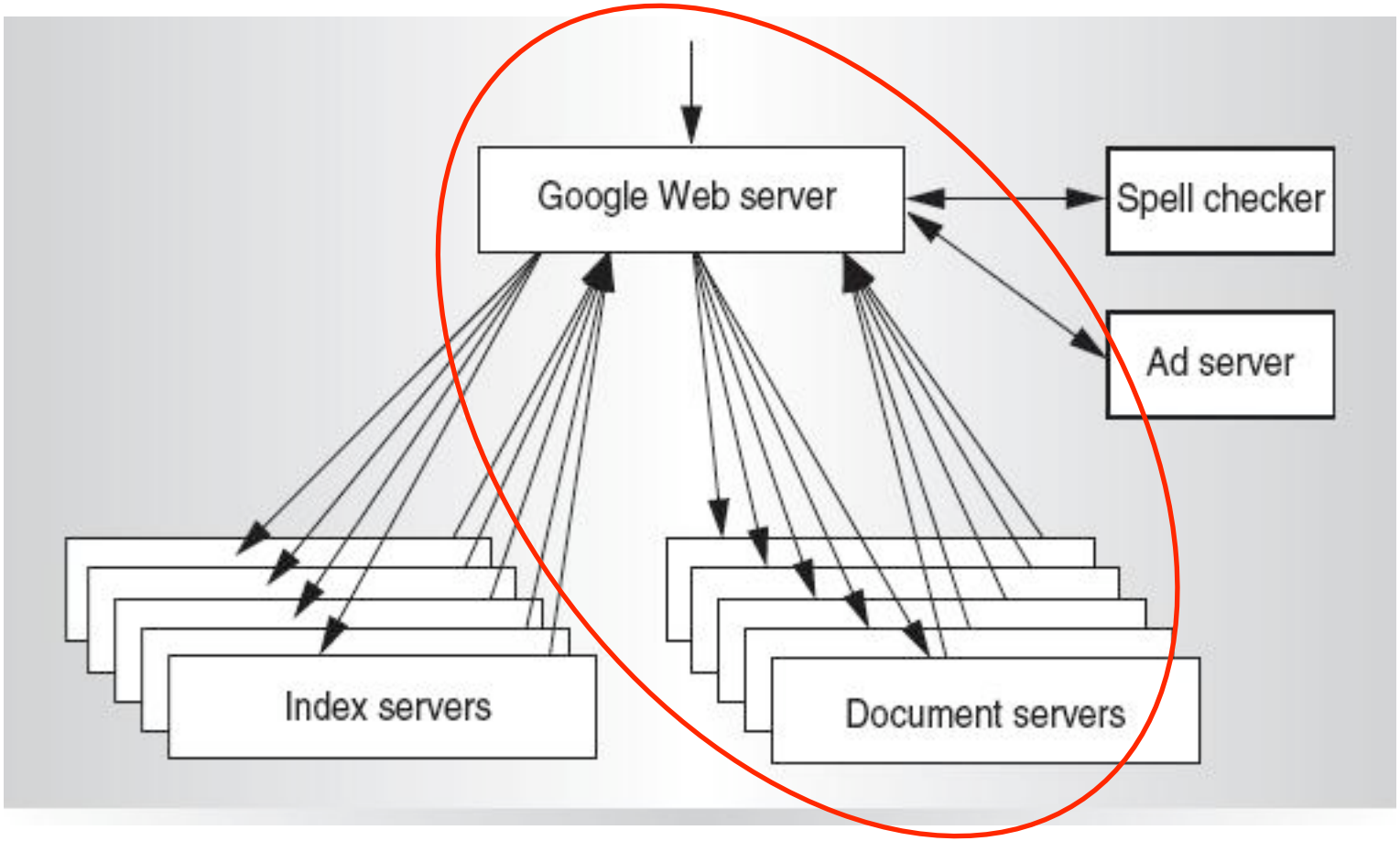


Processing a Query - Phase 2

Now have ordered list of matching *docids*

Doc servers hold random subsets

Route requests through load balancer



Phase 2 - More Details

- Doc servers split into shards
  - shard must be deterministic function of *docid* (e.g. hash)
  - otherwise would need to send every *docid* to every shard doc server
- Load balancing options as above

"Embarrassingly Parallel" Problem

- Each GWS (and processing of each query) is essentially independent of all the others
- Weak consistency requirements
  - If a search result is slightly out-of-date w.r.t. latest Web crawl data, so what?
  - rolling insert to doc servers then shard servers
  - rolling delete from shard servers then doc servers

Design Principles Revisited

- Get reliability from fault-tolerant software
- Use replication to improve throughput and availability
- Price/performance beats peak performance
- Commodity PCs provide cheapest cycles

More Issues

- "Maintaining 1000 servers is not more expensive than 100 if they all have identical configurations."
  - with *really good* cluster mgt software
- Power: densely packed PCs = 400-700 w/ft
  - data center = 150 w/ft
- Claim: Pentium 4 speculative execution already beyond point of diminishing returns
  - for this application mix

Google File System (GFS) Paper

- A more detailed design description
- Scalability (up to a point) and a stronger consistency model:
  - centralized server for synchronization
  - offload most of work to slaves

Design Assumptions

- Many commodity PCs
  - software failure detection
  - software recovery
  - you *can* make a silk purse from sows' ears as long as you use *enough* of them ...
- A few (million) large (>100MB) files
  - smaller files need not be supported very efficiently

Design Assumptions - 2

- Read workload
  - *large* streaming reads
  - *small* random reads
- Write workload
  - large appends
  - *atomic* append
- Infrequent reuse
  - *caching* at file system level inappropriate

Design Assumptions - 3

- Performance criteria:
  - *throughput* more important than *latency* of individual operation
  - *aggregate throughput* more important than individual client throughput
    - Remark: a commercial RDBMS that does 10,000 TPS can't process a transaction in 100 microseconds!

Architecture Diagram

- Single *master* and multiple *chunk servers*

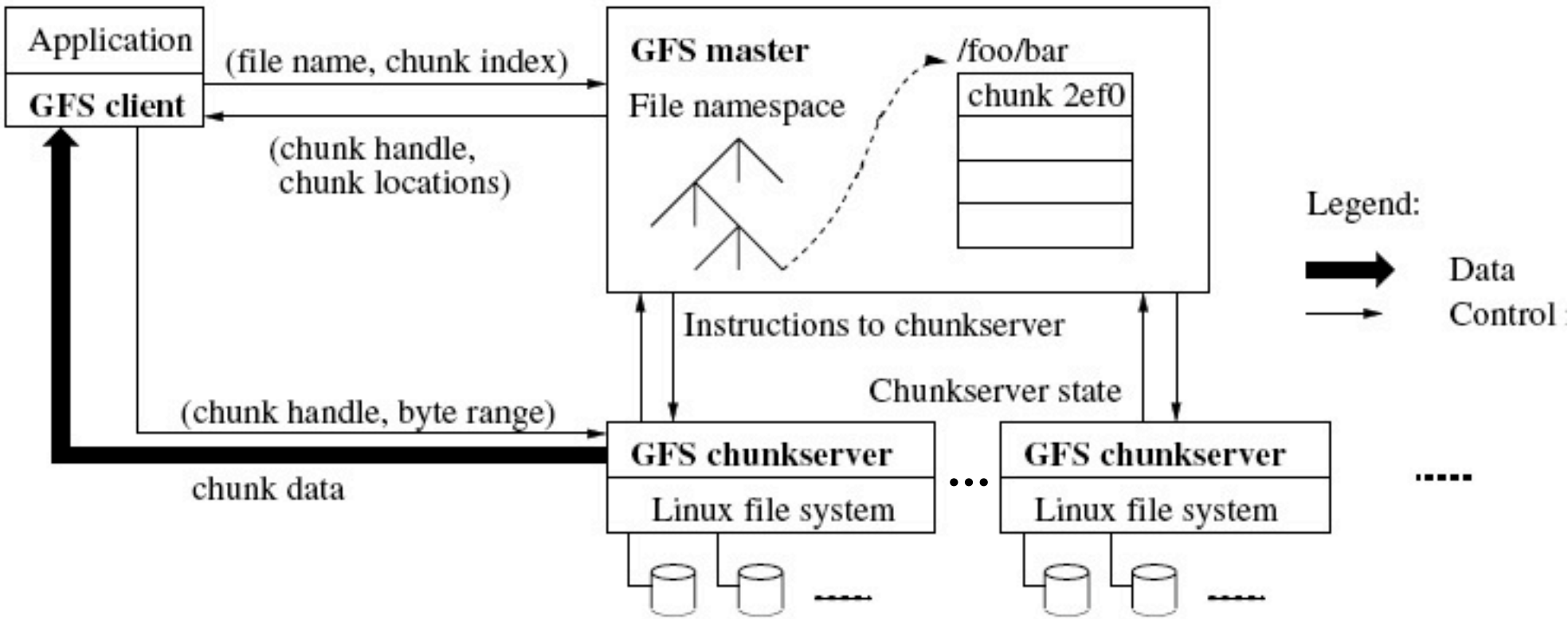


Figure 1: GFS Architecture

File Organization

- File divided into fixed size chunks
  - chunks are large (64MB)
  - 64-bit *chunk handle* (guid)
  - chunk data replicated at multiple chunk servers
  - directory metadata stored reliably (recoverably) at master

Client Read

- Client uses master only once per chunk

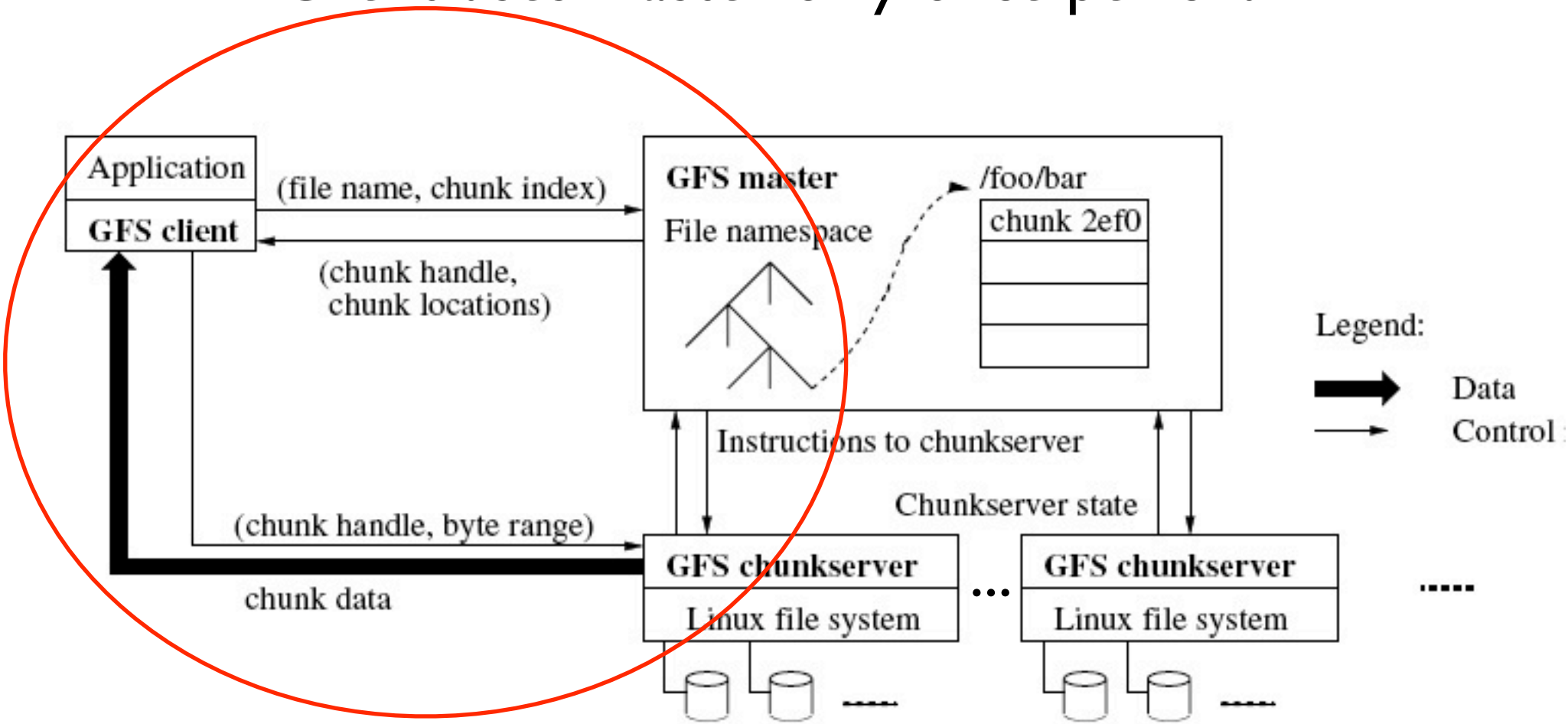


Figure 1: GFS Architecture

Metadata at Master

- File namespace: name fileid
- File composition: fileid chunkids
- Chunk location: chunkid servers

Chunk Location Metadata

- The chunk server is the final arbiter of what chunks / versions it actually has ...
  - so when master comes up it polls the chunk servers to construct in-memory chunk location map
  - then updates the map dynamically as it runs
  - if there is a disagreement, the chunk server wins
- This seems to be weakest part of design -- startup takes minutes

Namespace and File Chunk Metadata

- This is mostly read-only
- Kept in memory and recoverably on disk
- Changes are transactional!
  - Replicated snapshot and change log
  - Like DBMS journalling
  - no undo (txn rollback) required
- This adds e.g. write latency
  - most writes are large
  - group commit techniques used

Consistency Model

- *Consistent*: all clients see the same data
  - (all replicas are equal)
- *Defined*: predictable from client operations

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

Table 1: File Region State After Mutation

Replicated Writes

- Use *leases*
  - master chooses *one* chunk server to be *primary* replica
    - the primary controls the write
    - lease held for a while (lease duration)
  - same trick as master / chunk server:
    - one site has control
    - if it fails, we notice and recover state

Control Flow for Replicated Writes

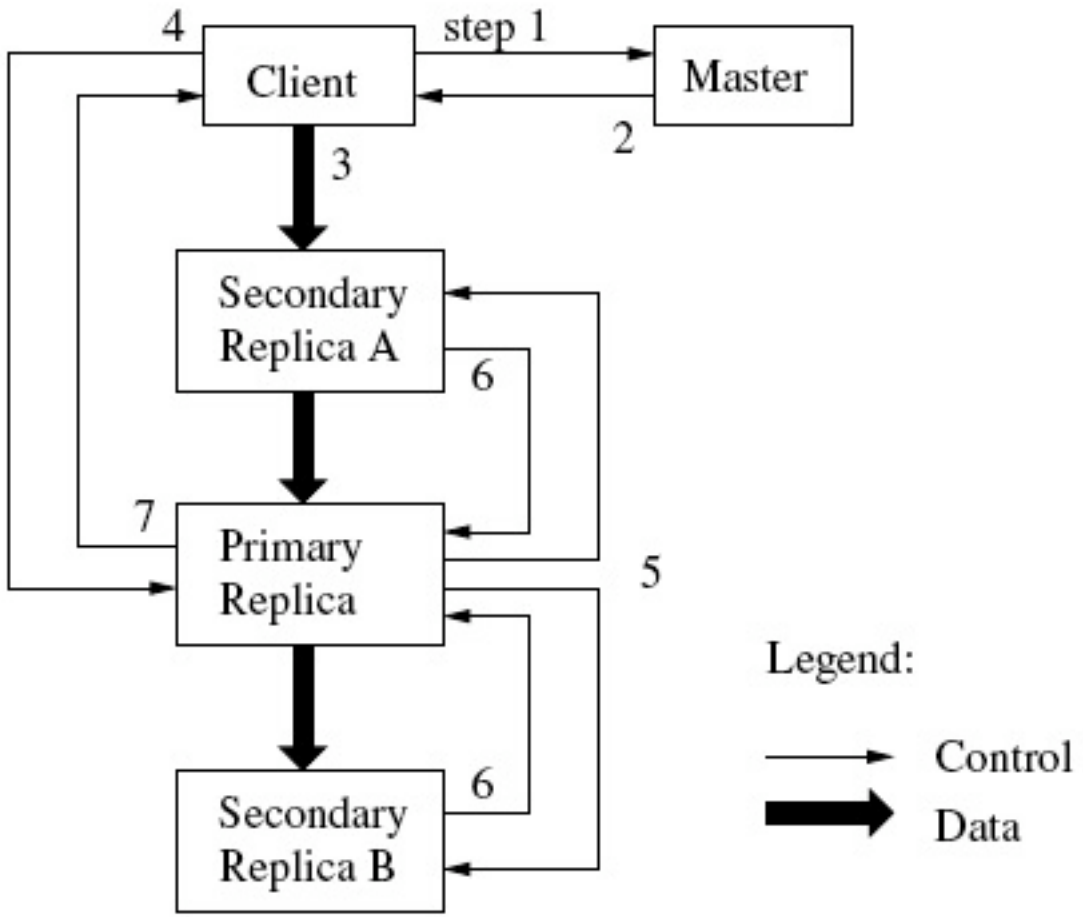


Figure 2: Write Control and Data Flow

- 1-2: Client gets primary and secondary replica list from master

Control Flow for Replicated Writes

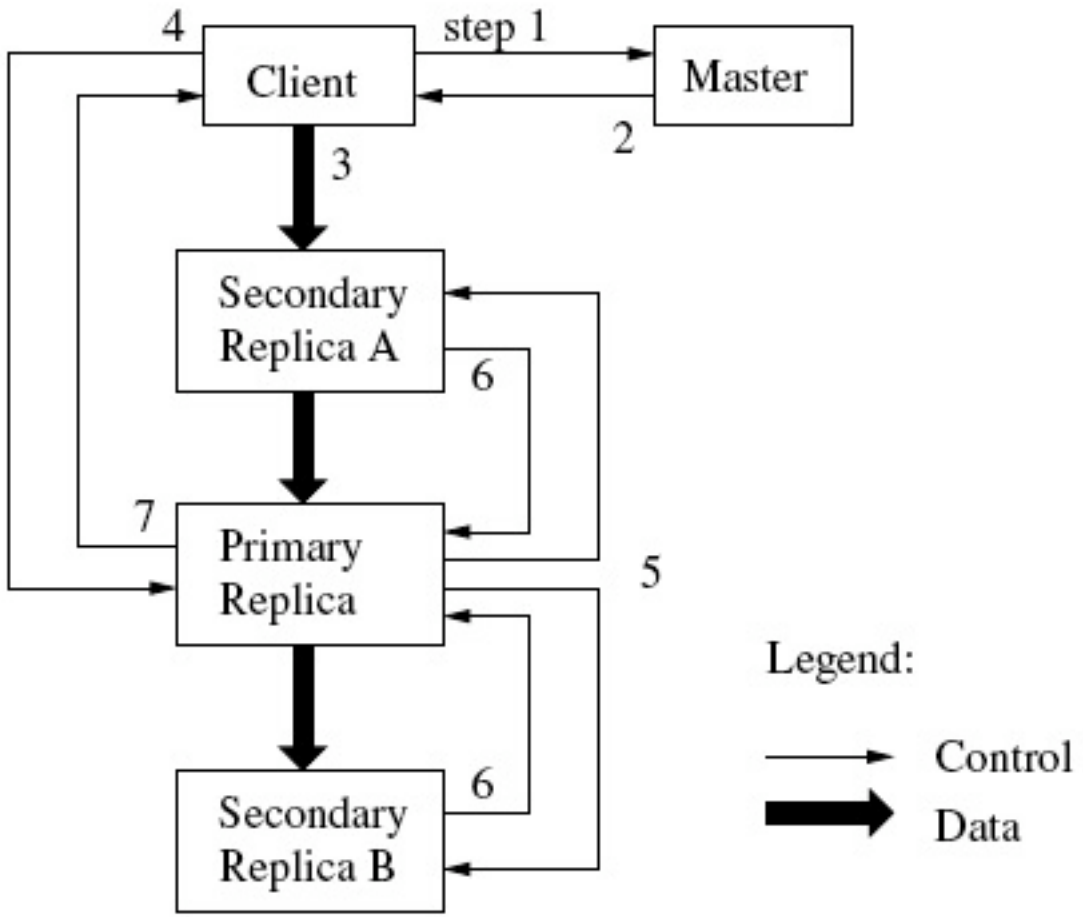


Figure 2: Write Control and Data Flow

- 3: Client pushes data to replicas in network-efficient order
- stored in LRU buffer cache

Control Flow for Replicated Writes

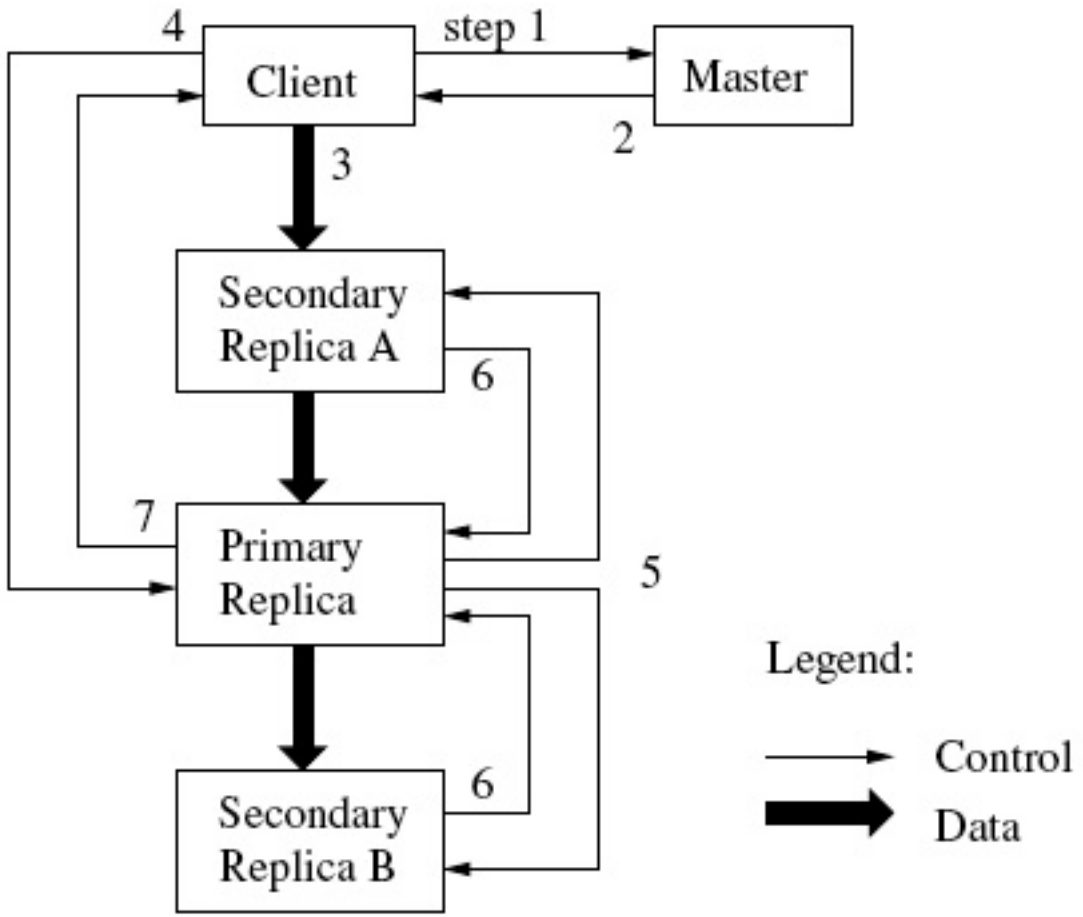


Figure 2: Write Control and Data Flow

- 4: Client sends write request to primary
- primary serializes and applies locally

Control Flow for Replicated Writes

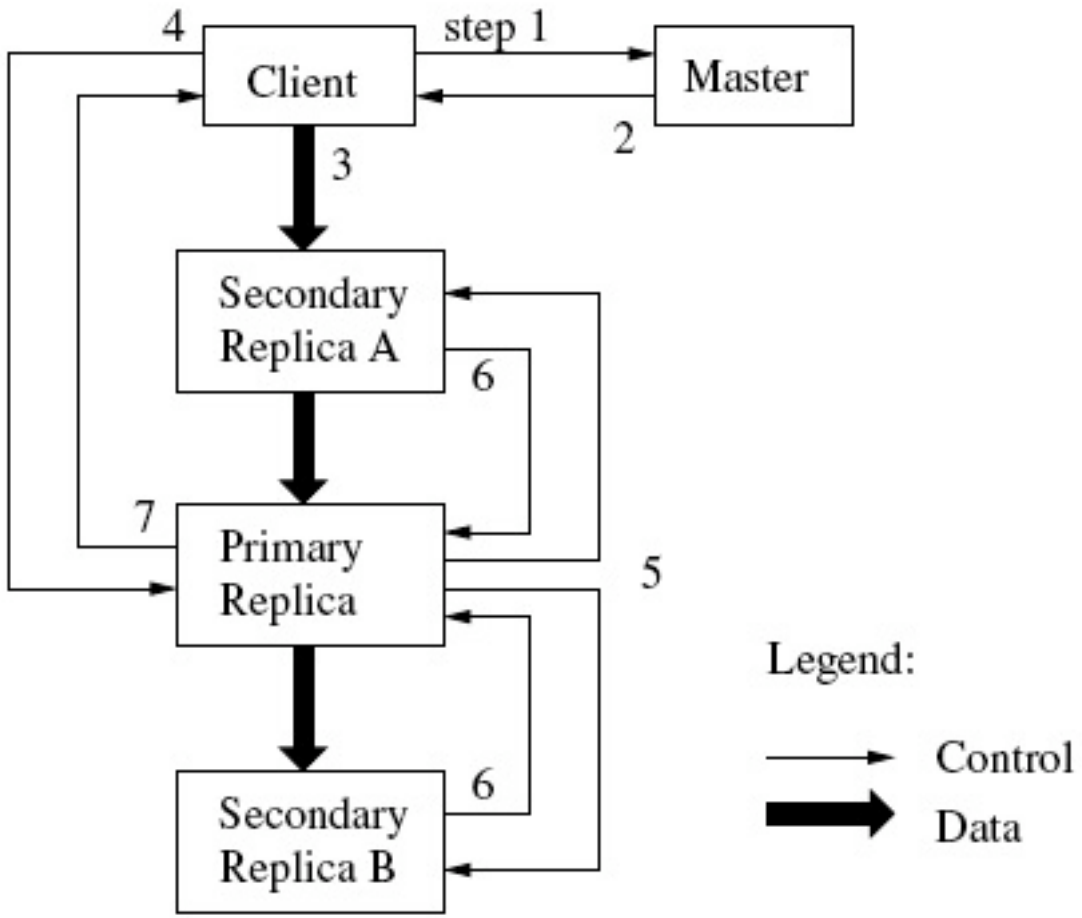


Figure 2: Write Control and Data Flow

- 5-7: primary sends write request to secondaries and waits for replies

Atomic Append?

- Similar ...
- Primary checks that data fits in current chunk of file
  - otherwise force client retry
- Failure leaves inconsistent state
- Clients deal with append-at-least-once semantics:
  - checksum
  - sequence numbers eliminate duplicates

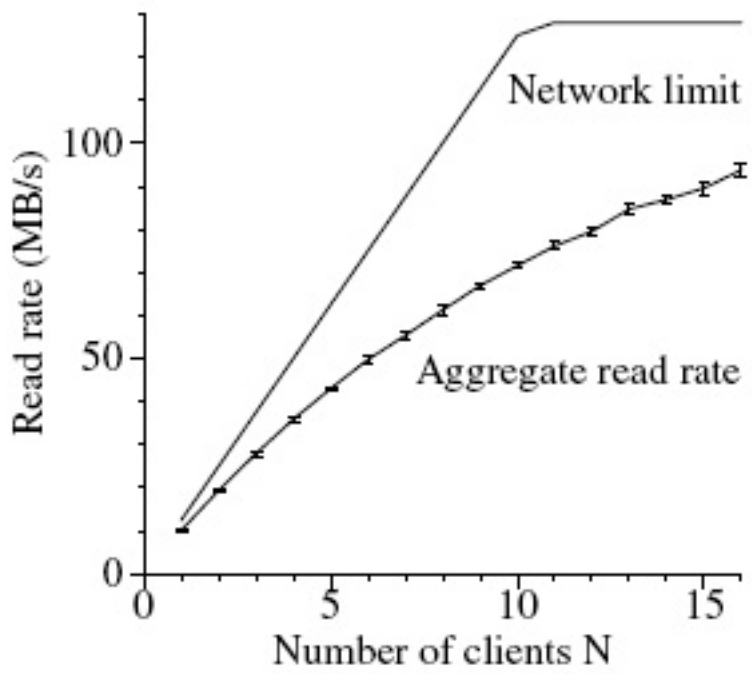
Housecleaning

- Chunk replica creation, placement, load balancing, garbage collection
  - all controlled by the master in background
  - central master greatly simplifies this part of design
- Master replication
  - a "hot standby" stays (nearly) current by "sniffing" one of the master's log replicas
- Garbage collection
  - chunk servers periodically check master

Data Integrity

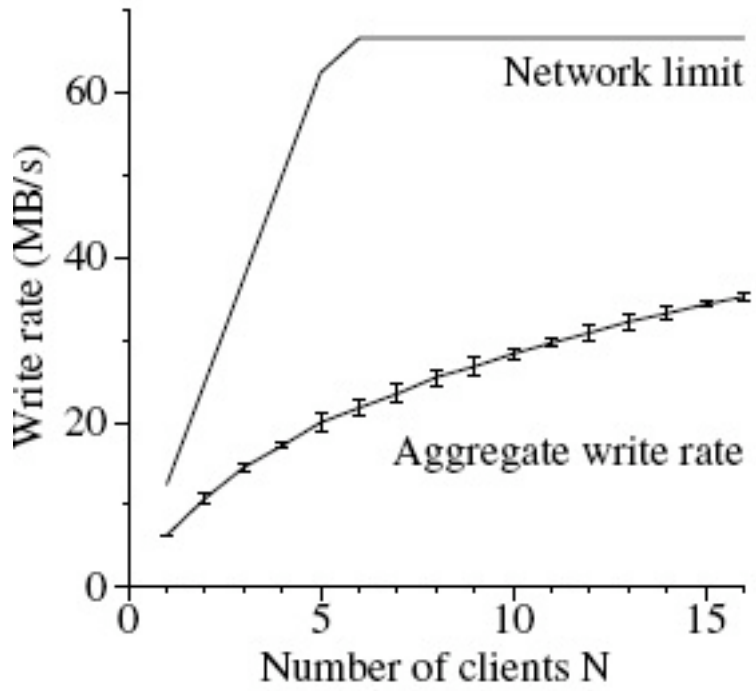
- Block (64KB) checksums maintained by chunk servers when data written
- Checksums also scanned in background by chunk servers
  - to handle infrequently referenced files
- Repair by recreating a replica
  - under control of master

Performance



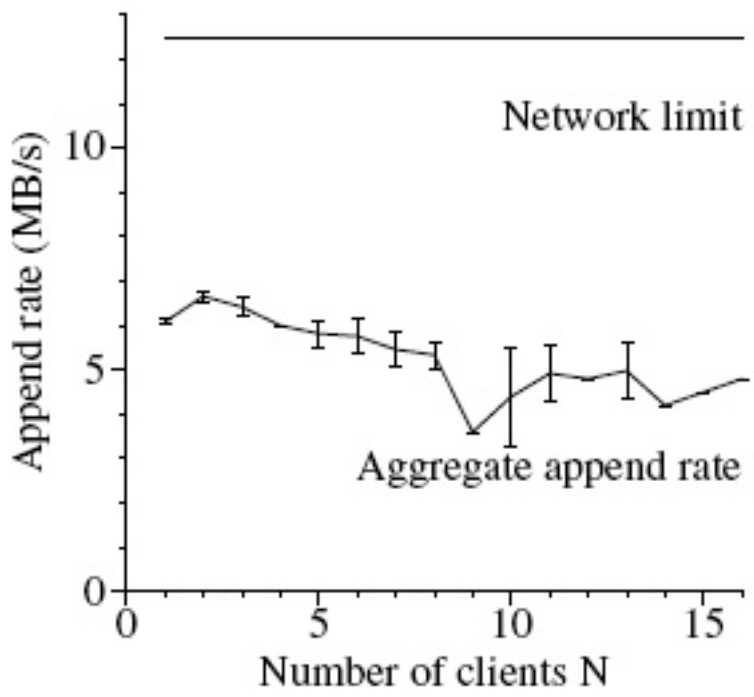
(a) Reads

Performance



(b) Writes

Performance



(c) Record appends

Performance

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

Table 3: Performance Metrics for Two GFS Clusters

Workload

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.

Workload

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.

Workload

- Operations at master

Cluster	X	Y
Open	26.1	16.3
Delete	0.7	1.5
FindLocation	64.3	65.8
FindLeaseHolder	7.8	13.4
FindMatchingFiles	0.6	2.2
All other combined	0.5	0.8

Table 6: Master Requests Breakdown by Type (%)

Other Performance Issues

- IDE Protocol Versions
  - arrgh!
- Linux fsync() performance
- Linux mmap() lock contention

Conclusions

- Reexamined traditional assumptions
  - unreliable hardware
  - few large read-mostly, append-mostly files
- Replication for performance and availability
- Software failure detection and recovery
- High aggregate throughput more important than individual client