Operating Systems (Honor Track)

File System 5: Storage and File Systems in Modern Computer Systems

Xin Jin

Spring 2023

Acknowledgments: Ion Stoica, Berkeley CS 162

Storage and File Systems in Modern Computer Systems

- IO devices: disks with dedup
  - FAST’08 Dedup
- IO: end-to-end management
  - SOSP’13 IOFlow
- Modern file systems
  - SOSP’03 GFS
- RAID and erasure coding
  - OSDI’16 EC-Cache
- File systems for distributed applications
  - SIGCOMM’01 Chord

Avoiding the Disk Bottleneck in the Data Domain Deduplication File System

Benjamin Zhu, Kai Li, Hugo Patterson

USENIX FAST 2008

Acknowledgments: Kai Li

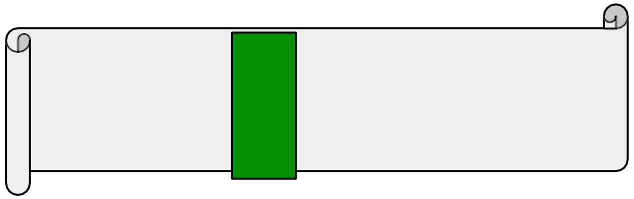
What is “Deduplication?”

- Deduplication is ***global compression*** that removes the redundant segments globally (across **many** files)

- ***Local compression*** tools (gzip, winzip, …) encode redundant strings in a small window (within a file)

Idea of Deduplication

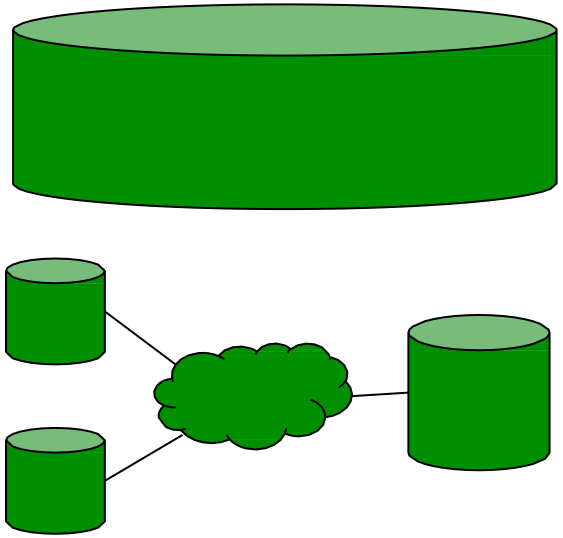
Traditional local compression



Encode a sliding window of bytes (e.g. 100K) [Ziv&Lempel77]

~2-3X compression

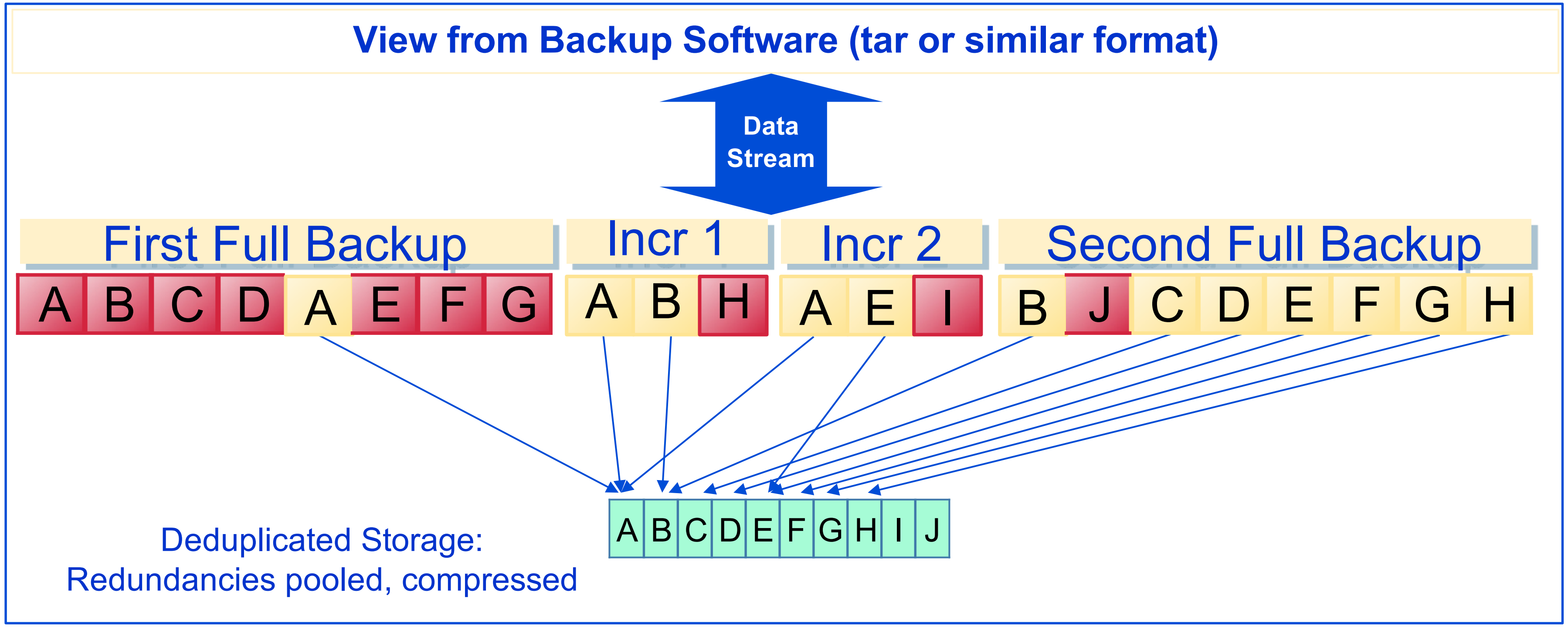
Deduplication



~10-50X compression

Large window ⇒ more redundant data

Backup Data Example



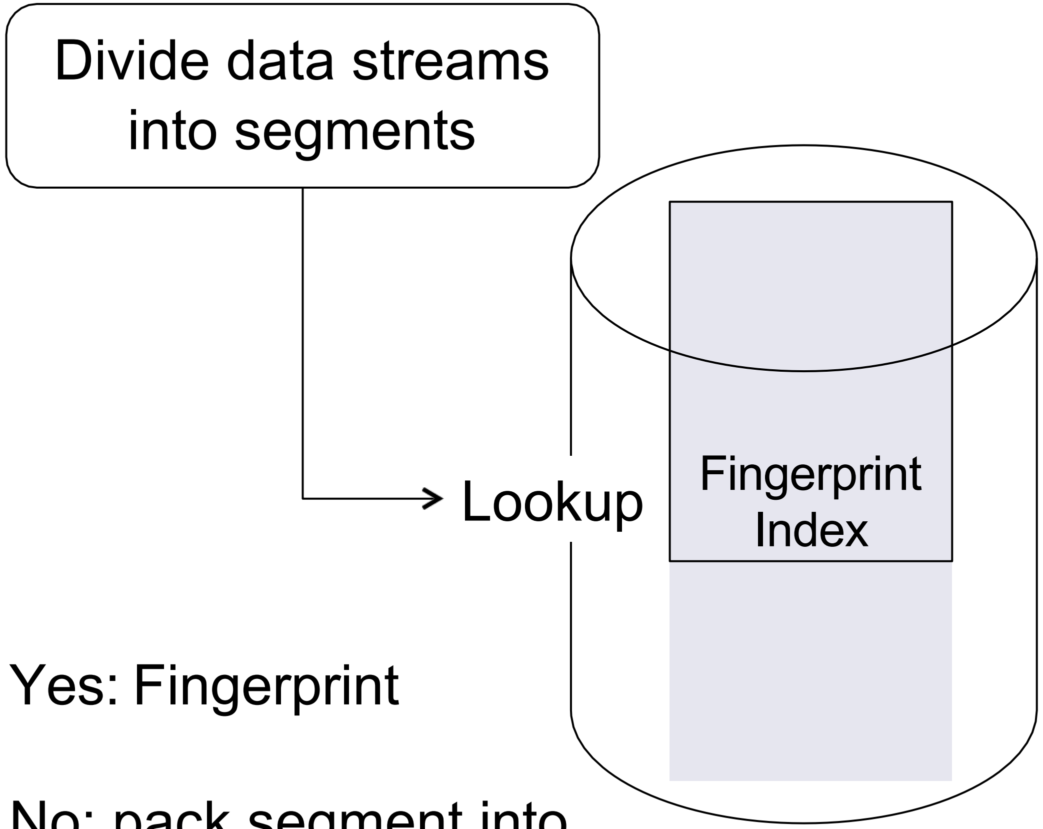
🟥 = Unique variable segments

🟨 = Redundant data segments

🟩 = Compressed unique segments

Deduplication Process

(Fingerprinting)



Yes: Fingerprint

No: pack segment into
container, apply
local compression,
write out to disk

Index size for 80TB
w/ 8KB segments
= (80TB/8KB) * 20B
= **200GB!**

High-Speed High Compression at Low HW Cost

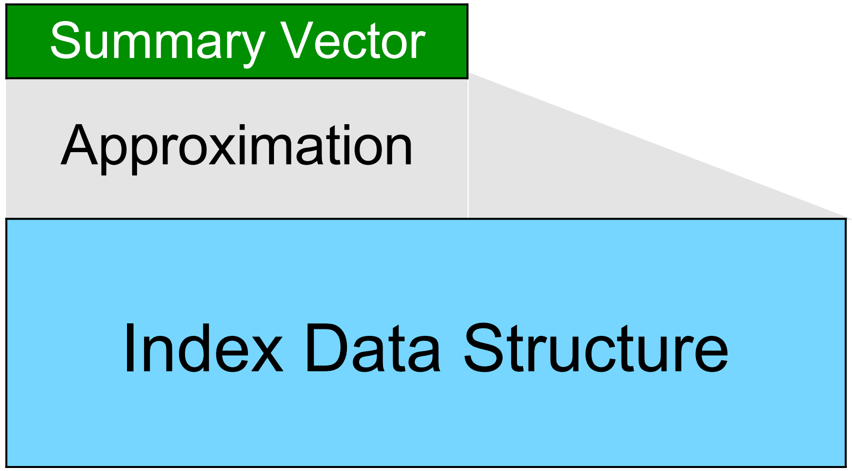
- Three techniques
  - Summary vector
  - Stream informed segment layout
  - Locality preserved caching (LPC)

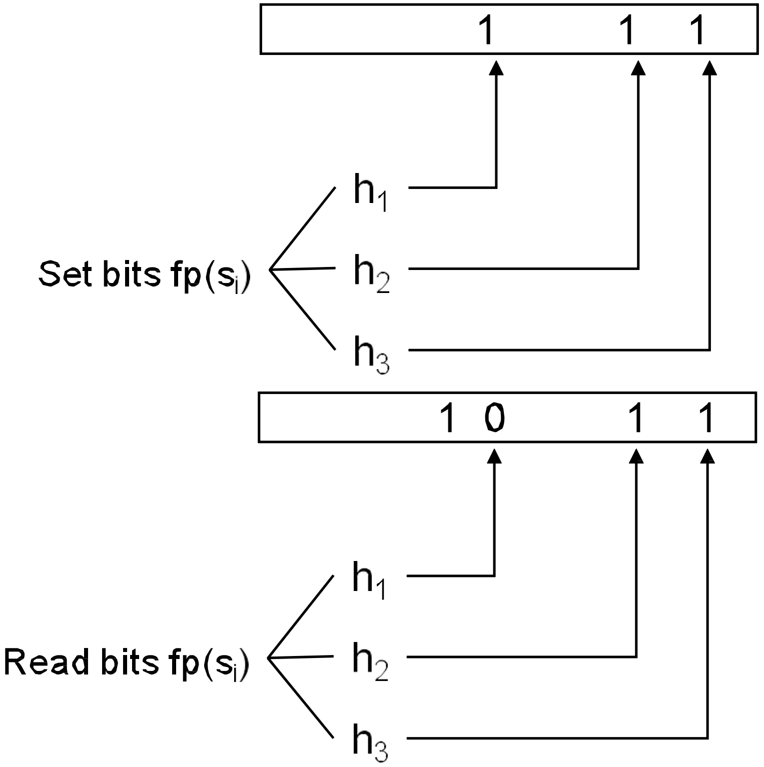
Summary Vector

Goal: Use minimal memory to test for new data

⇒ Summarize what segments have been stored, with Bloom filter (Bloom'70) in RAM

⇒ If Summary Vector says no, it's new segment

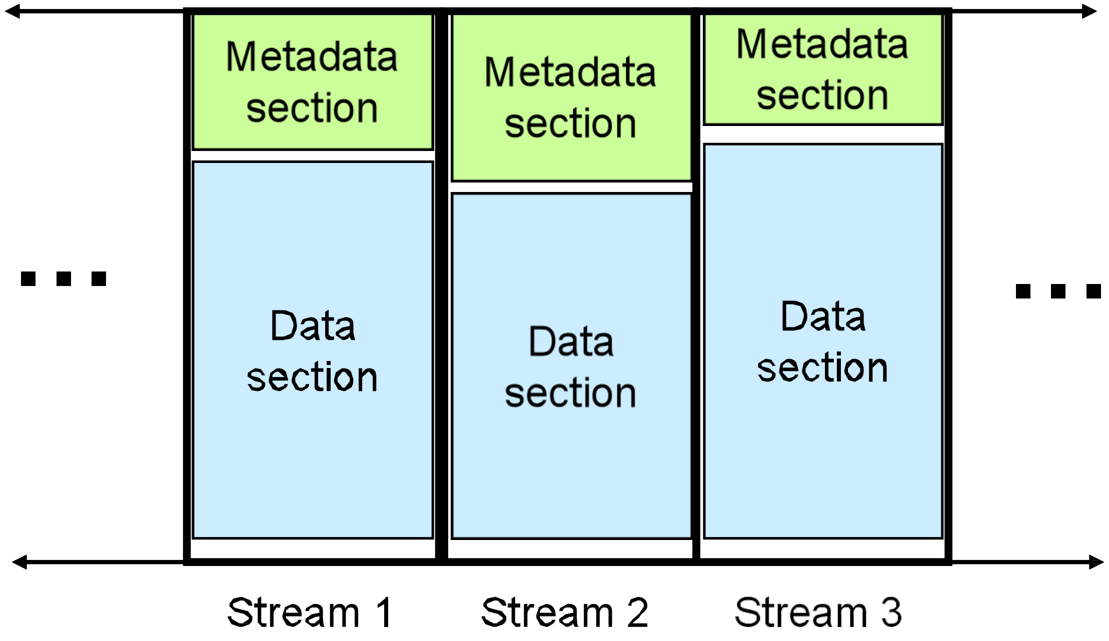




Stream Informed Segment Layout

Goal: Capture “duplicate locality” on disk

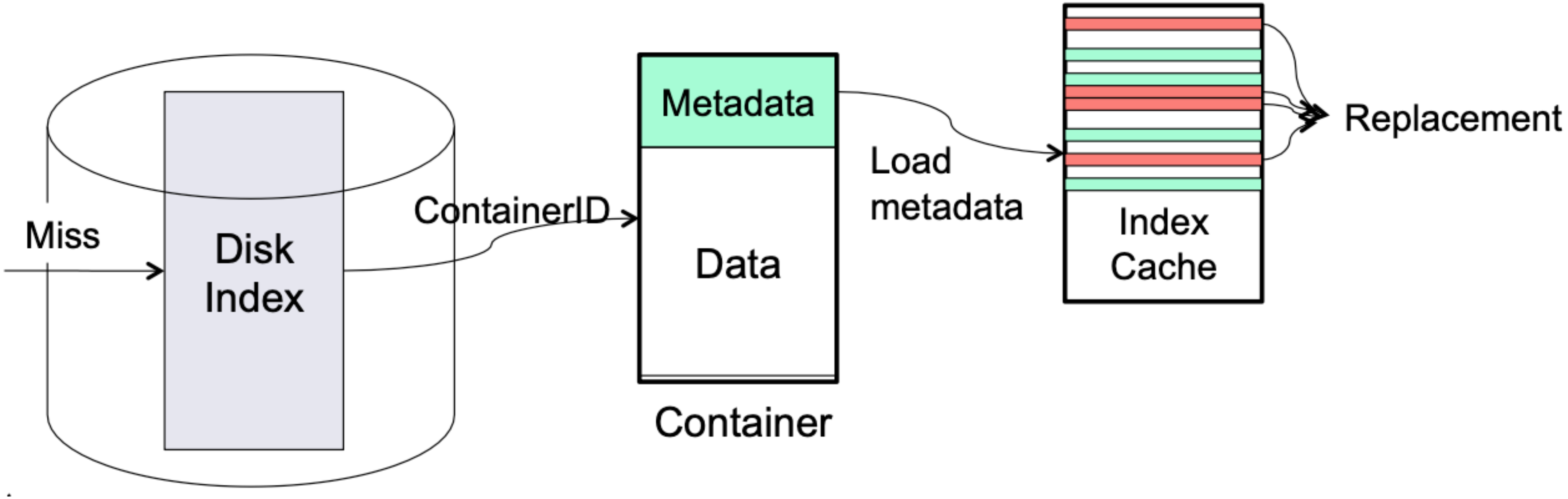
- Segments from the same stream are stored in the same “containers”
- Metadata (index data) are also in the containers



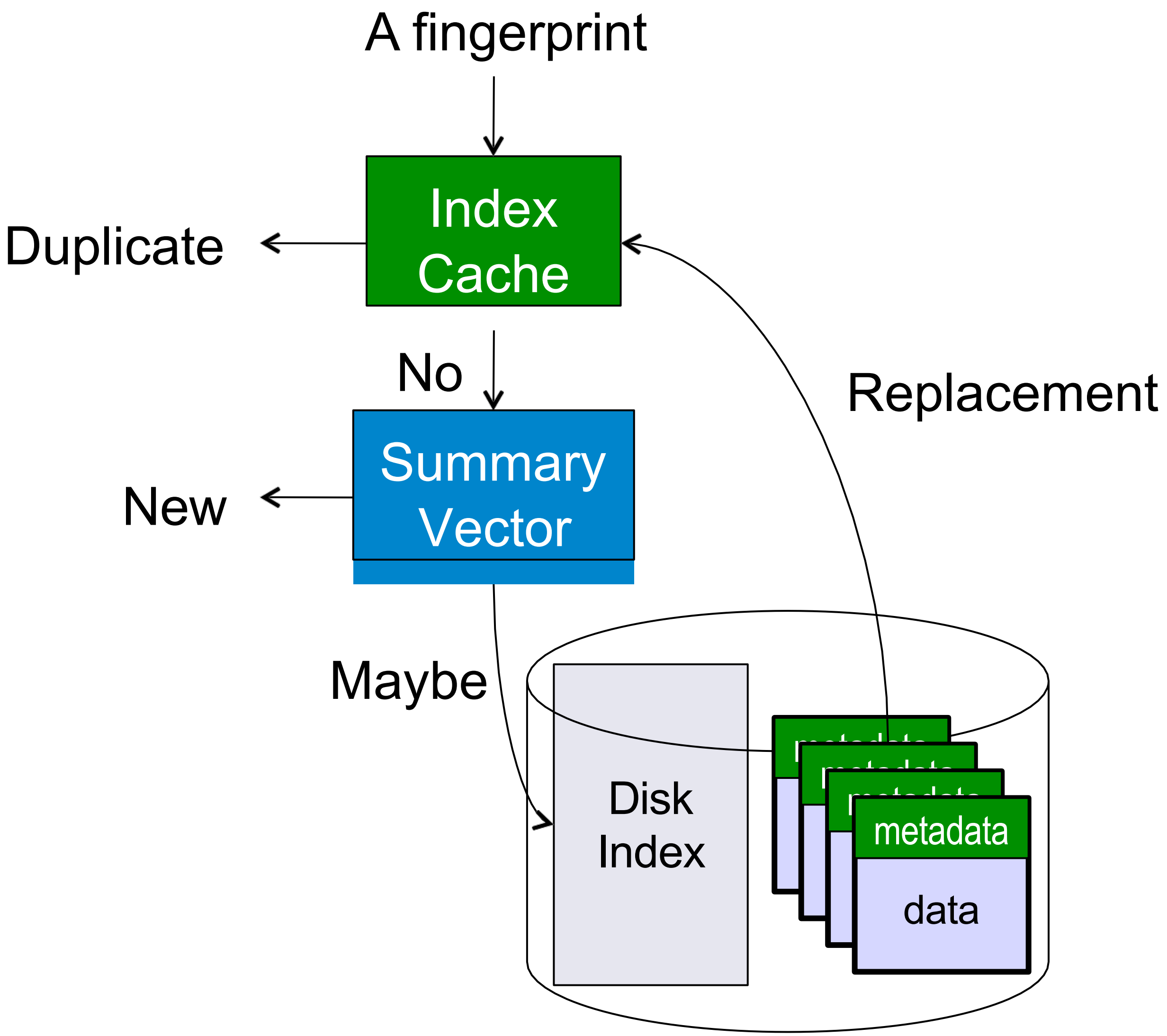
Locality Preserved Caching (LPC)

Goal: Maintain “duplicate locality” in the cache

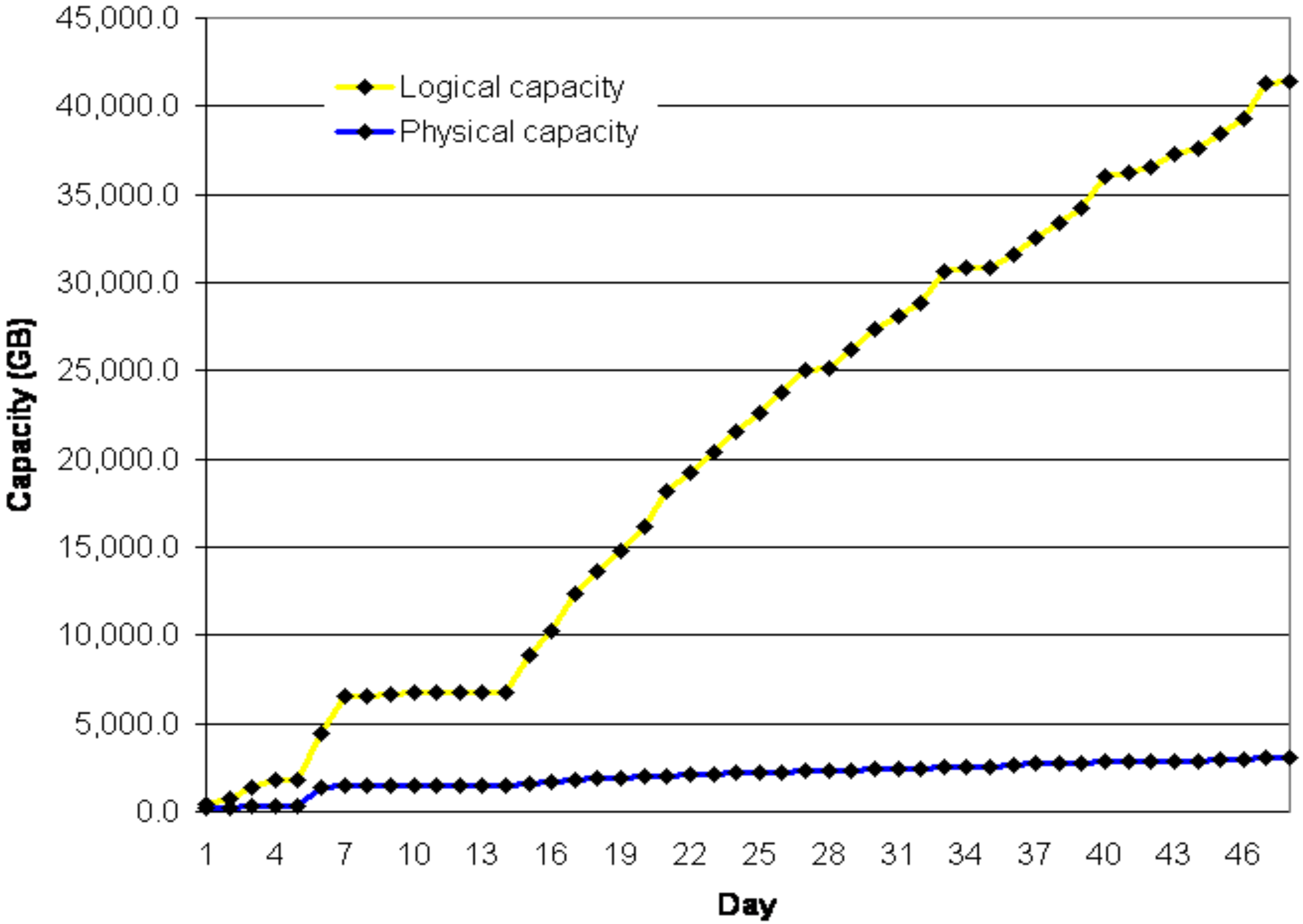
- Disk Index has all <fingerprint, containerID> pairs
- Index Cache caches a subset of such pairs
- On a miss, lookup Disk Index to find containerID
- Load the metadata of a container into Index Cache, replace if needed



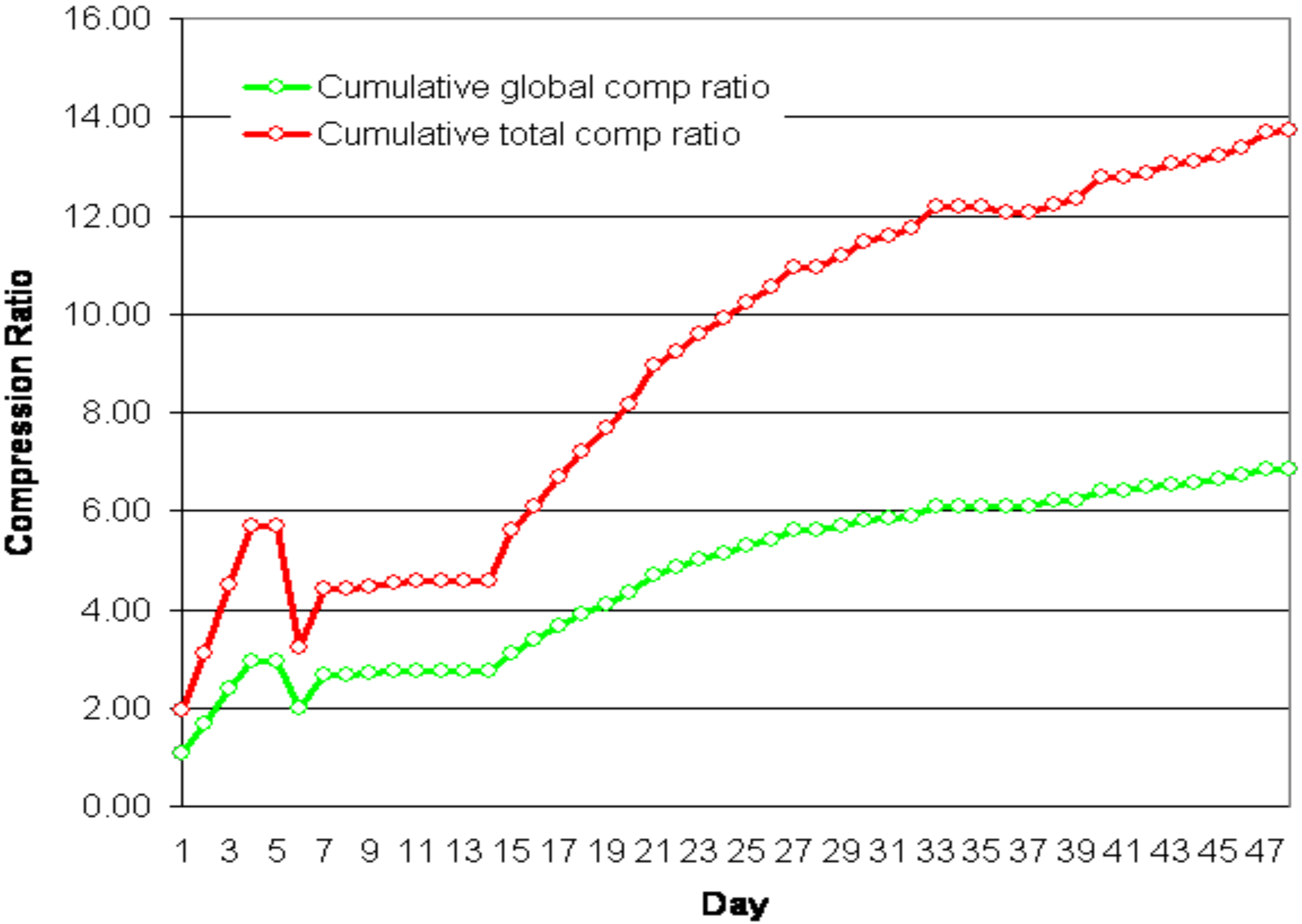
Putting Them Together



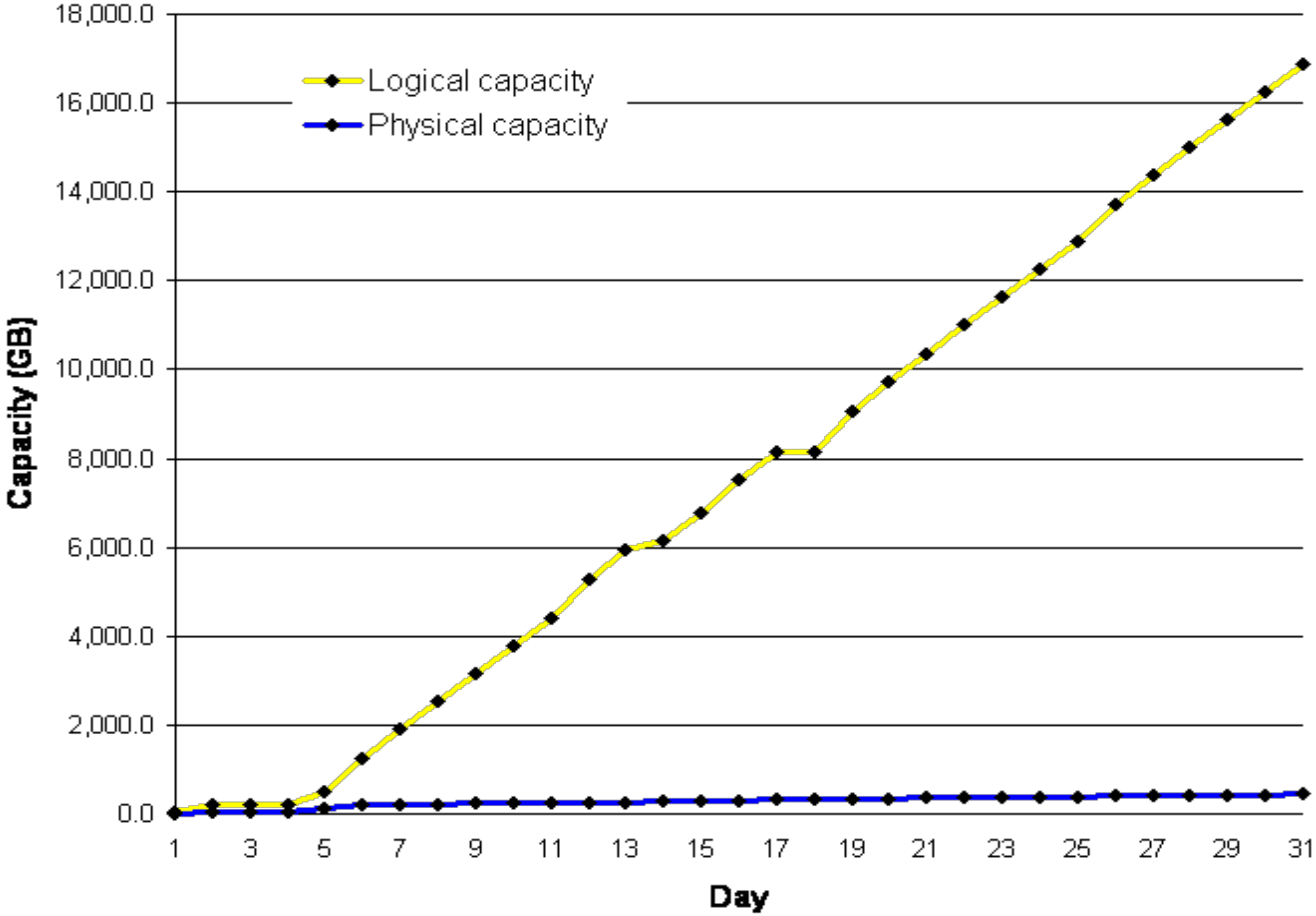
Real World Example at Datacenter A



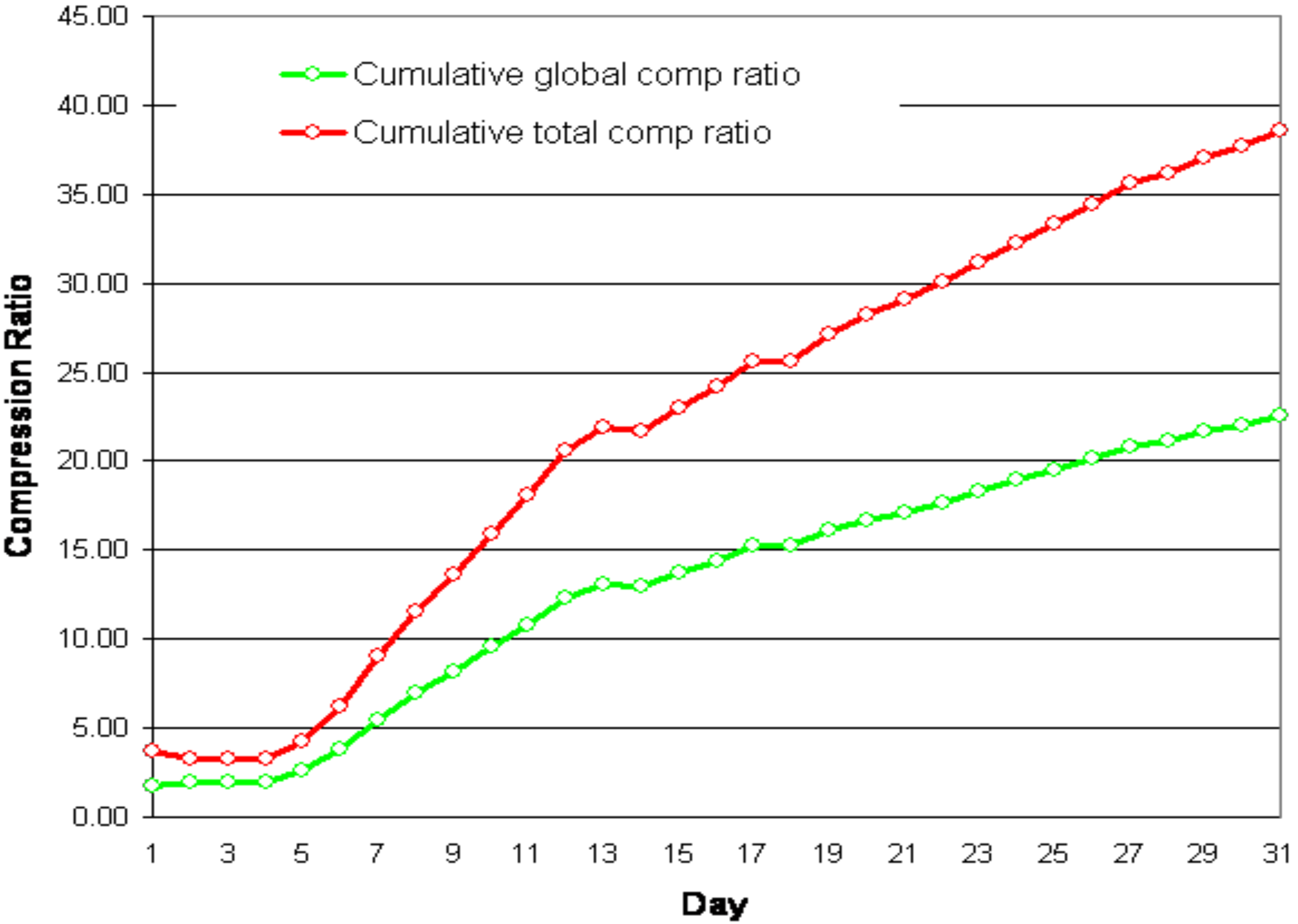
Real World Compression at Datacenter A



Real World Example at Datacenter B



Real World Compression at Datacenter B



Summary

- Deduplication removes redundant data globally
- Advanced deduplication file system
  - Has become a de facto standard to store highly redundant data because of reduction in cost, performance, power, space, …
- Three techniques to improve performance
  - Summary vector
  - Stream informed segment layout
  - Locality preserved caching (LPC)

Storage and File Systems in Modern Computer Systems

- IO devices: disks with dedup
  - FAST’08 Dedup
- IO: end-to-end management
  - SOSP’13 IOFlow
- Modern file systems
  - SOSP’03 GFS
- RAID and erasure coding
  - OSDI’16 EC-Cache
- File systems for distributed applications
  - SIGCOMM’01 Chord

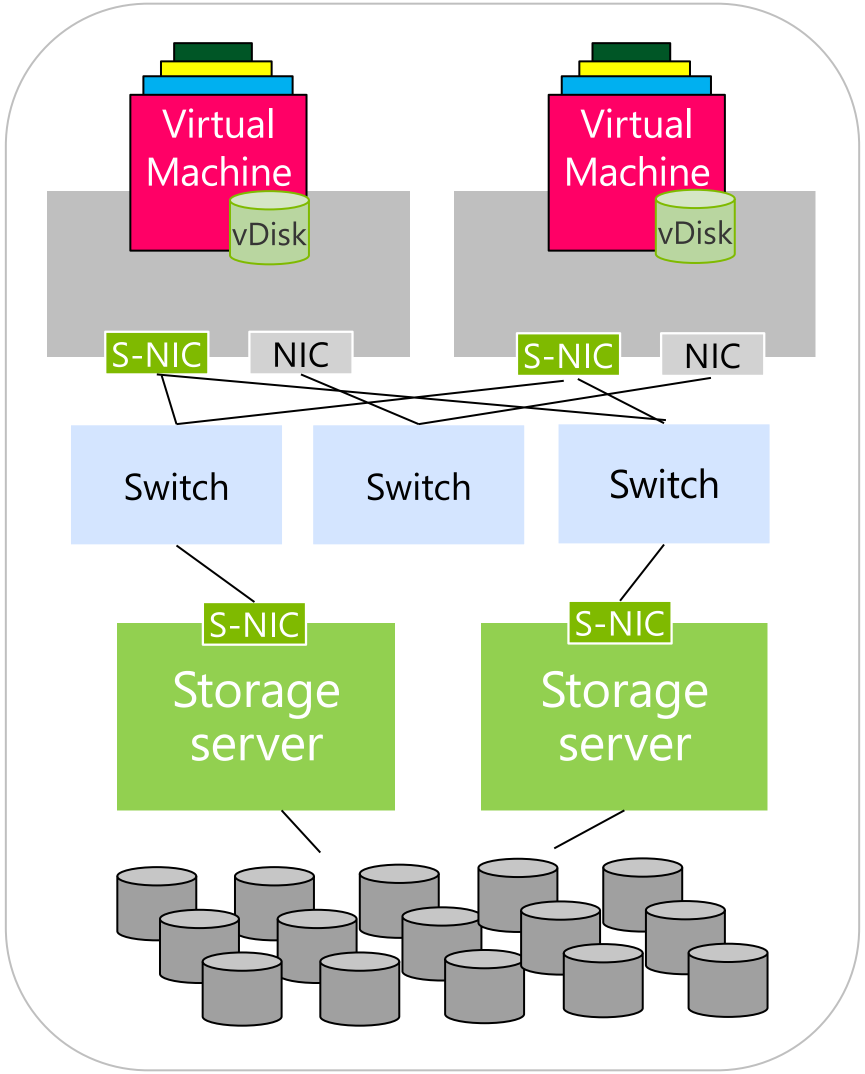
IOFlow: a Software-Defined Storage Architecture

Eno Thereska, Hitesh Ballani, Greg O’Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, Timothy Zhu

Microsoft Research

“IOFlow: A Software-Defined Storage Architecture. Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, and Timothy Zhu. In SOSP'13, Farmington, PA, USA. November 3-6, 2013. “

Background: Enterprise data centers



- General purpose applications
- Application runs on several VMs
- Separate network for VM-to-VM traffic and VM-to-Storage traffic
- Storage is virtualized
- Resources are shared

Motivation

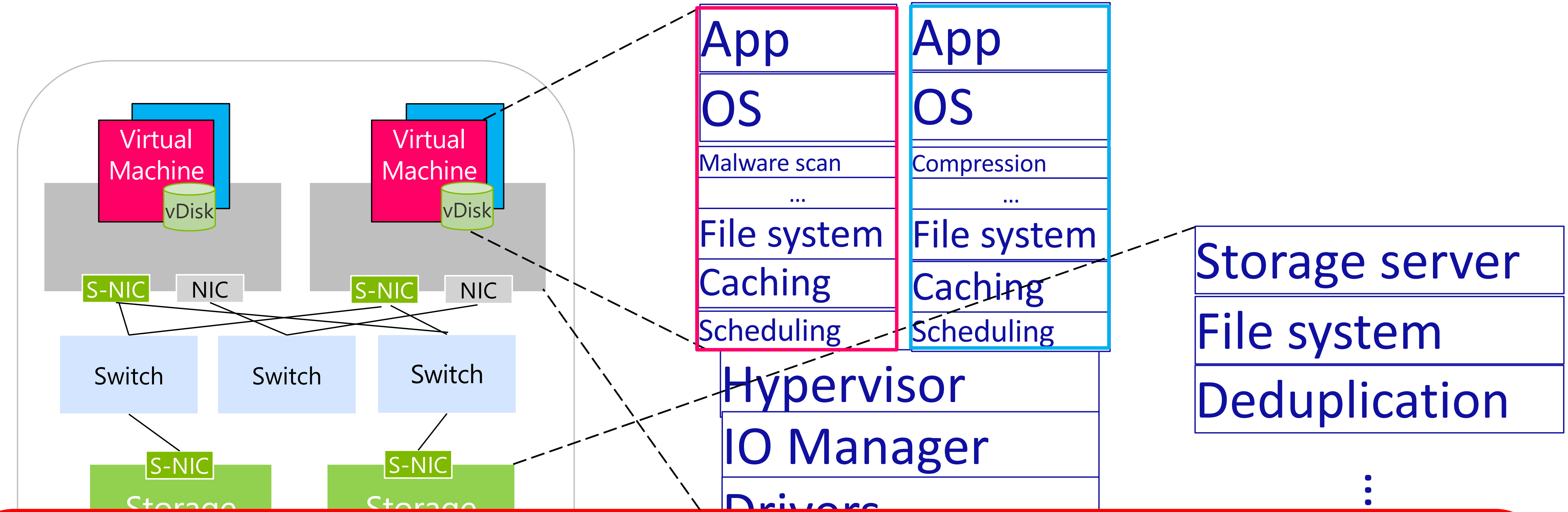
Want: predictable application behaviour and performance

Need system to provide end-to-end SLAs, e.g.,

- Guaranteed storage bandwidth B
- Guaranteed high IOPS and priority
- Per-application control over decisions along IOs’ path

It is hard to provide such SLAs today

Example: guarantee aggregate bandwidth B for Red tenant



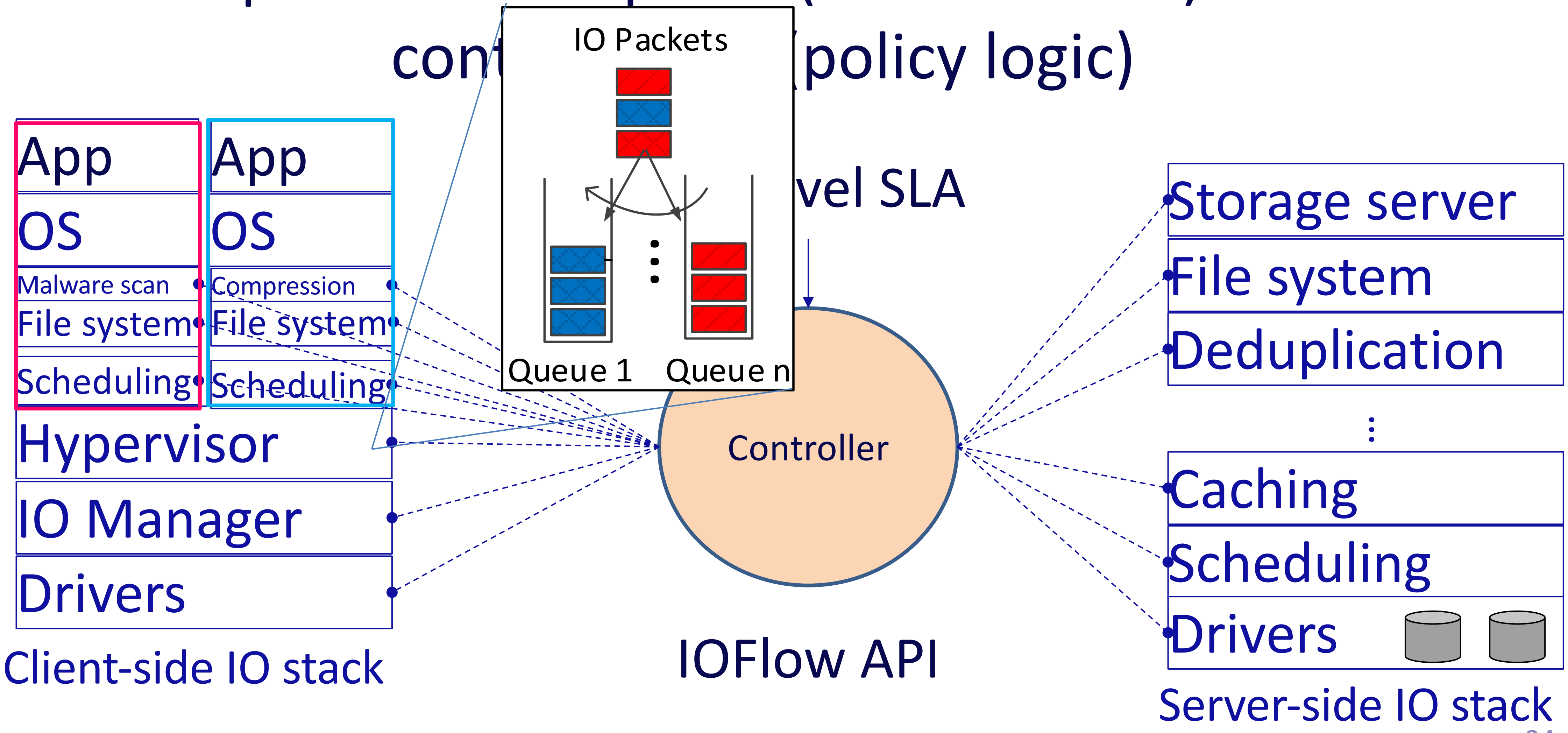
Deep IO path with 18+ different layers that are configured and operate independently and do not understand SLAs

Challenges in enforcing end-to-end SLAs

- No storage control plane
- No enforcing mechanism along storage data plane
- Aggregate performance SLAs
  - Across VMs, files and storage operations
- Want non-performance SLAs: control over IOs’ path
- Want to support unmodified applications and VMs

IOFlow architecture

Decouples the data plane (enforcement) from the cont (policy logic)



Contributions

- Defined and built storage control plane
- Controllable queues in data plane
- Interface between control and data plane (IOFlow API)
- Built centralized control applications that demonstrate power of architecture

Storage flows

Storage “Flow” refers to all IO requests to which an SLA applies

$$<\underbrace{\{\text{VMs}\}}_{\text{source set}}, \{\text{File Operations}\}, \underbrace{\{\text{Files}\}, \{\text{Shares}\}}_{\text{destination sets}}> \text{ ---> SLA}$$

- Aggregate, per-operation and per-file SLAs, e.g.,

  *<{VM 1-100}, write, *, \\share\db-log}>---> high priority*

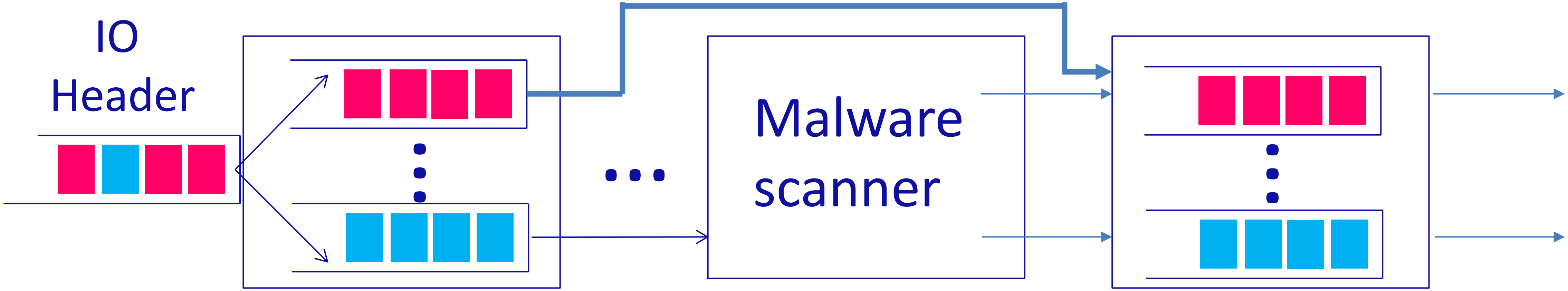
  *<{VM 1-100}, *, *, \\share\db-data}> ---> min 100,000 IOPS*

- Non-performance SLAs, e.g., path routing

  *<VM 1, *, *, \\share\dataset>---> bypass malware scanner*

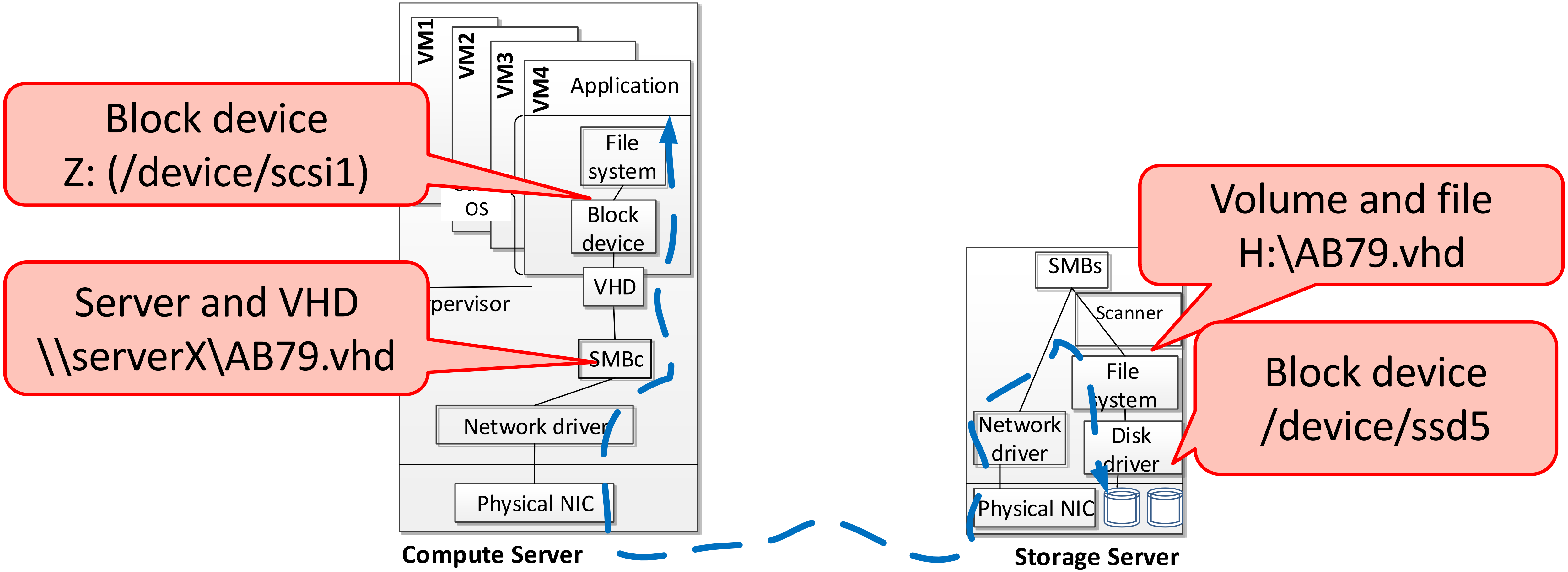
IOFlow API: programming data plane queues

1. Classification [IO Header -> *Queue*]
2. Queue servicing [Queue -> *<token rate, priority, queue size>*]
3. Routing [Queue -> *Next-hop*]



Lack of common IO Header for storage traffic

SLA: <VM 4, *, *, \\share\dataset> --> Bandwidth B



Block device
Z: (/device/scsi1)

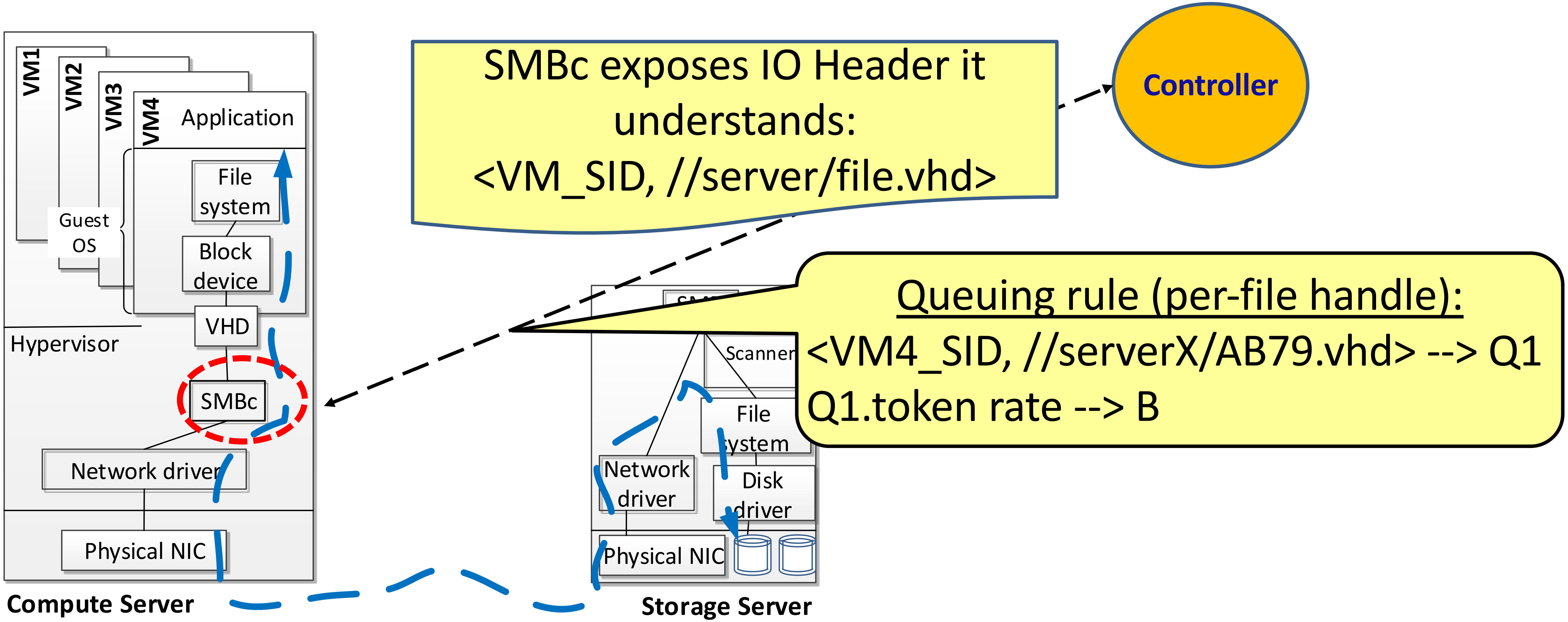
Server and VHD
\\serverX\AB79.vhd

Volume and file
H:\AB79.vhd

Block device
/device/ssd5

Flow name resolution through controller

SLA: {VM 4, *, *, //share/dataset} --> Bandwidth B



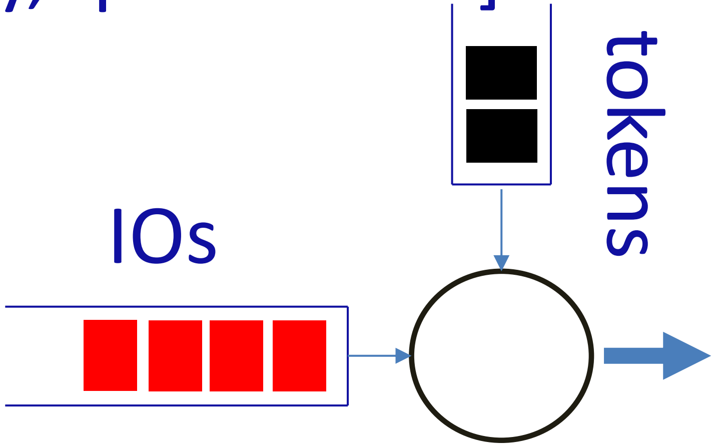
SMBc exposes IO Header it understands:
<VM_SID, //server/file.vhd>

Queuing rule (per-file handle):
<VM4_SID, //serverX/AB79.vhd> --> Q1
Q1.token rate --> B

Rate limiting for congestion control

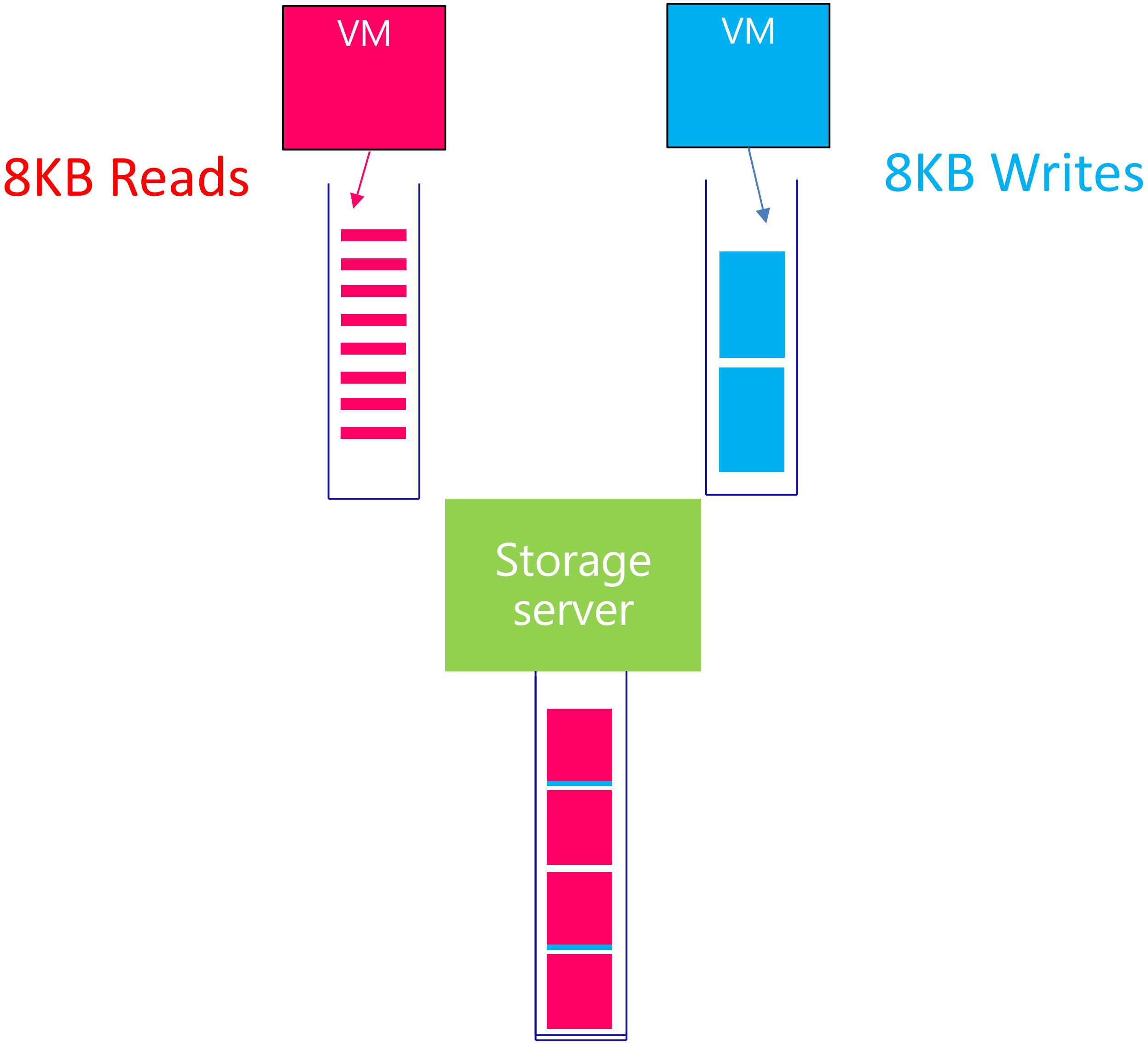
Queue servicing [Queue -> <token rate, priority, queue size>]

- Important for performance SLAs
- Today: no storage congestion control

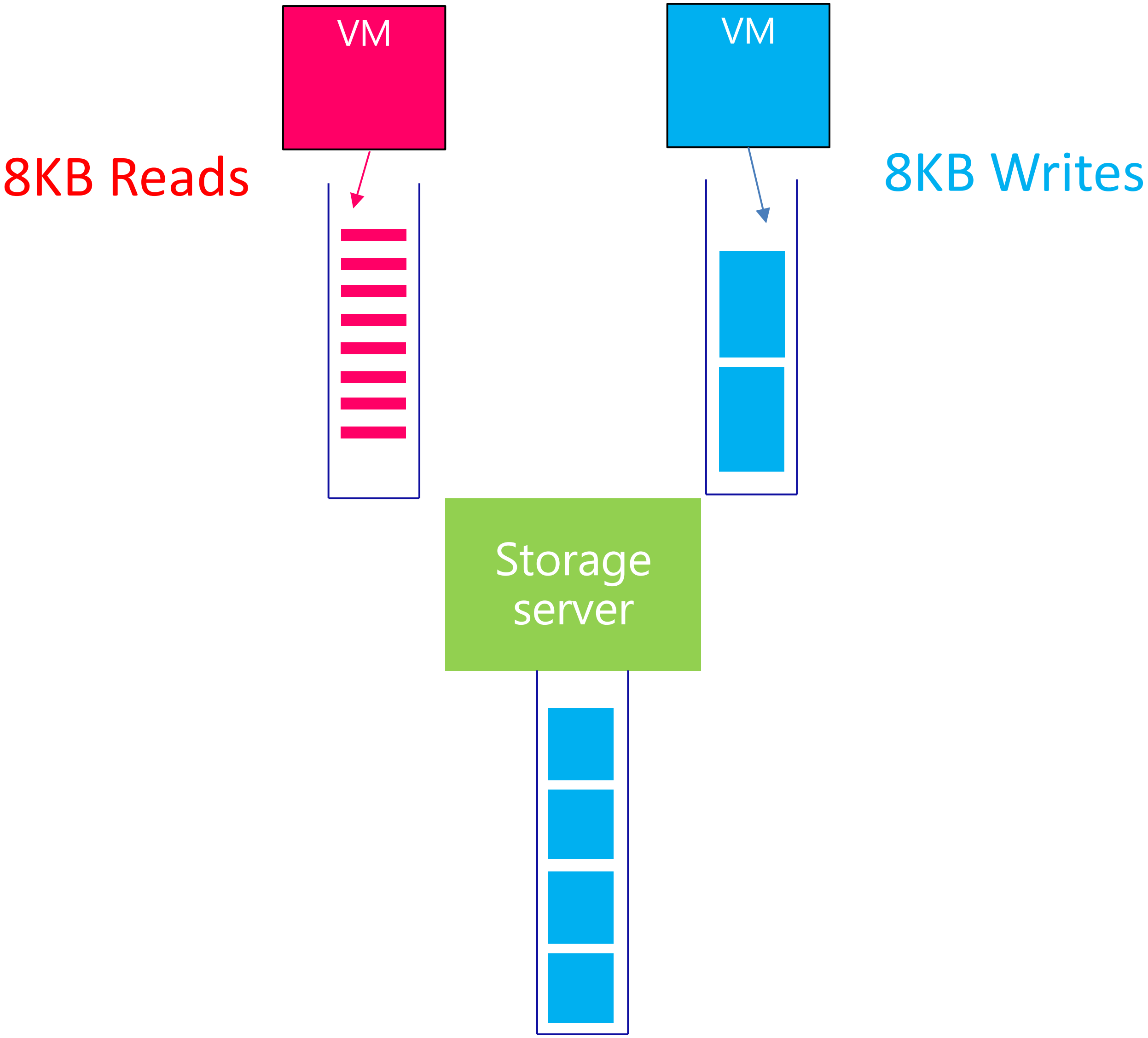


Challenging for storage: e.g., how to rate limit two VMs, one reading, one writing to get equal storage bandwidth?

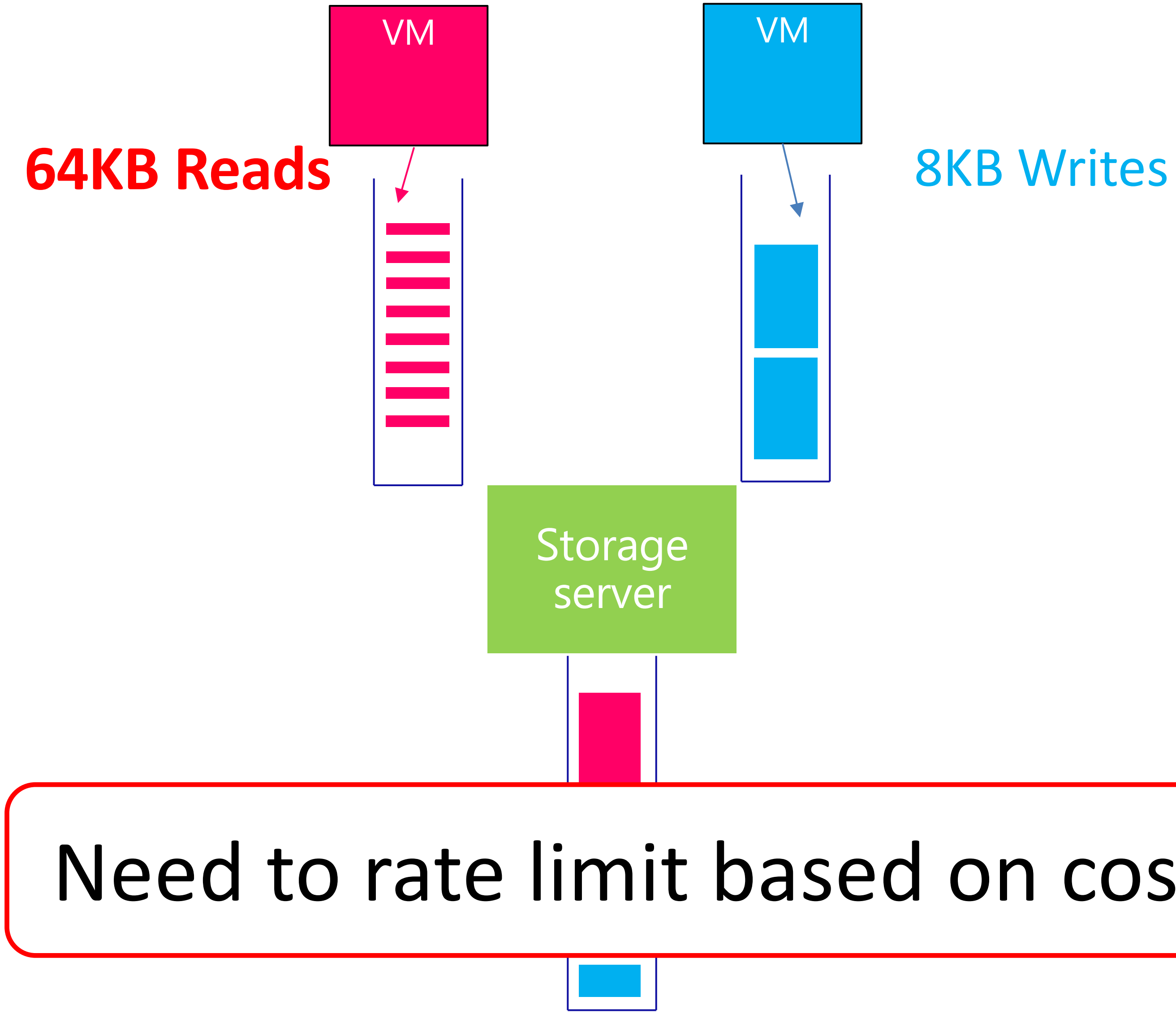
Rate limiting on payload bytes does not work



Rate limiting on bytes does not work



Rate limiting on IOPS does not work



Need to rate limit based on cost

Rate limiting based on cost

- Controller constructs empirical cost models based on device type and workload characteristics
  - RAM, SSDs, disks: read/write ratio, request size

- Cost models assigned to each queue
  - *ConfigureTokenBucket [Queue -> cost model]*

- Large request sizes split for pre-emption

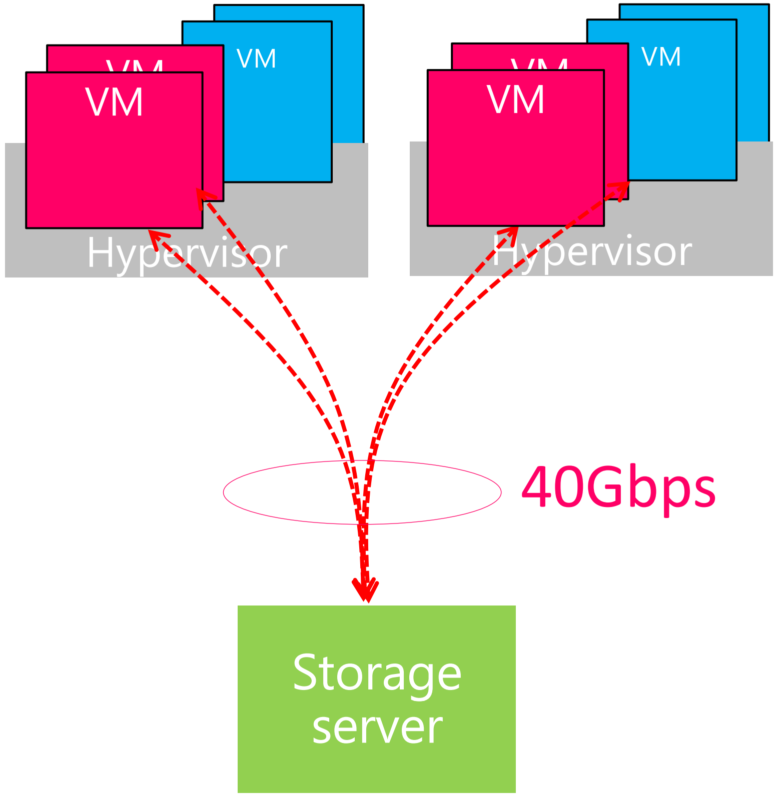
Recap: Programmable queues on data plane

- Classification [IO Header -> *Queue*]
  - Per-layer metadata exposed to controller
  - Controller out of critical path
- Queue servicing [Queue -> *<token rate, priority, queue size>*]
  - Congestion control based on operation cost
- Routing [Queue -> *Next-hop*]

How does controller enforce SLA?

Distributed, dynamic enforcement

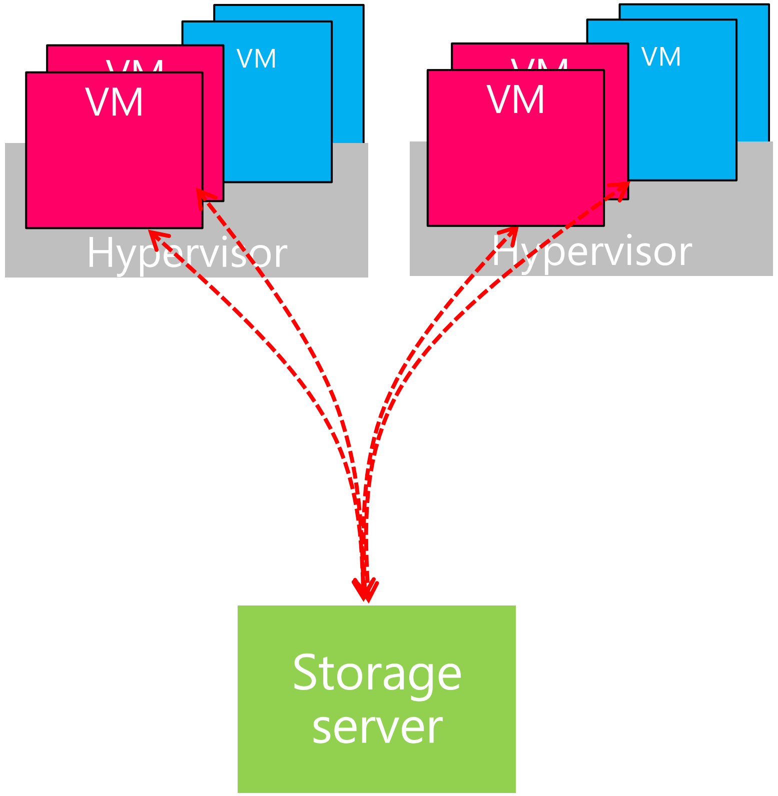
<{Red VMs 1-4}, *, * //share/dataset> --> Bandwidth 40 Gbps



- SLA needs per-VM enforcement
- Need to control the aggregate rate of VMs 1-4 that reside on different physical machines

- Static partitioning of bandwidth is sub-optimal

Work-conserving solution



- VMs with traffic demand should be able to send it as long as the aggregate rate does not exceed 40 Gbps

- **Solution:** *Max-min fair sharing*

Max-min fair sharing

Well studied problem in networks

- Existing solutions are distributed
  - Each VM varies its rate based on congestion
  - Converge to max-min sharing
- *Drawbacks*: complex and requires congestion signal

But we have a centralized controller

- Converts to simple algorithm at controller

Controller-based max-min fair sharing

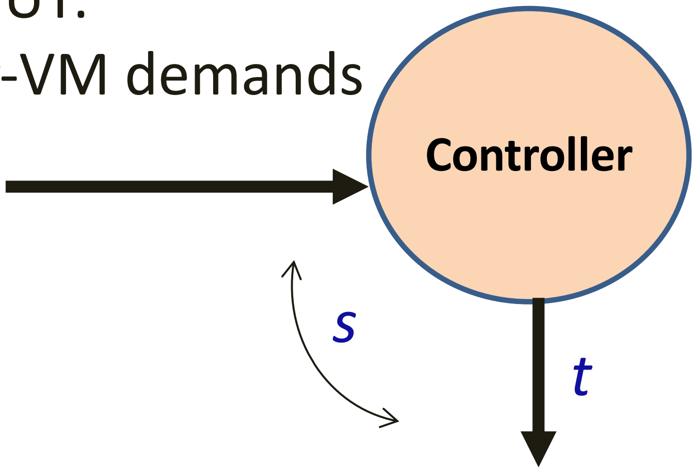
t = control interval

s = stats sampling interval

What does controller do?

- Infers VM demands
- Uses centralized max-min within a tenant and across tenants
- Sets VM token rates
- Chooses best place to enforce

INPUT:
per-VM demands



Controller

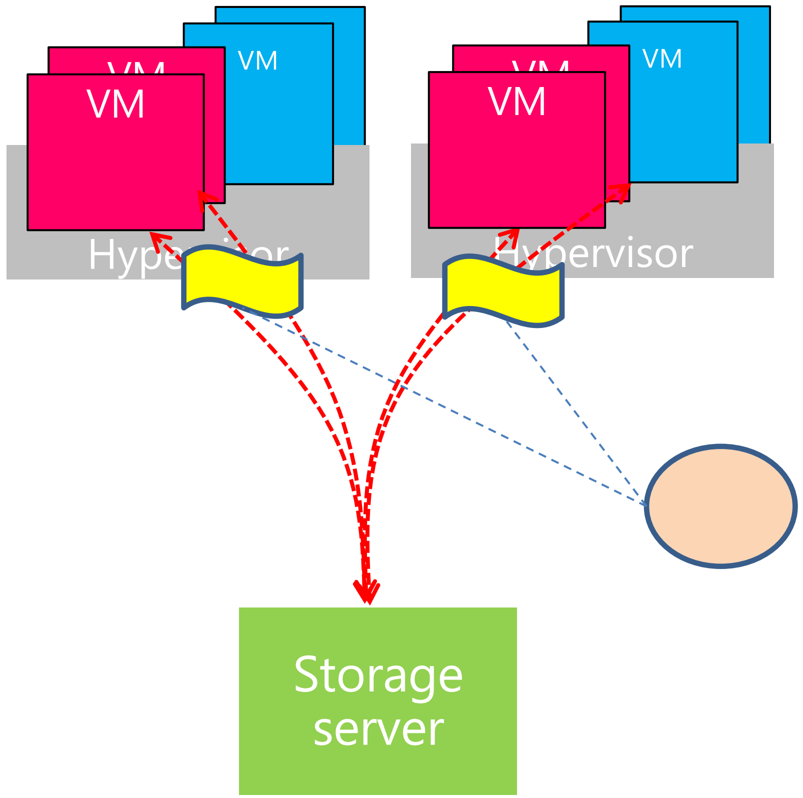
s

t

OUTPUT:
per-VM allocated token rate

Controller decides *where* to enforce

Minimize # times IO is queued and distribute rate limiting load



<u>SLA constraints</u>

- Queues where resources shared
- Bandwidth enforced close to source
- Priority enforced end-to-end

<u>Efficiency considerations</u>

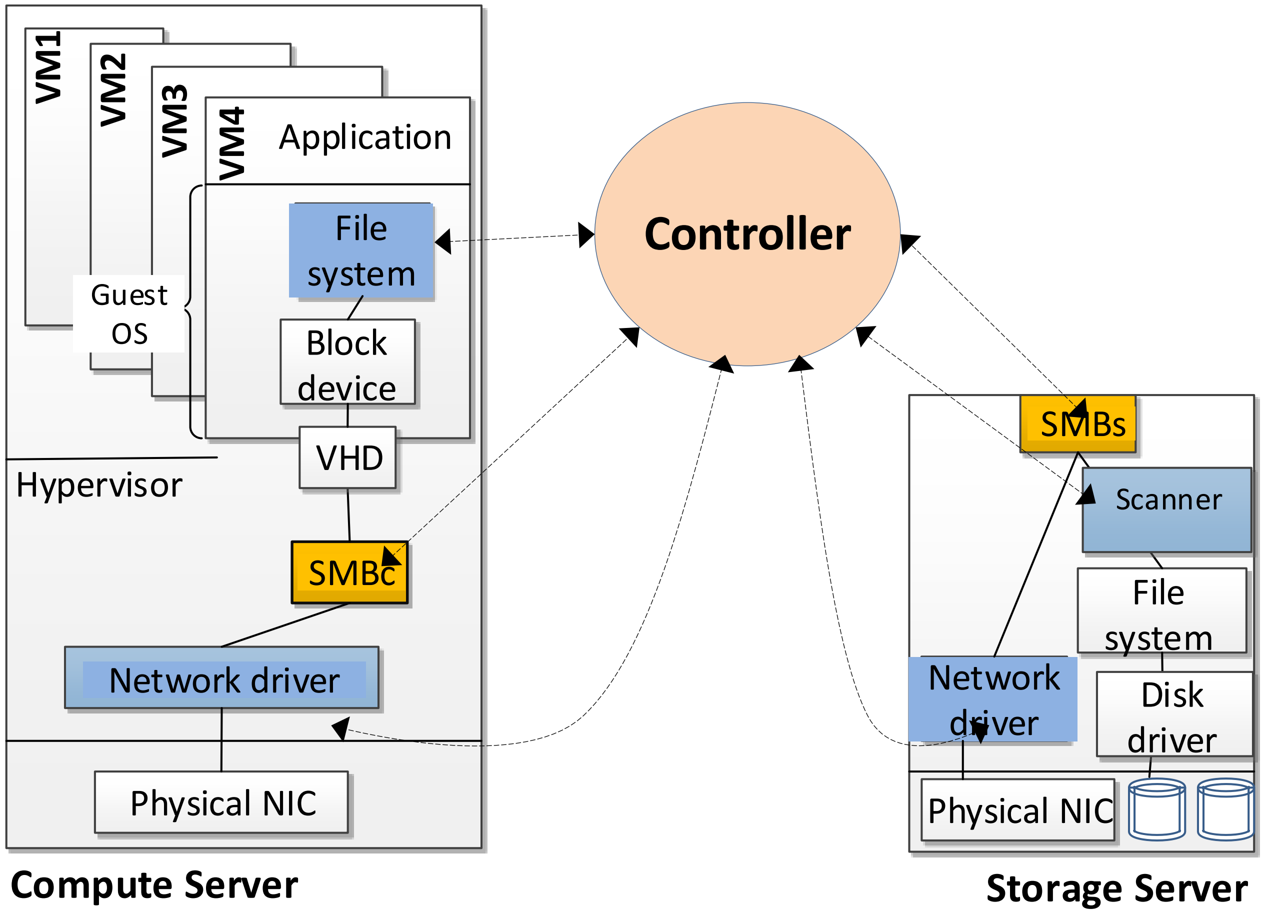
- Overhead in data plane ~ # queues
- Important at 40+ Gbps

Centralized vs. decentralized control

Centralized controller in SDS allows for simple algorithms that focus on SLA enforcement and ***not*** on distributed system challenges

Analogous to benefits of centralized control in software-defined networking (SDN)

IOFlow implementation



🟨 2 key layers for VM-to-Storage performance SLAs

4 other layers 🟦

. Scanner driver (routing)

. User-level (routing)

. Network driver

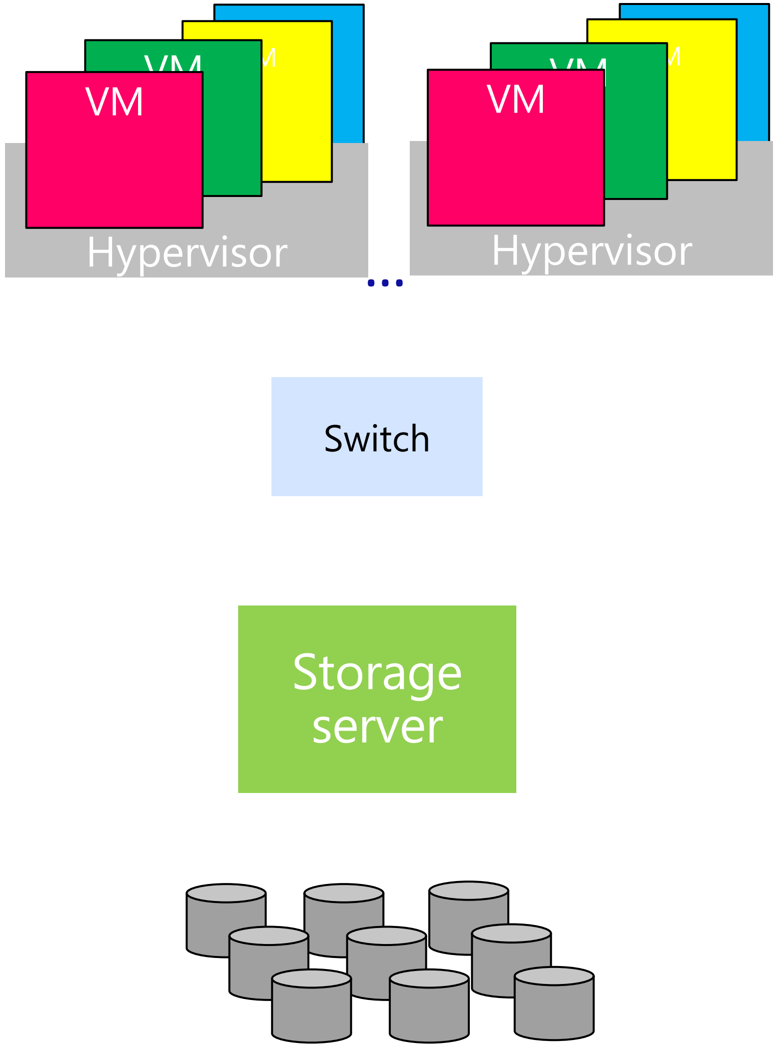
. Guest OS file system

Implemented as filter drivers on top of layers

Evaluation map

- IOFlow's ability to enforce end-to-end SLAs
  - Aggregate bandwidth SLAs
  - Priority SLAs and routing application in paper
- Performance of data and control planes

Evaluation setup



Clients:10 hypervisor servers, 12 VMs each

4 tenants (Red, Green, Yellow, Blue)

30 VMs/tenant, 3 VMs/tenant/server

Storage network:

Mellanox 40Gbps RDMA RoCE full-duplex

1 storage server:

16 CPUs, 2.4GHz (Dell R720)

SMB 3.0 file server protocol

3 types of backend: RAM, SSDs, Disks

Controller: 1 separate server

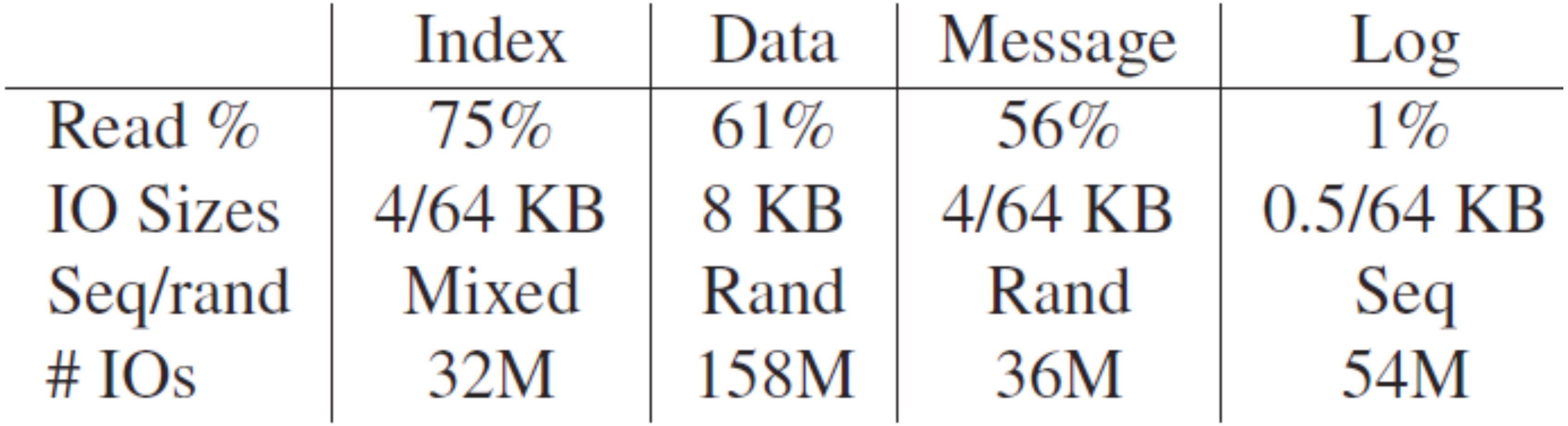
1 sec control interval (configurable)

Workloads

- 4 Hotmail tenants {Index, Data, Message, Log}

  Used for trace replay on SSDs (see paper)

- IoMeter is parametrized with Hotmail tenant characteristics (read/write ratio, request size)



	Index	Data	Message	Log
Read %	75%	61%	56%	1%
IO Sizes	4/64 KB	8 KB	4/64 KB	0.5/64 KB
Seq/rand	Mixed	Rand	Rand	Seq
# IOs	32M	158M	36M	54M

Enforcing bandwidth SLAs

4 tenants with different storage bandwidth SLAs

Tenant	SLA
Red	{VM1 – 30} -> Min 800 MB/s
Green	{VM31 – 60} -> Min 800 MB/s
Yellow	{VM61 – 90} -> Min 2500 MB/s
Blue	{VM91 – 120} -> Min 1500 MB/s

Tenants have different workloads

- **Red** tenant is aggressive: generates more requests/second

Things to look for

Distributed enforcement across 4 competing tenants

- Aggressive tenant(s) under control

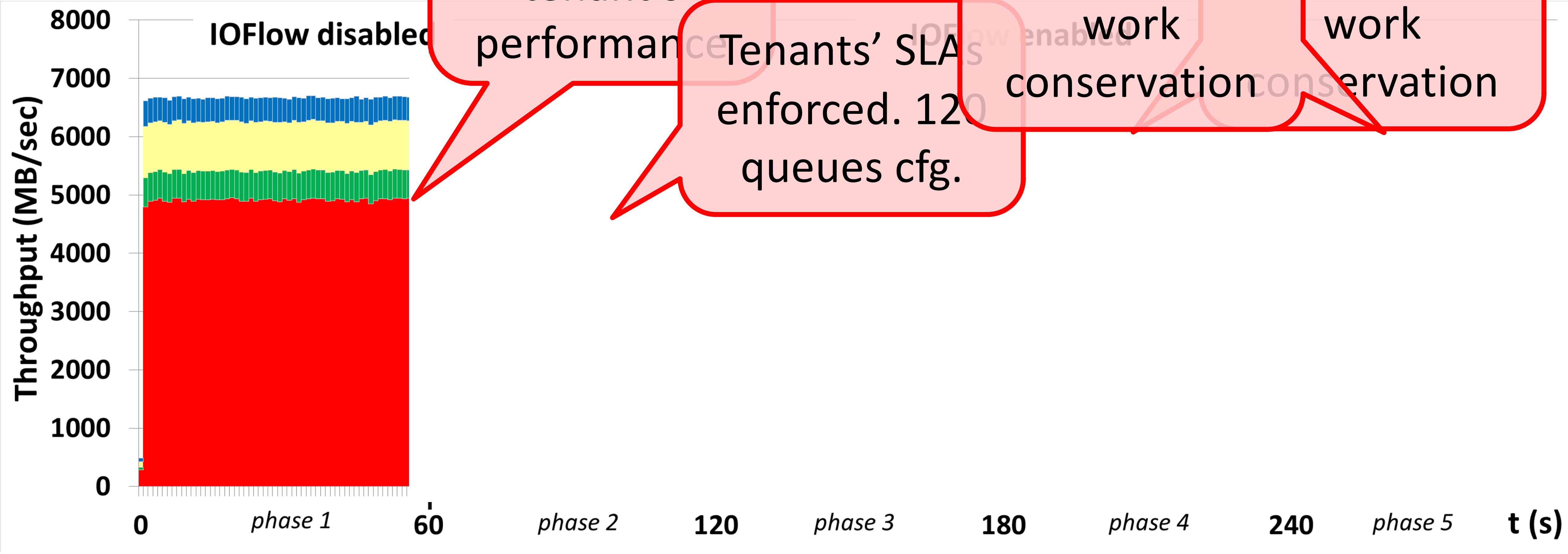
Dynamic inter-tenant work conservation

- Bandwidth released by idle tenant given to active tenants

Dynamic intra-tenant work conservation

- Bandwidth of tenant’s idle VMs given to its active VMs

Results



Controller notices red tenant's performance

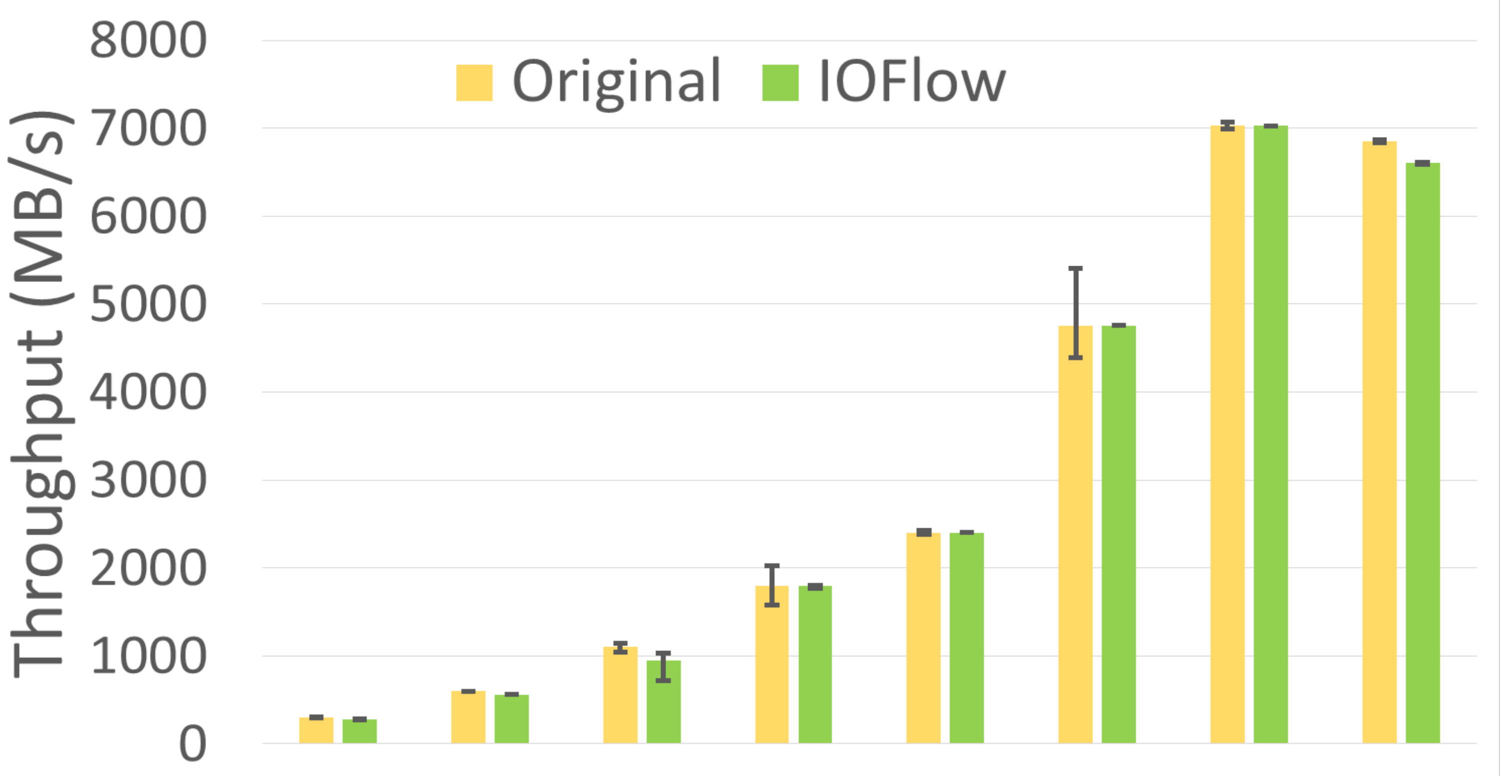
Tenants' SLAs enforced. 120 queues cfg.

Intra-tenant work conservation

Inter-tenant work conservation

Data plane overheads at 40Gbps RDMA

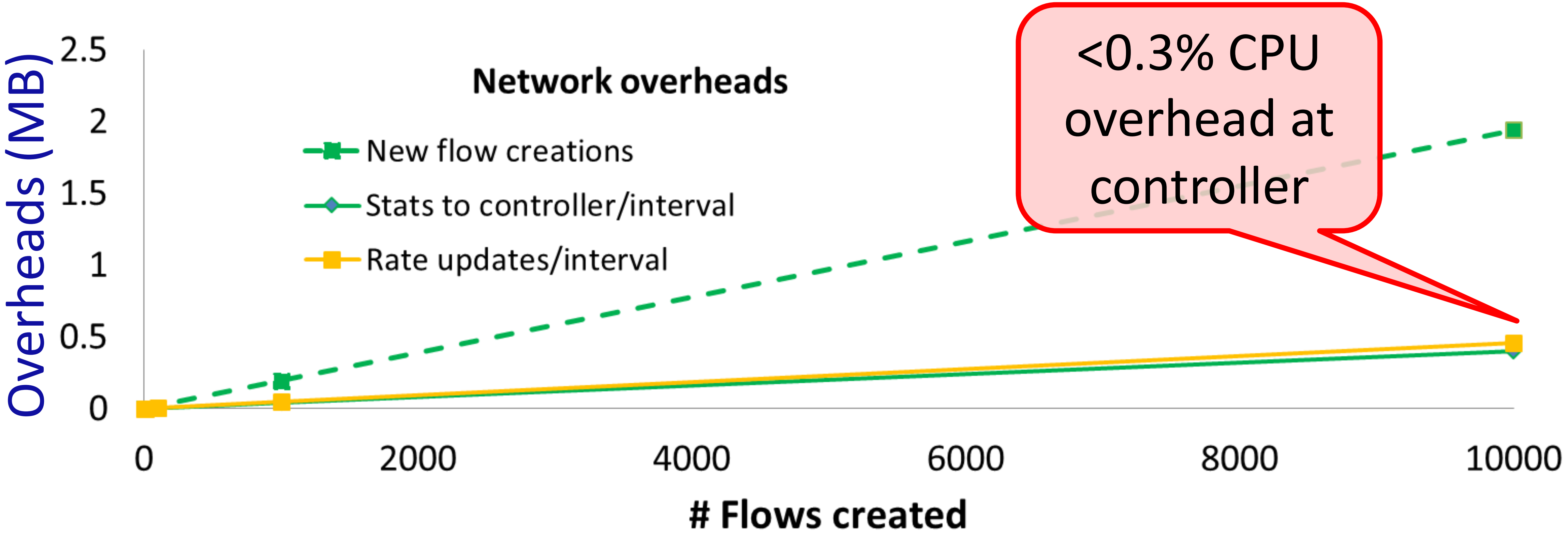
Negligible in previous experiment. To bring out worst case varied IO sizes from 512Bytes to 64KB



Reasonable overheads for enforcing SLAs

Control plane overheads: network and CPU

Controller configures queue rules, receives statistics and updates token rates every interval



<0.3% CPU overhead at controller

Summary of contributions

- Defined and built storage control plane
- Controllable queues in data plane
- Interface between control and data plane (IOFlow API)
- Built centralized control applications that demonstrate power of architecture
- *Ongoing work: applying to public cloud scenarios*

Storage and File Systems in Modern Computer Systems

- IO devices: disks with dedup
  - FAST’08 Dedup
- IO: end-to-end management
  - SOSP’13 IOFlow
- Modern file systems
  - SOSP’03 GFS
- RAID and erasure coding
  - OSDI’16 EC-Cache
- File systems for distributed applications
  - SIGCOMM’01 Chord

The Google File System

Firas Abuzaid

The Google File System. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. In ACM SOSP'03.

Why build GFS?

- Node failures happen frequently
- Files are huge – multi-GB
- Most files are modified by appending at the end
    - Random writes (and overwrites) are practically non-existent
- High sustained bandwidth is more important than low latency
    - Place more priority on processing data in bulk

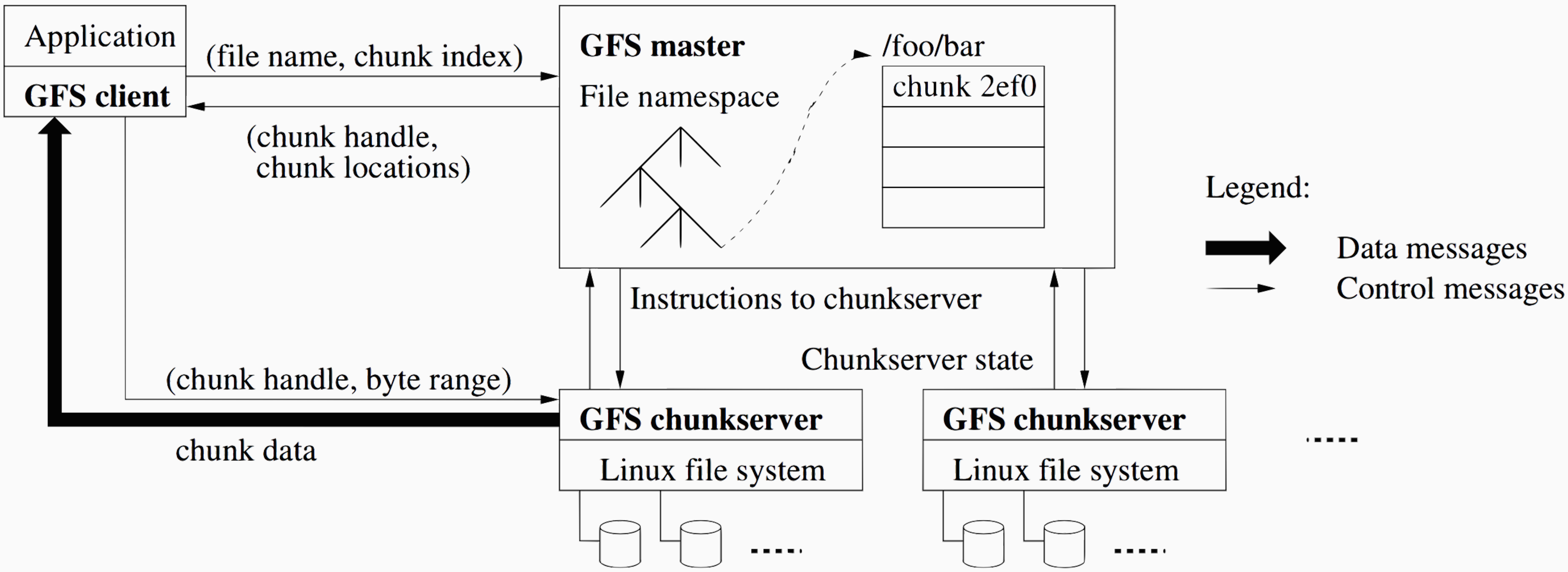
Typical workloads on GFS

- Two kinds of reads: large streaming reads & small random reads
  - Large streaming reads usually read 1MB or more
  - Oftentimes, applications read through contiguous regions in the file
  - Small random reads are usually only a few KBs at some arbitrary offset
- Also many large, sequential writes that append data to files
  - Similar operation sizes to reads
  - Once written, files are seldom modified again
  - Small writes at arbitrary offsets do not have to be efficient
- Multiple clients (e.g. ~100) concurrently appending to a single file
  - e.g. producer-consumer queues, many-way merging

Interface

- Not POSIX-compliant, but supports typical file system operations: `create`, `delete`, `open`, `close`, `read`, and `write`
- `snapshot`: creates a copy of a file or a directory tree at low cost
- `record append`: allow multiple clients to append data to the same file concurrently
  - At least the very first append is guaranteed to be atomic

Architecture



Architecture

- **Very important**: data flow is decoupled from control flow
  - Clients interact with the master for metadata operations
  - Clients interact directly with chunkservers for all files operations
  - This means performance can be improved by scheduling expensive data flow based on the network topology

The Master Node

- Responsible for all system-wide activities
  - managing chunk leases, reclaiming storage space, load-balancing
- Maintains all file system metadata
  - Namespaces, ACLs, mappings from files to chunks, and current locations of chunks
  - all kept in memory, namespaces and file-to-chunk mappings are also stored persistently in ***operation log***
- Periodically communicates with each chunkserver in `HeartBeat` messages
  - This let's master determines chunk locations and assesses state of the overall system

The Operation Log

- Only persistent record of metadata
- Also serves as a logical timeline that defines the serialized order of concurrent operations
- Master recovers its state by replaying the operation log
    - To minimize startup time, the master checkpoints the log periodically

Why a Single Master?

- The master now has global knowledge of the whole system, which drastically simplifies the design
- But the master is (hopefully) never the bottleneck
  - Clients never read and write file data through the master; client only requests from master which chunkservers to talk to
  - Master can also provide additional information about subsequent chunks to further reduce latency
  - Further reads of the same chunk don’t involve the master, either

Why a Single Master?

- Master state is also replicated for reliability on multiple machines, using the operation log and checkpoints
  - If master fails, GFS can start a new master process at any of these replicas and modify DNS alias accordingly
  - “Shadow” masters also provide read-only access to the file system, even when primary master is down
    - They read a replica of the operation log and apply the same sequence of changes
    - Not mirrors of master – they lag primary master by fractions of a second
    - This means we can still read up-to-date file contents while master is in recovery!

Chunks and Chunkservers

- Files are divided into fixed-size ***chunks***, which has an immutable, globally unique 64-bit ***chunk handle***
  - By default, each chunk is replicated three times across multiple chunkservers (user can modify amount of replication)
- Chunkservers store the chunks on local disks as Linux files
  - Metadata per chunk is < 64 bytes (stored in master)
    - Current replica locations
    - Reference count (useful for copy-on-write)
    - Version number (for detecting stale replicas)

Chunk Size

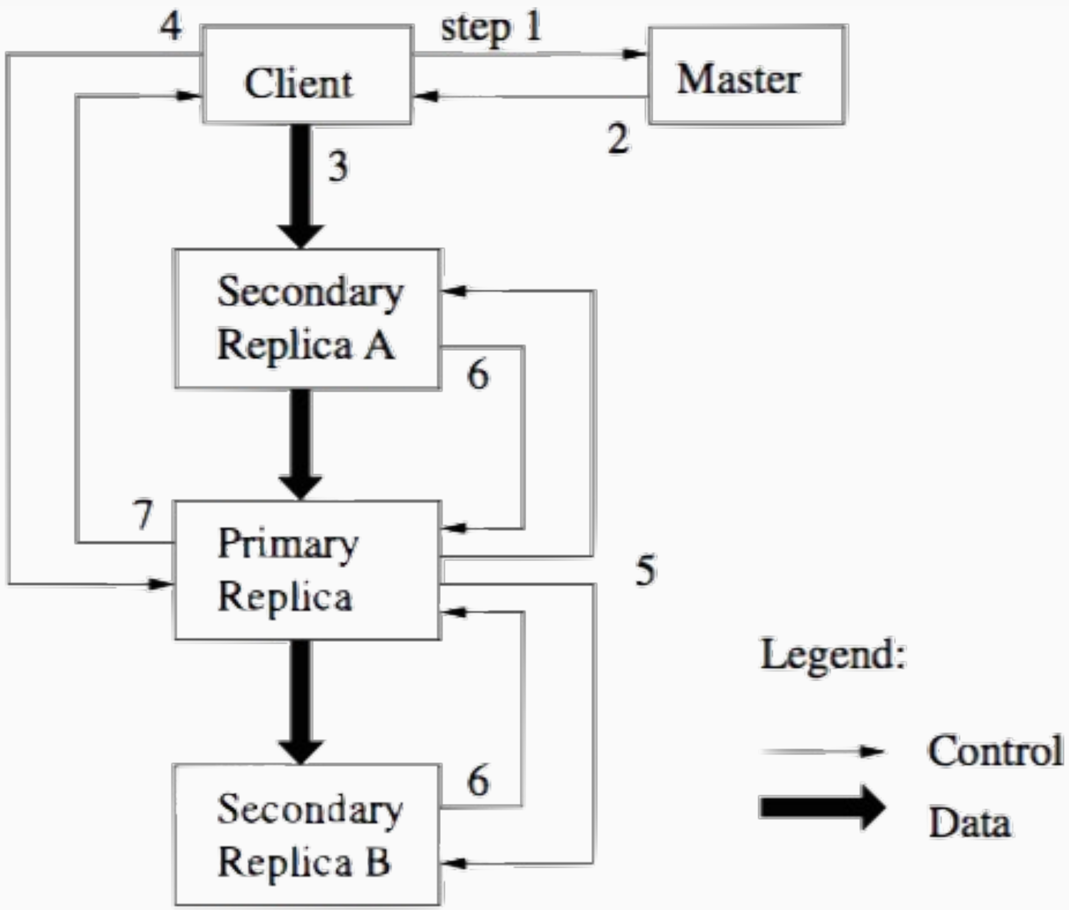
- 64 MB, a key design parameter (Much larger than most file systems.)
- Disadvantages:
  - Wasted space due to internal fragmentation
  - Small files consist of a few chunks, which then get lots of traffic from concurrent clients
    - This can be mitigated by increasing the replication factor
- Advantages:
  - Reduces clients’ need to interact with master (reads/writes on the same chunk only require one request)
  - Since client is likely to perform many operations on a given chunk, keeping a persistent TCP connection to the chunkserver reduces network overhead
  - Reduces the size of the metadata stored in master → metadata can be entirely kept in memory

System Interactions

- If the master receives a modification operation for a particular chunk:
  - Master finds the chunkservers that have the chunk and grants a ***chunk lease*** to one of them
    - This server is called the ***primary***, the other servers are called ***secondaries***
    - The primary determines the serialization order for all of the chunk's modifications, and the secondaries follow that order
  - After the lease expires (~60 seconds), master may grant primary status to a different server for that chunk
    - The master can, at times, revoke a lease (e.g. to disable modifications when file is being renamed)
    - As long as chunk is being modified, the primary can request an extension indefinitely
  - If master loses contact with primary, that's okay: just grant a new lease after the old one expires

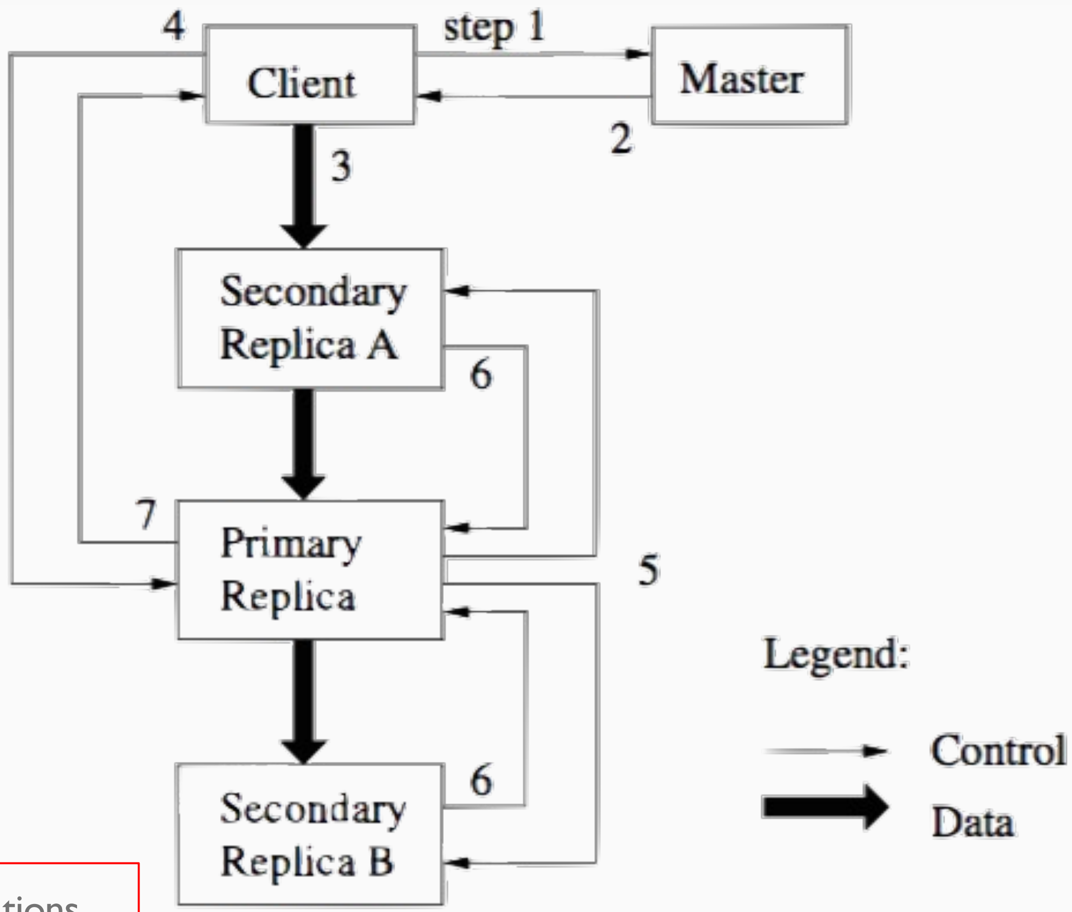
System Interactions

1. Client asks master for all chunkservers (including all secondaries)
2. Master grants a new lease on chunk, increases the chunk version number, tells all replicas to do the same. Replies to client. Client no longer has to talk to master
3. Client pushes data to all servers, not necessarily to primary first
4. Once data is acked, client sends write request to primary. Primary decides serialization order for all incoming modifications and applies them to the chunk



System Interactions

5. After finishing the modification, primary forwards write request and serialization order to secondaries, so they can apply modifications in same order. (If primary fails, this step is never reached.)
6. All secondaries reply back to the primary once they finish the modifications
7. Primary replies back to the client, either with success or error
   - If write succeeds at primary but fails at any of the secondaries, then we have inconsistent state → error returned to client
   - Client can retry steps (3) through (7)



Note: If a `write` straddles chunk boundary, GFS splits this into multiple `write` operations

Conclusions

- Decouple data plane from control plane
    - Control plane: centralized single master
    - Data plane: distributed chuck servers
- Similar concept has been applied to many other systems

	Google	Hadoop
File System	GFS	Hadoop File System (HDFS)
Structured Data Management	BigTable	HBase
Data Processing Engine	MapReduce	Hadoop → Spark

Storage and File Systems in Modern Computer Systems

- IO devices: disks with dedup
  - FAST’08 Dedup
- IO: end-to-end management
  - SOSP’13 IOFlow
- Modern file systems
  - SOSP’03 GFS
- RAID and erasure coding
  - OSDI’16 EC-Cache
- File systems for distributed applications
  - SIGCOMM’01 Chord

EC-Cache: Load-balanced, Low-latency Cluster Caching with Online Erasure Coding

Rashmi Vinayak
UC Berkeley

Joint work with

Mosharaf Chowdhury, Jack Kosaian (U Michigan)
Ion Stoica, Kannan Ramchandran (UC Berkeley)

Caching for data-intensive clusters

- Data-intensive clusters rely on **distributed, in-memory caching** for high performance
  - Reading from memory orders of magnitude faster than from disk/ssd
  - Example: Alluxio (formerly Tachyon[†])

[†] Li et al. SOCC 2014

Imbalances prevalent in clusters

Sources of imbalance:

- Skew in object popularity
- Background network imbalance
- Failures/unavailabilities

Imbalances prevalent in cluster

Sources of imbalance:

- Skew in object popularity
- Background network imbalance
- Failures/unavailabilities

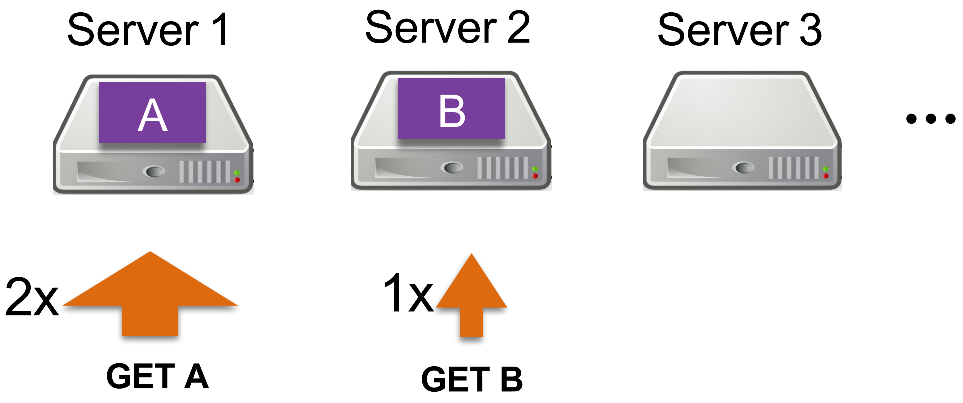
➡ Adverse effects:

- load imbalance
- high read latency

Single copy in memory often not sufficient to get good performance

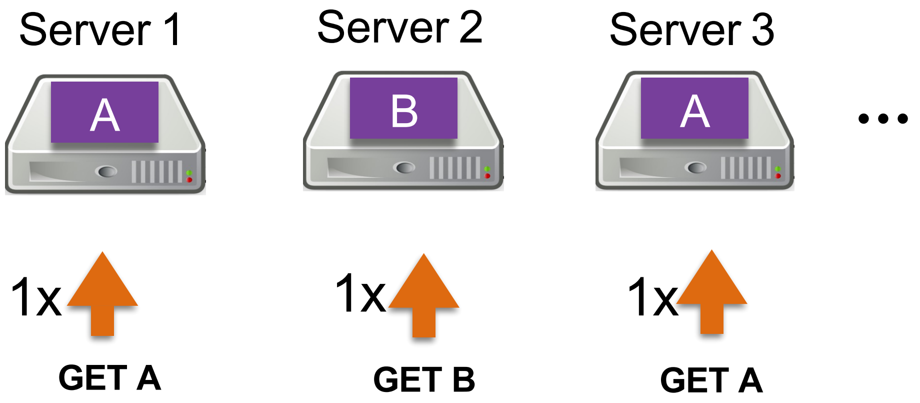
Popular approach: Selective Replication

- Uses some memory overhead to cache replicas of objects based on their popularity
  - more replicas for more popular objects



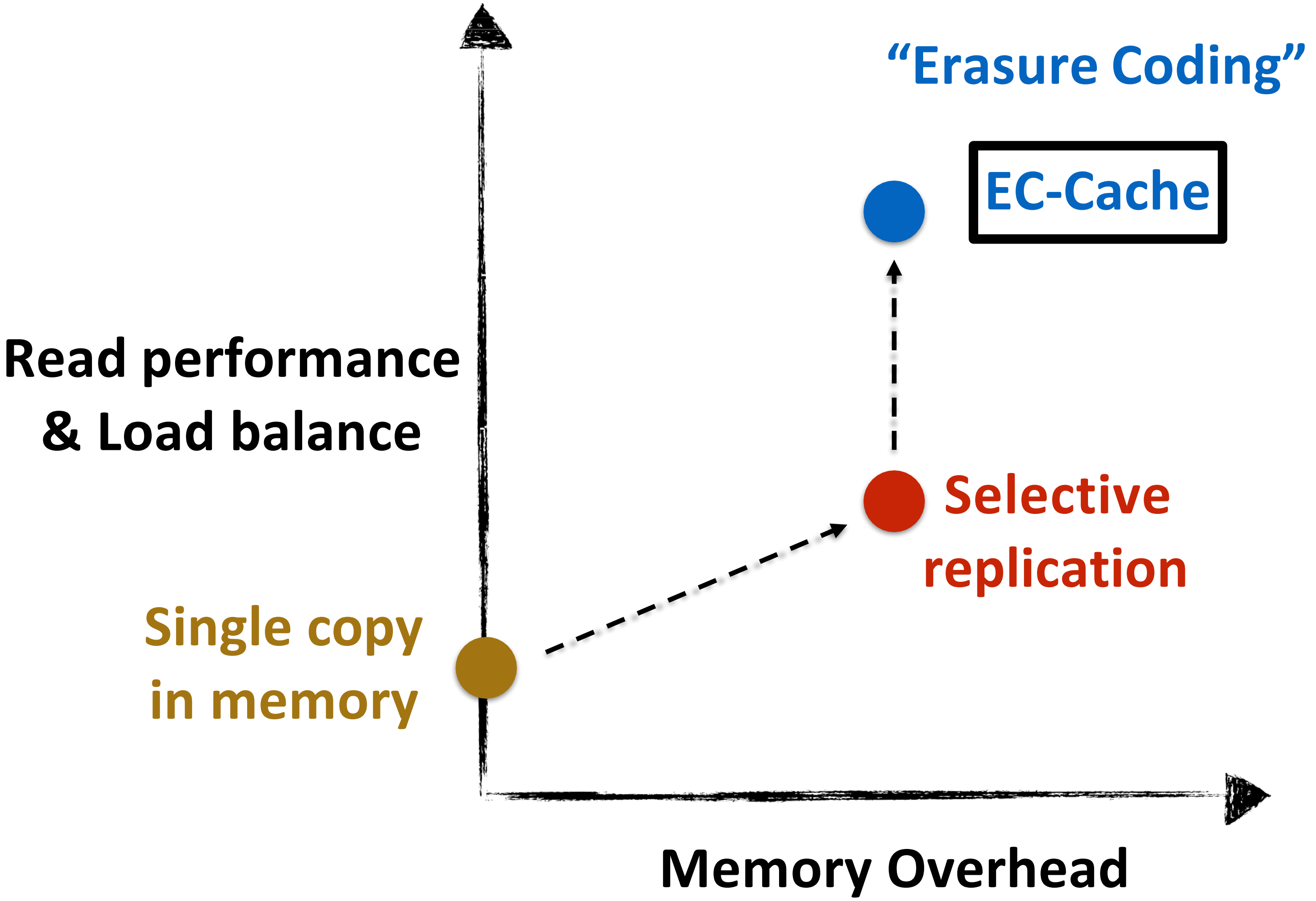
Popular approach: Selective Replication

- Uses some memory overhead to cache replicas of objects based on their popularity
  - more replicas for more popular objects



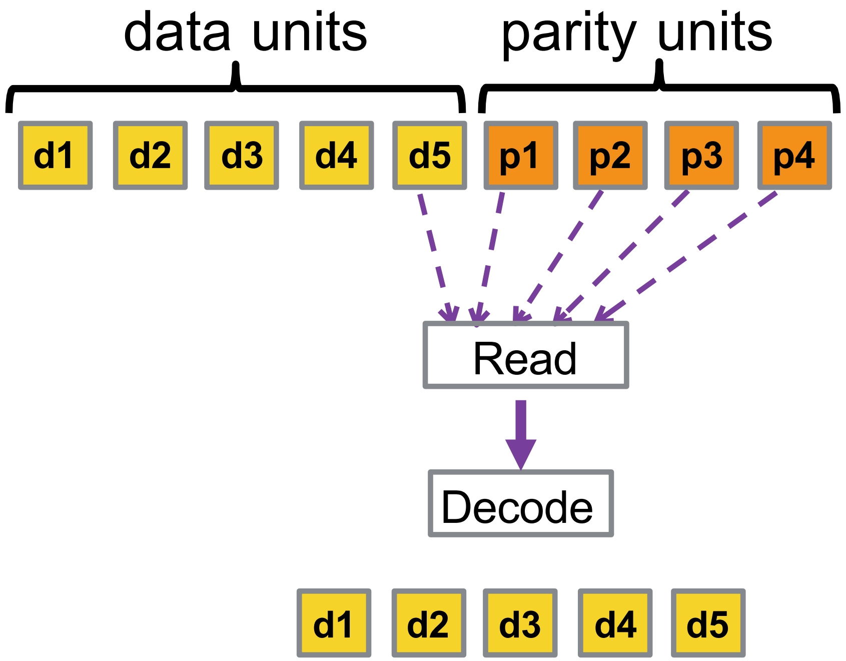
- Used in data-intensive clusters[†] as well as widely used in key-value stores for many web services such as Facebook Tao[‡]

[†] Ananthanarayanan et al. NSDI 2011, [‡] Bronson et al. ATC 2013


"Erasure Coding"
EC-Cache
Read performance & Load balance
Selective replication
Single copy in memory
Memory Overhead

Quick primer on erasure coding

- Takes in ***k*** data units and creates ***r*** "parity" units
- ***Any k*** of the (k+r) units are sufficient to decode the original k data units



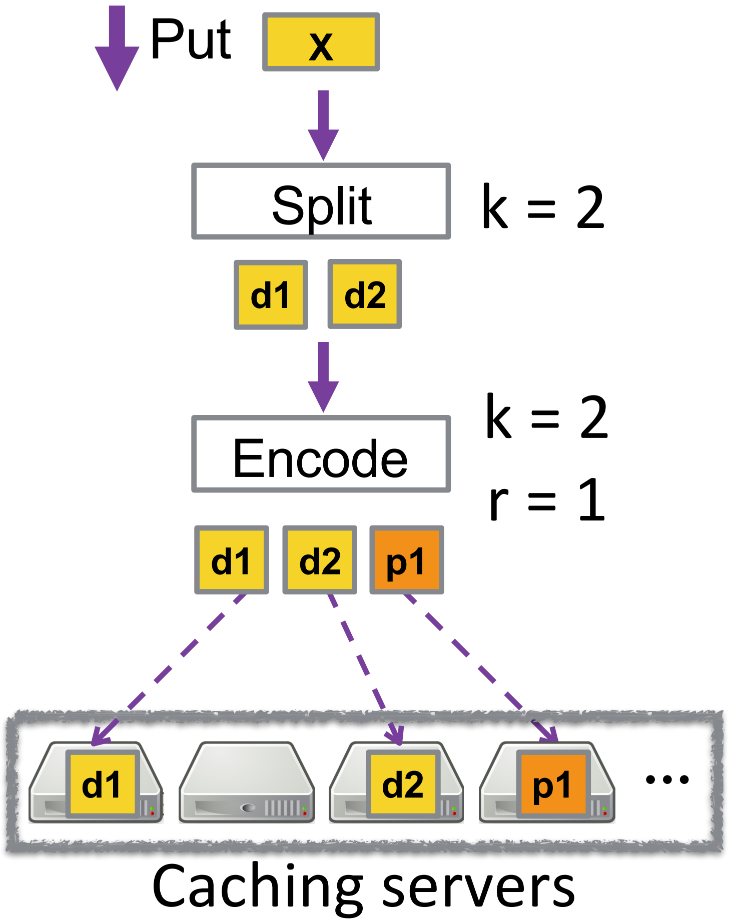
- k = 5
- r = 4

EC-Cache bird's eye view: Writes

- Object split into k data units

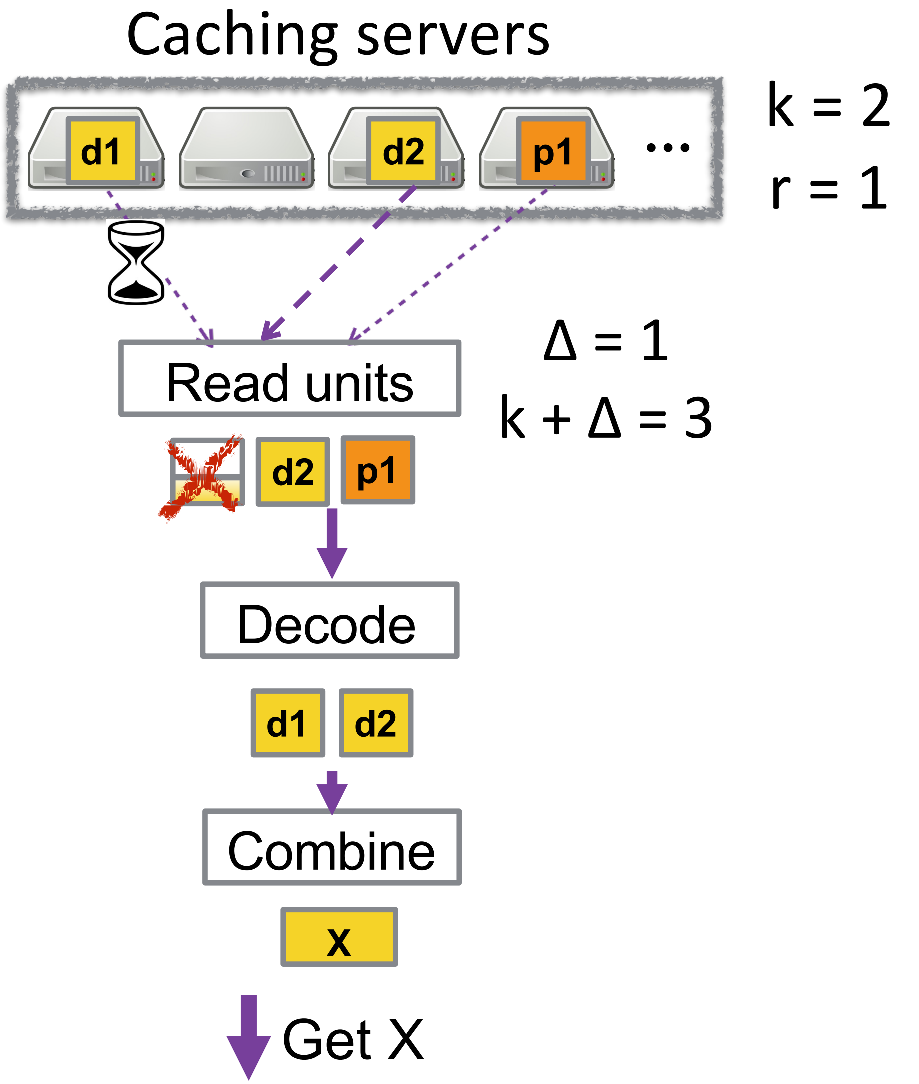
- Encoded to generate r parity units

- (k+r) units cached on distinct servers chosen uniformly at random



EC-Cache bird's eye view: Reads

- Read from (k + Δ) units of the object chosen uniformly at random
  - "Additional reads"
- Use the first k units that arrive
- Decode the data units
- Combine the decoded units



Erasure coding: How does it help?

1. **Finer control over memory overhead**
   - Selective replication allows only integer control
   - Erasure coding allows fractional control
   - E.g., k = 10 allows control in of multiples of 0.1

2. **Object splitting helps in load balancing**
   - Smaller granularity reads help to smoothly spread load
   - Analysis on a certain simplified model:

$$\frac{\mathrm{Var}(L_{\text{EC-Cache}})}{\mathrm{Var}(L_{\text{Selective Replication}})} = \frac{1}{k}$$

Erasure coding: How does it help?

3. **Object splitting reduces median latency but hurts tail latency**
   - Read parallelism helps reduce median latency
   - Straggler effect hurts tail latency (if no additional reads)

4. **“Any k out of (k+r)” property helps to reduce tail latency**
   - Read from (k + Δ) and use the first k that arrive
   - Δ = 1 often sufficient to reign in tail latency

Design considerations

1. Purpose of erasure codes

Storage systems	EC-Cache
- Space-efficient fault tolerance	- Reduce read latency - Load balance

Design considerations

2. Choice of erasure code

Storage systems	EC-Cache
• Optimize resource usage during reconstruction operations† • Some codes do not have “any k out of (k+r)” property	• No reconstruction operations in caching layer; data persisted in underlying storage • “Any k out of (k+r)” property helps in load balancing and reducing latency when reading objects

†Rashmi et al. SIGCOMM 2014, Sathiamoorthy et al. VLDB 2013, Huang et al. ATC 2012

Design considerations

3. How do we use erasure coding: across vs. within objects

Storage systems	EC-Cache
• Some systems encode across objects (e.g., HDFS-RAID); some within (e.g., Ceph) • Does not affect fault tolerance	• Need to encode within objects · To spread load across both data & parity • Encoding across: Very high BW overhead for reading object using parities[†]

[†] Rashmi et al. SIGCOMM 2014, HotStorage 2013

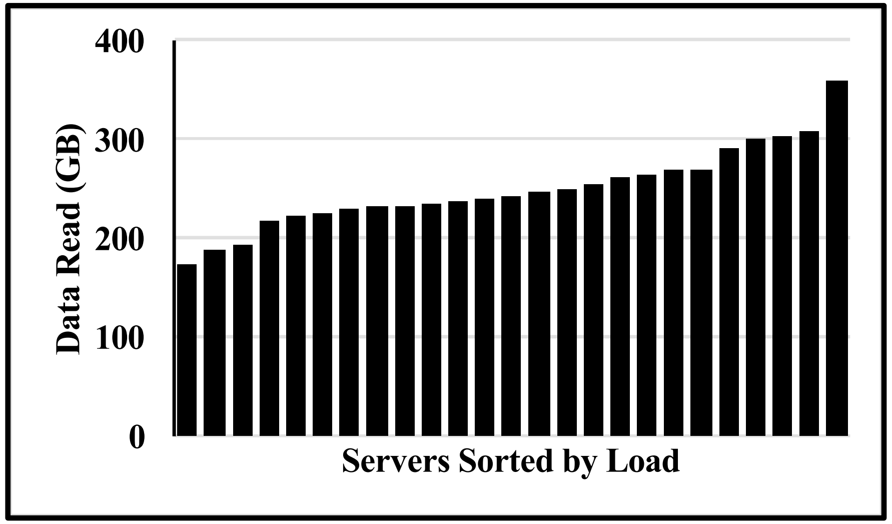
Implementation

- EC-Cache on top of Alluxio (formerly Tachyon)
  - Backend caching servers: cache data — unaware of erasure coding
  - EC-Cache client library: all read/write logic handled
- Reed-Solomon code
  - Any k out of (k+r) property
- Intel ISA-L hardware acceleration library
  - Fast encoding and decoding

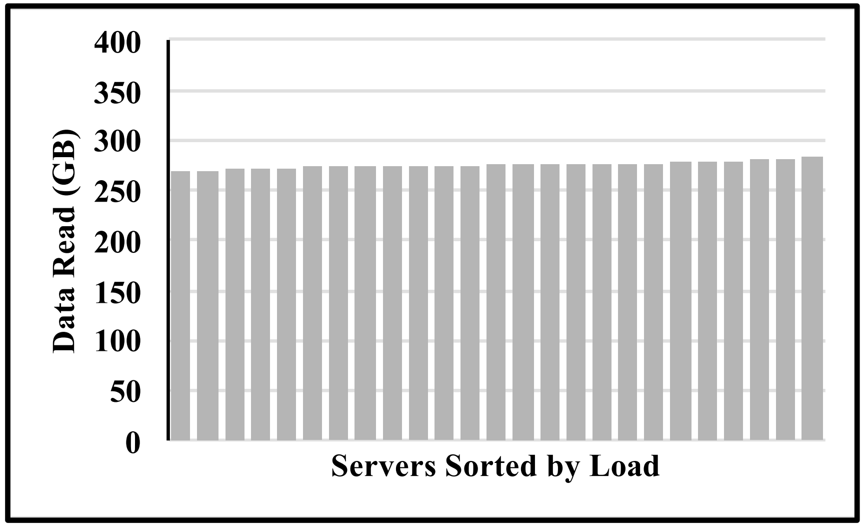
Evaluation set-up

- Amazon EC2
- 25 backend caching servers and 30 client servers
- Object popularity: Zipf distribution with high skew
- EC-Cache uses k = 10, Δ = 1
  - BW overhead = 10%
- Varying object sizes

Load balancing



Selective Replication



EC-Cache

- Percent imbalance metric:

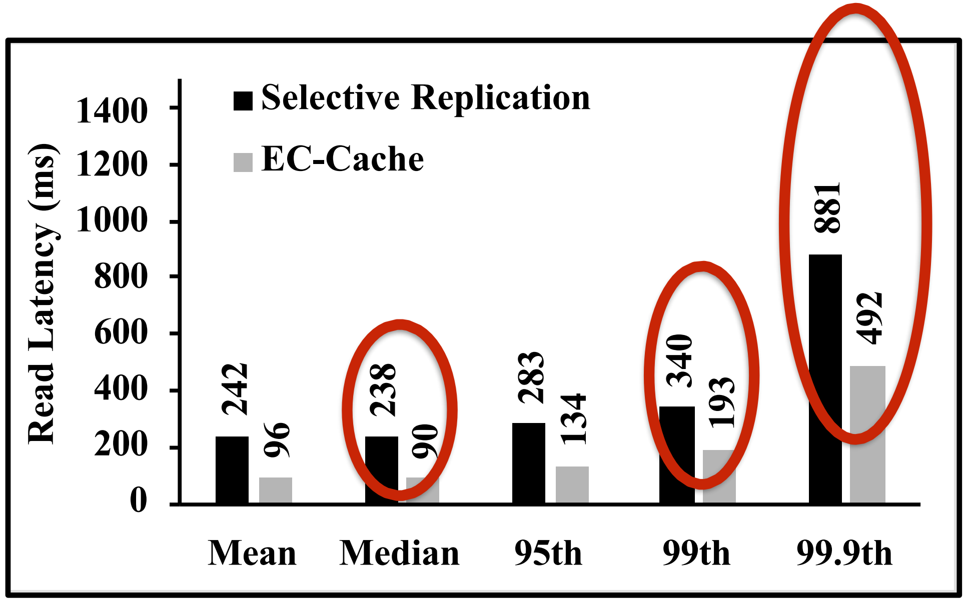
$$\lambda = \frac{L_{max} - L_{avg}}{L_{avg}} * 100$$

λ_{SR} = 43.45%

λ_{EC} = 13.14%

> 3x reduction in load imbalance metric

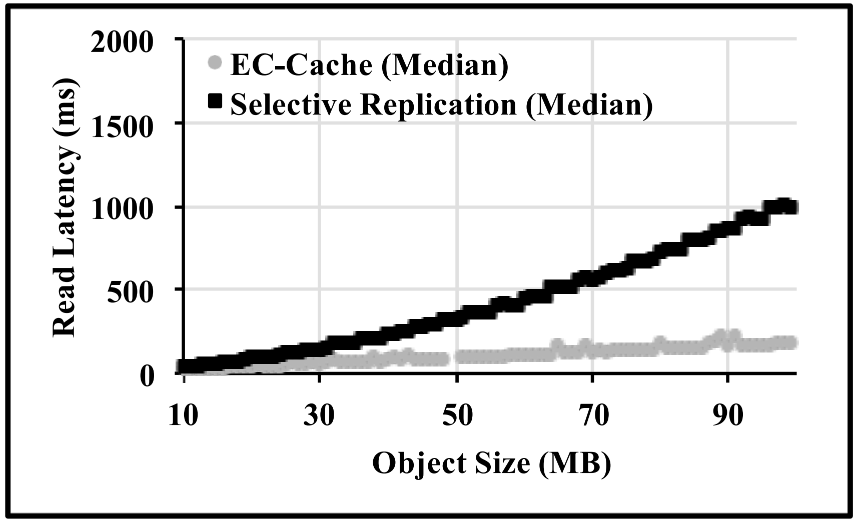
Read latency



- Median: **2.64x** improvement
- 99th and 99.9th: **~1.75x** improvement

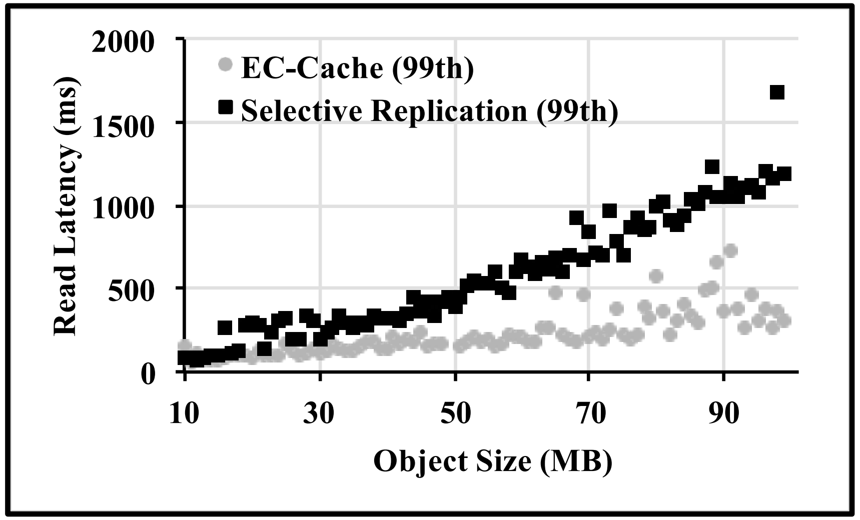
Varying object sizes

Median latency



5.5x improvement for 100MB

Tail latency

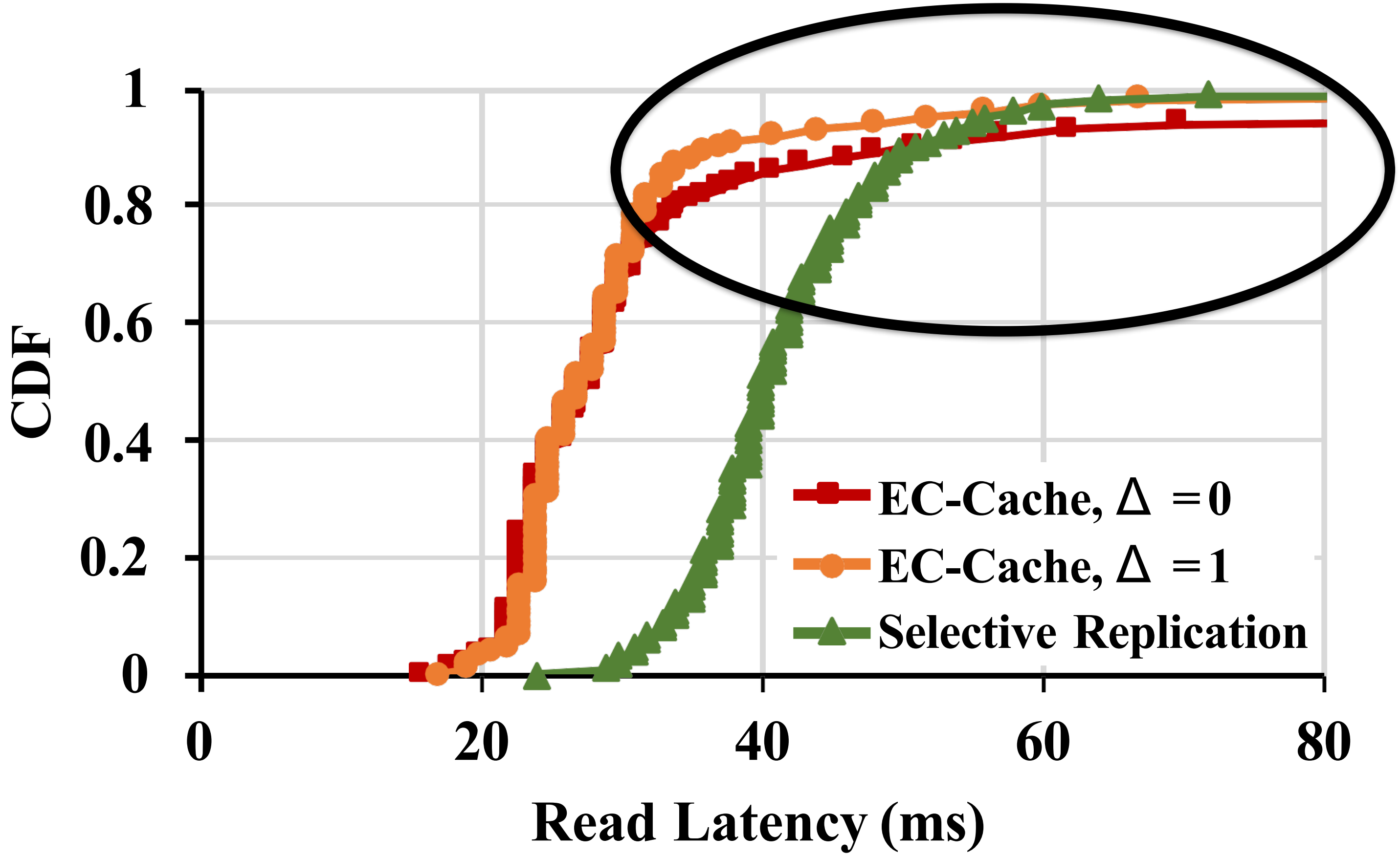


3.85x improvement for 100 MB

More improvement for larger object sizes

Role of additional reads (Δ)

Significant degradation in tail latency without additional reads (i.e., Δ = 0)



Summary

- EC-Cache
  - Cluster cache employing erasure coding for load balancing and reducing read latencies
  - Demonstrates new application and new goals for which erasure coding is highly effective
- Implementation on Alluxio
- Evaluation
  - Load balancing: > 3x improvement
  - Median latency: > 5x improvement
  - Tail latency: > 3x improvement

Thanks!

Storage and File Systems in Modern Computer Systems

- IO devices: disks with dedup
  - FAST’08 Dedup
- IO: end-to-end management
  - SOSP’13 IOFlow
- Modern file systems
  - SOSP’03 GFS
- RAID and erasure coding
  - OSDI’16 EC-Cache
- File systems for distributed applications
  - SIGCOMM’01 Chord

Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica, Robert Morris, David Karger,
M. Frans Kaashoek, Hari Balakrishnan

ACM SIGCOMM 2001

- presentation based on slides by Daniel Figueiredo and Robert Morris

Outline

□ Motivation and background

□ Consistency caching

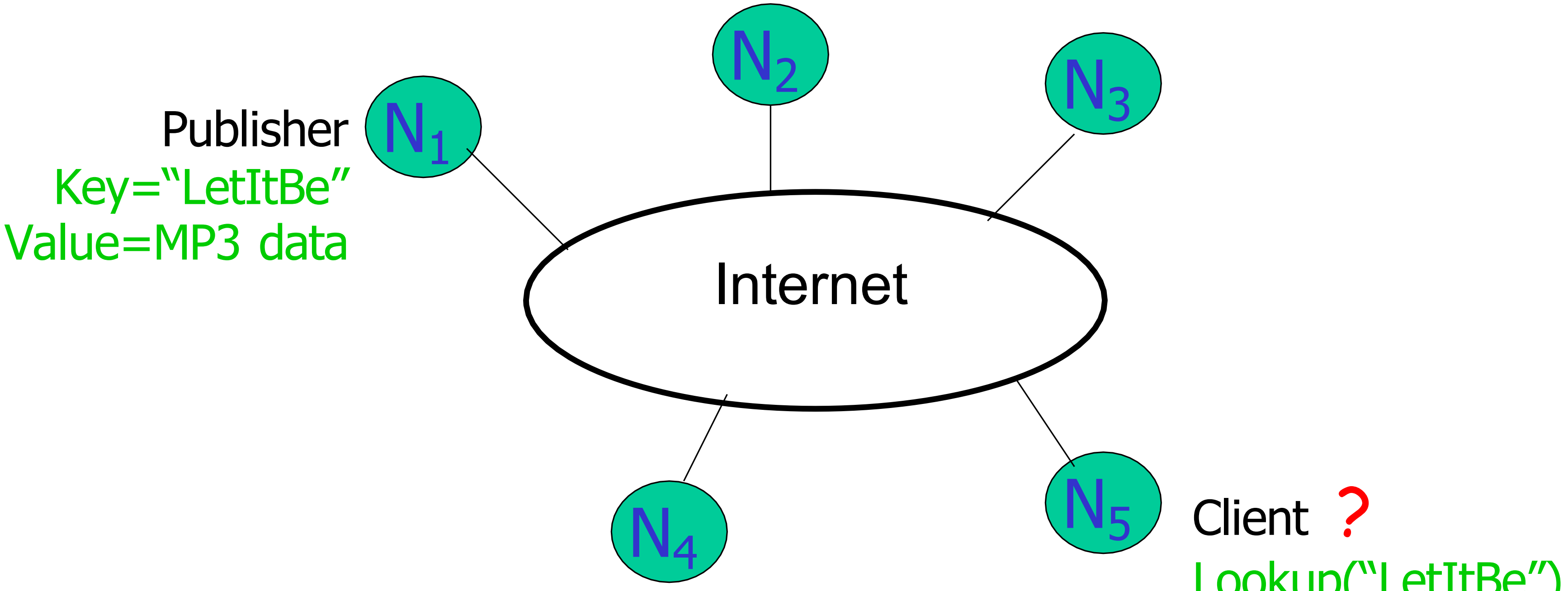
□ Chord

□ Performance evaluation

□ Conclusion and discussion

Motivation

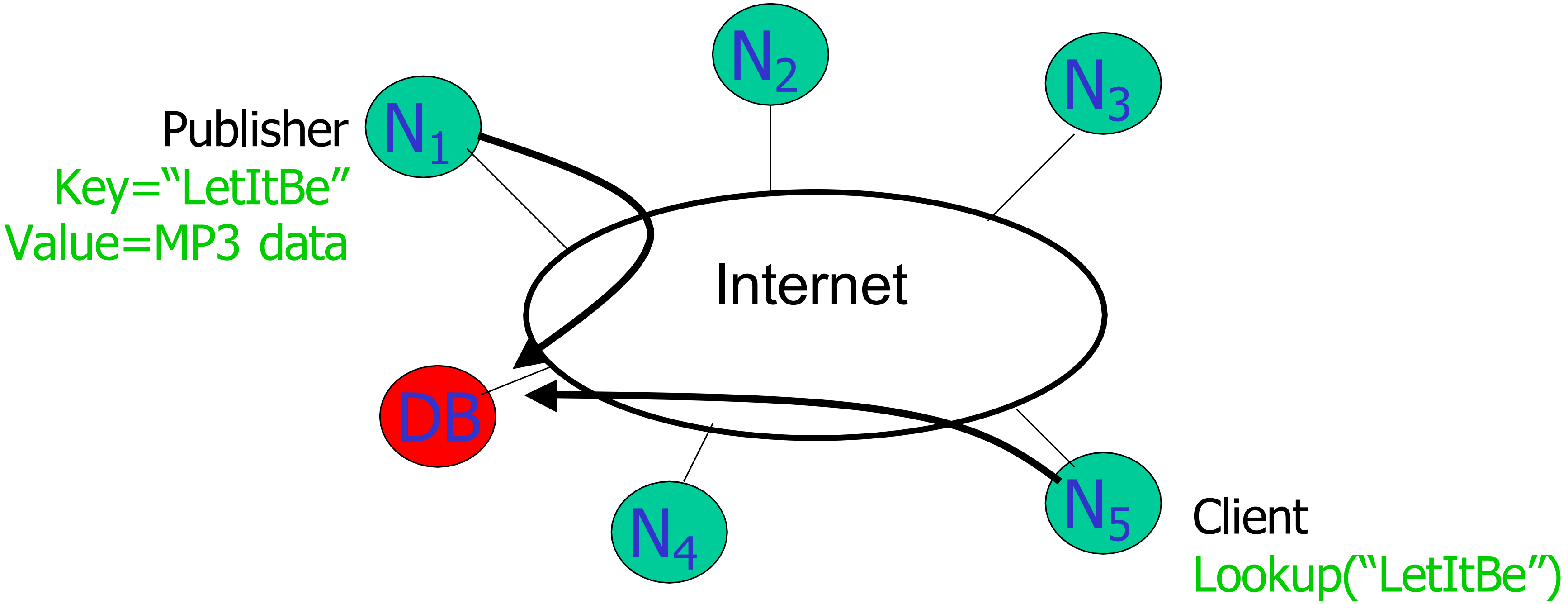
How to find data in a distributed file sharing system?



- Lookup is the key problem

Centralized Solution

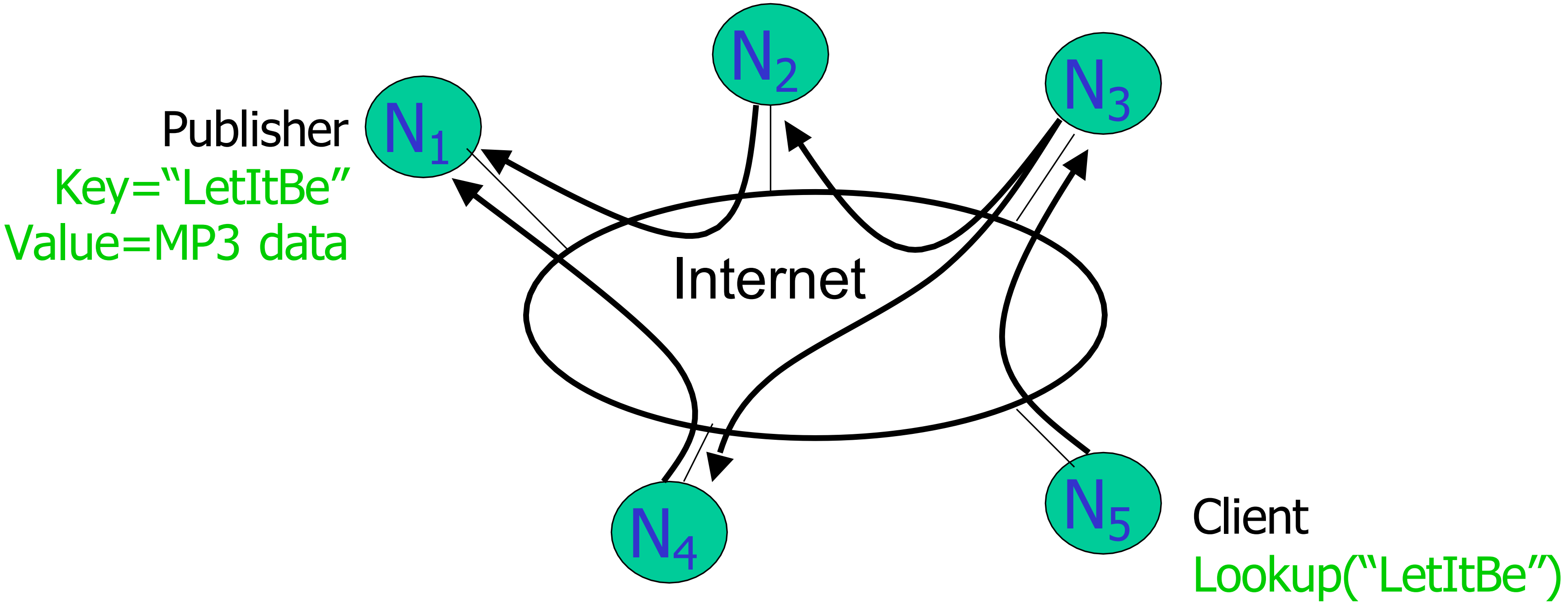
- Central server (Napster)



- Requires O(M) state
- Single point of failure

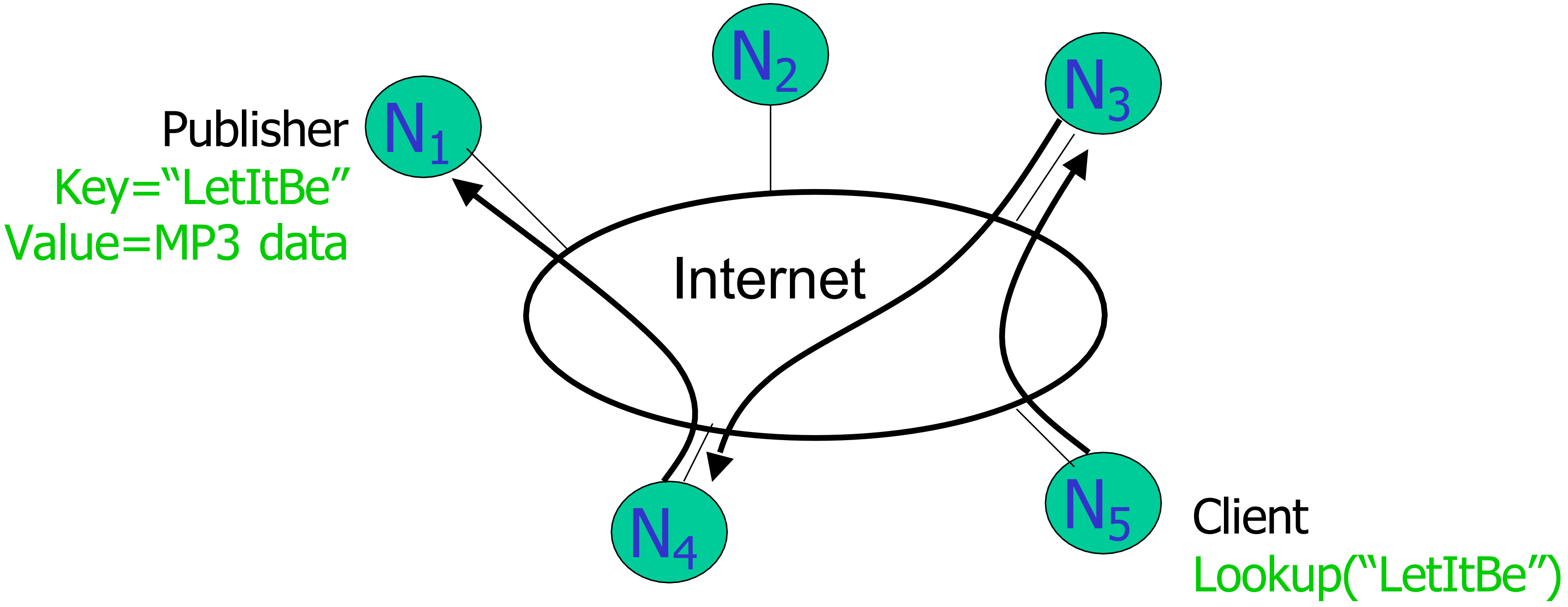
Naïve Distributed Solution

- Flooding (Gnutella, Morpheus, etc.)



- Worst case O(N) messages per lookup

Chord: Routed messages



Routing Challenges

- Define a useful key nearness metric
- Keep the hop count small
- Keep the routing tables "right size"
- Stay robust despite rapid changes in membership

Chord Overview

- Provides peer-to-peer hash lookup service:
    - Lookup(key) → IP address
    - Chord does not store the data
- How does Chord locate a node?
- How does Chord maintain routing tables?
- How does Chord cope with changes in membership?

Chord properties

- Efficient: O(Log N) messages per lookup
  - N is the total number of servers
- Scalable: O(Log N) state per node
- Robust: survives massive changes in membership

- Proofs are in paper / tech report
  - Assuming no malicious participants

Chord IDs

- m bit identifier space for both keys and nodes
- Key identifier = SHA-1(key)

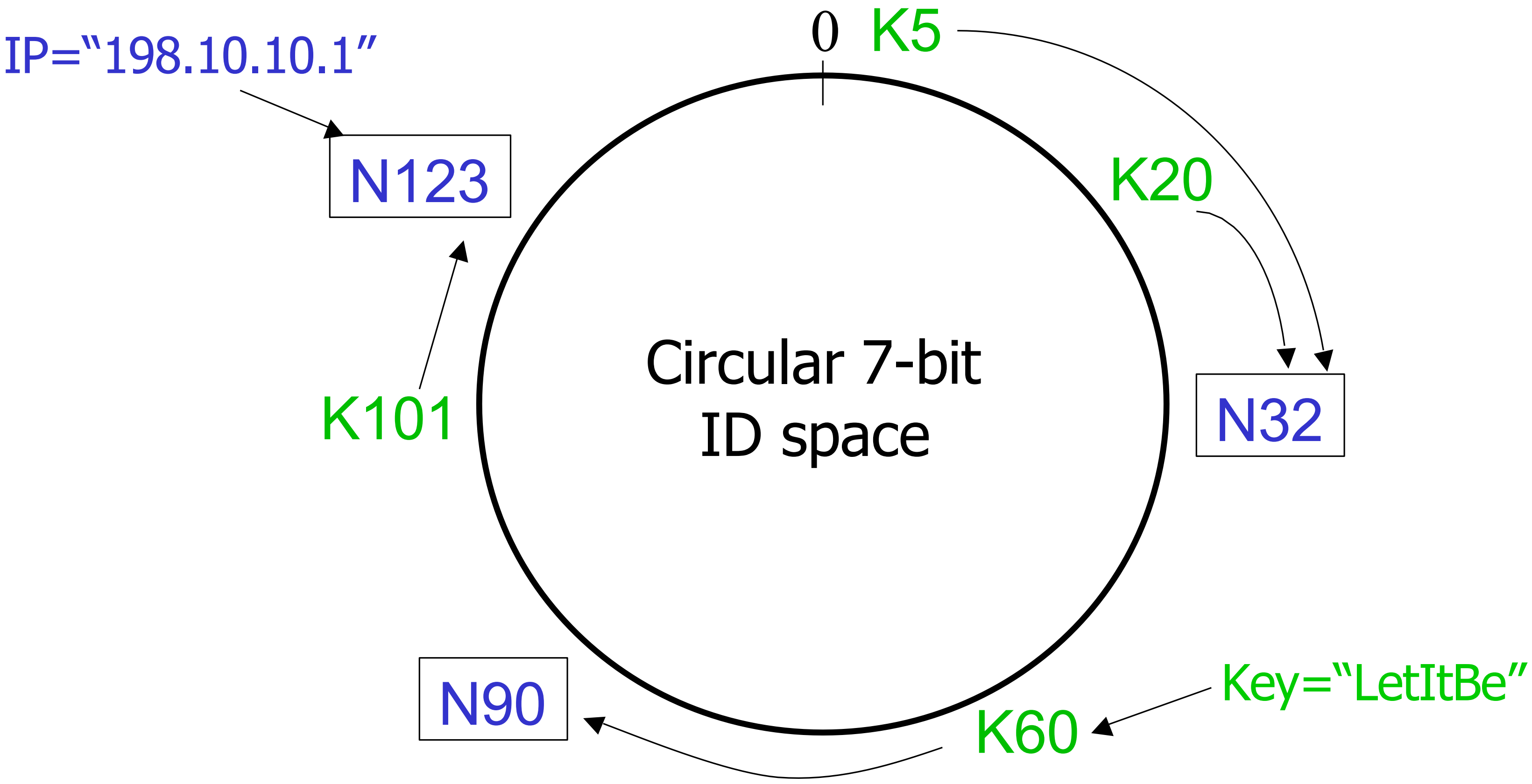
  Key=“LetItBe” $\xrightarrow{\text{SHA-1}}$ ID=60

- Node identifier = SHA-1(IP address)

  IP=“198.10.10.1” $\xrightarrow{\text{SHA-1}}$ ID=123

- Both are uniformly distributed
- How to map key IDs to node IDs?

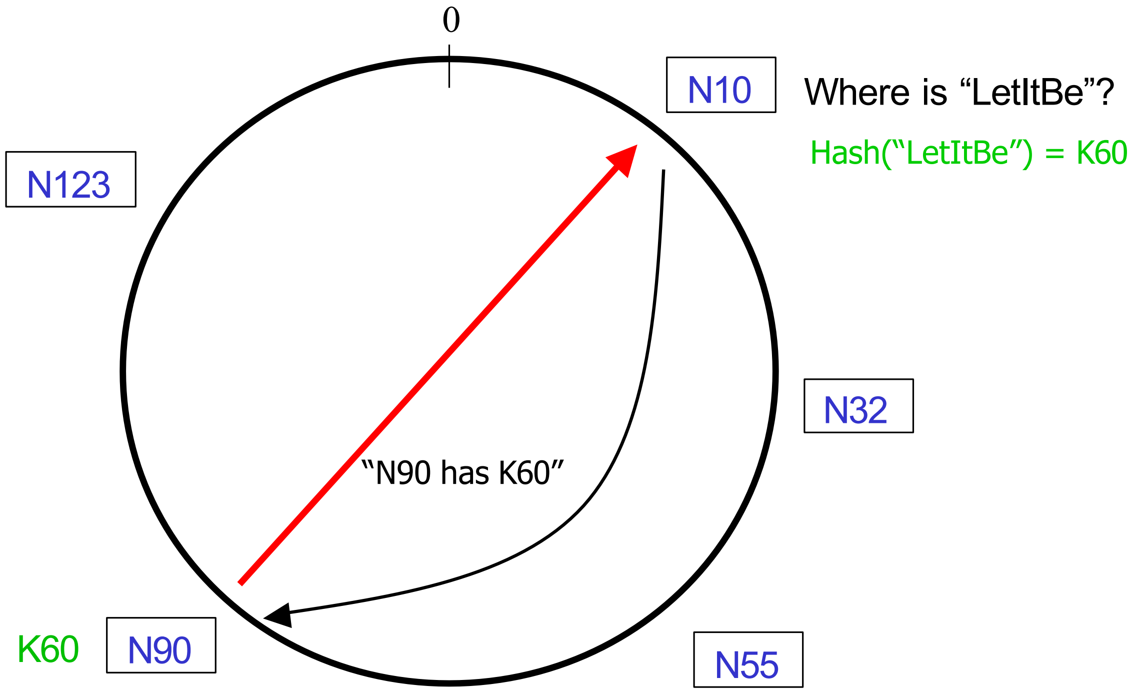
Consistent Hashing [Karger 97]



- A key is stored at its successor: node with next higher ID

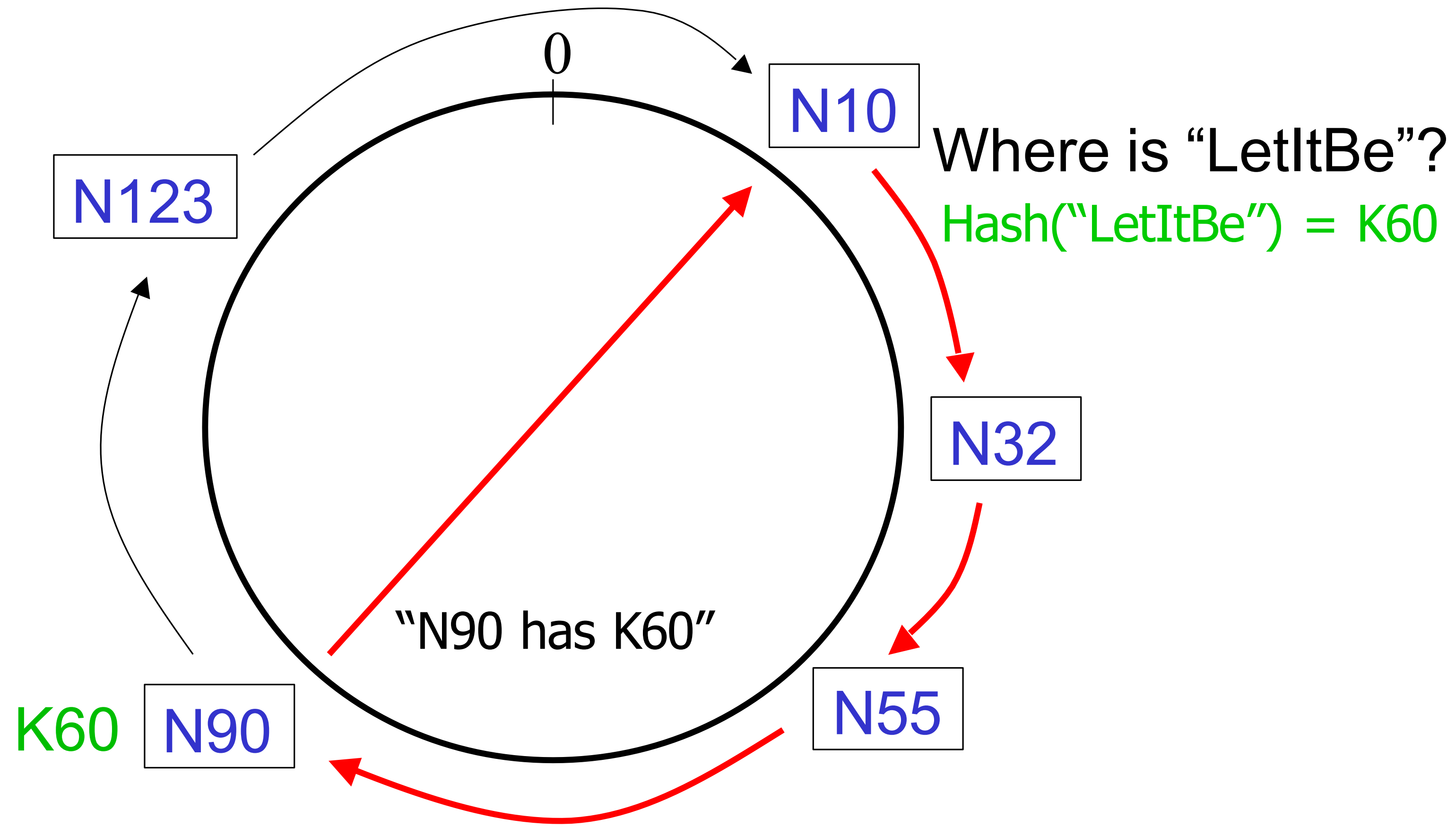
Consistent Hashing

- Every node knows of every other node
  - requires global information
- Routing tables are large O(N)
- Lookups are fast O(1)



Chord: Basic Lookup

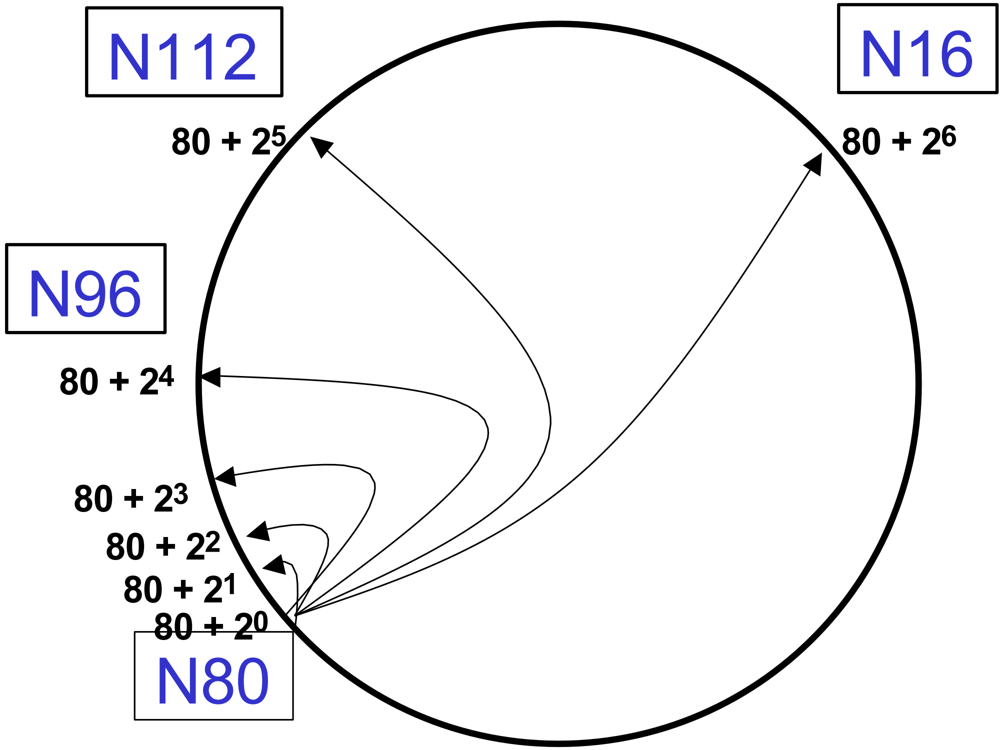
- Every node knows its successor in the ring



- requires O(N) time

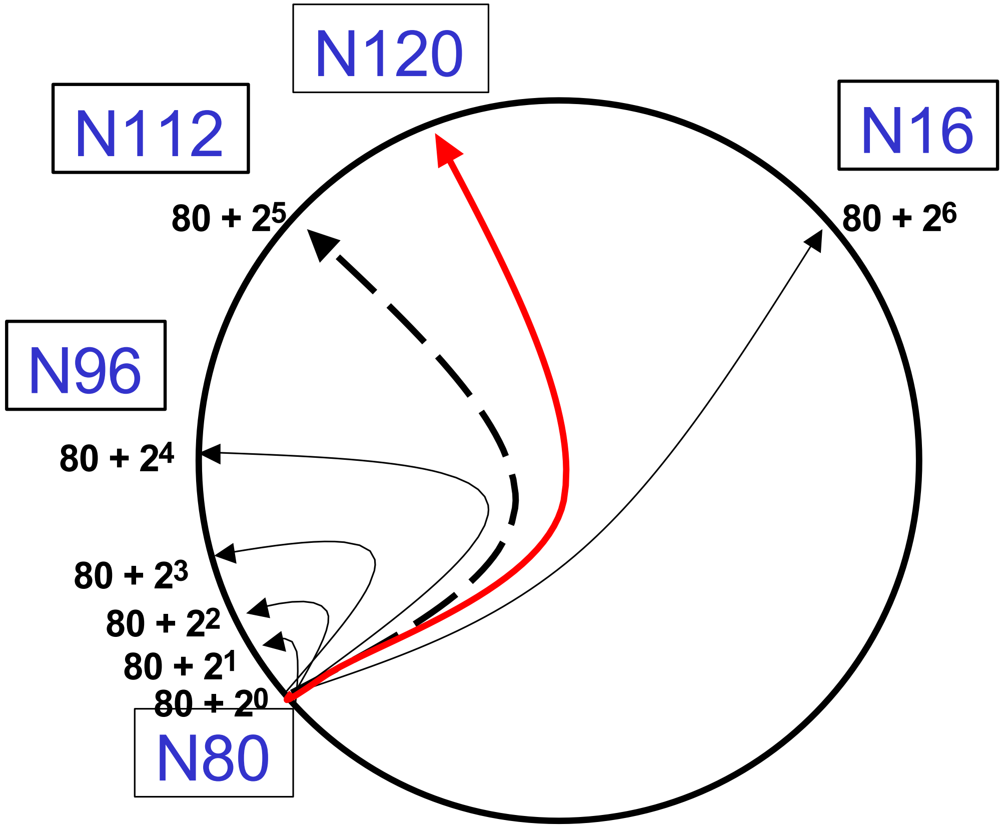
“Finger Tables”

- Every node knows *m* other nodes in the ring
- Increase distance exponentially



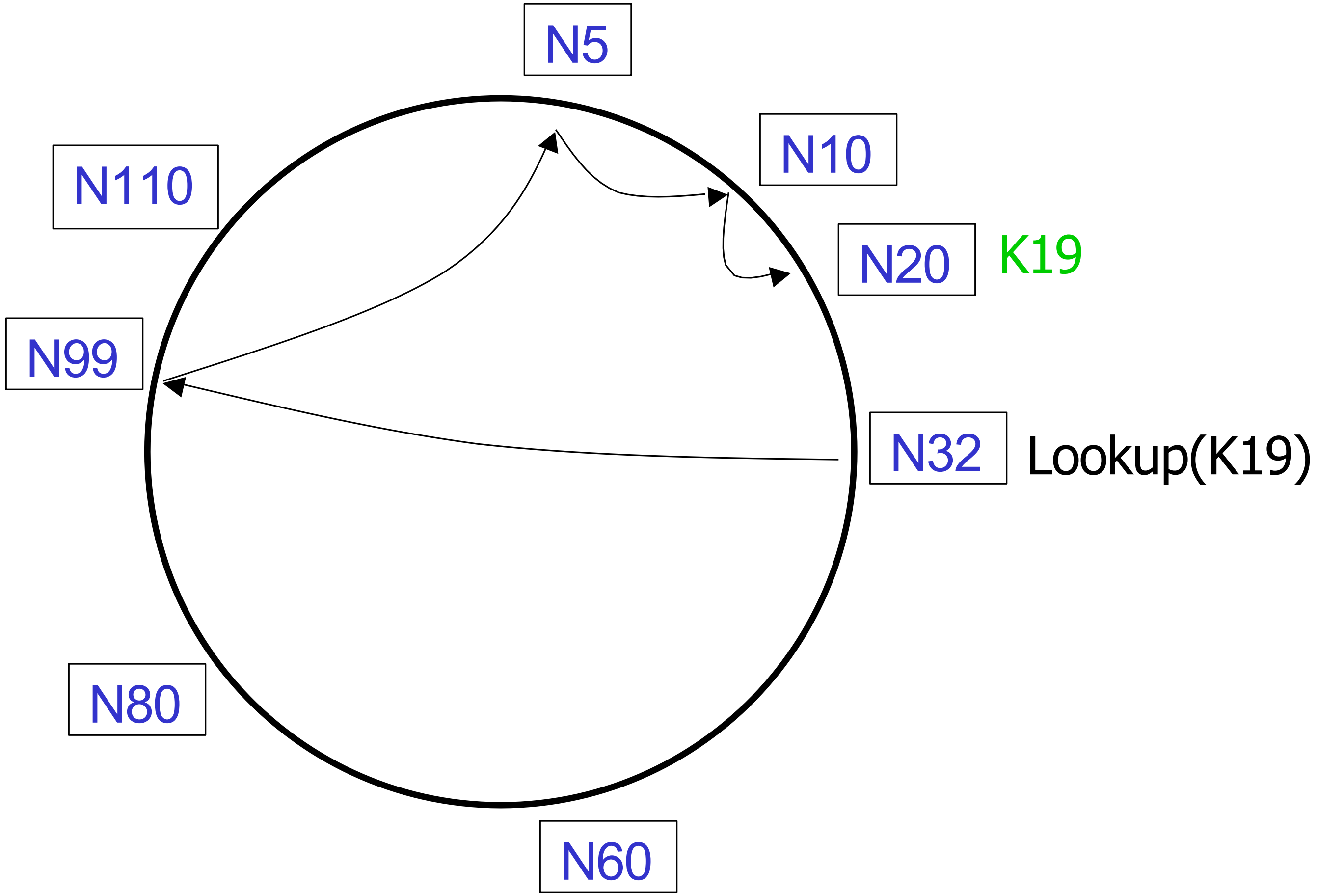
“Finger Tables”

- Finger *i* points to successor of $n+2^i$



Lookups are Faster

- Lookups take O(Log N) hops



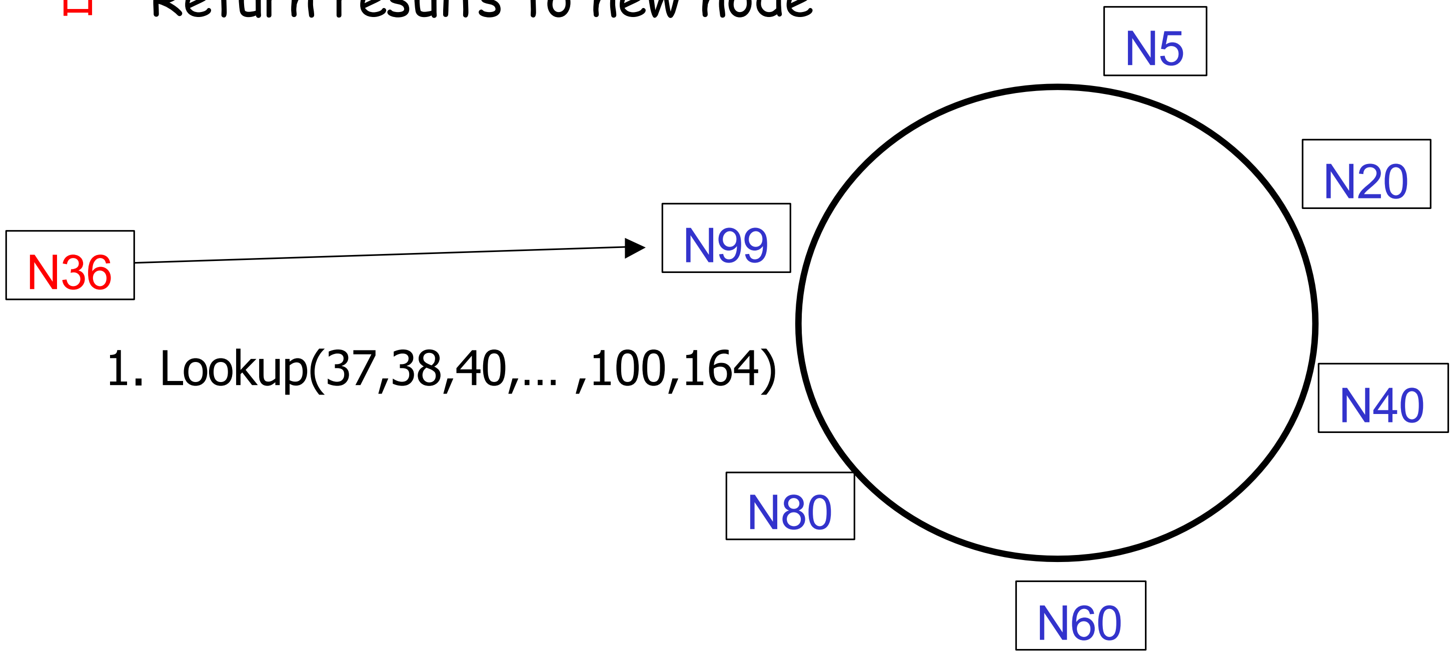
Joining the Ring

- Three step process:
  - Initialize all fingers of new node
  - Update fingers of existing nodes
  - Transfer keys from successor to new node

- Less aggressive mechanism (lazy finger update):
  - Initialize only the finger to successor node
  - Periodically verify immediate successor, predecessor
  - Periodically refresh finger table entries

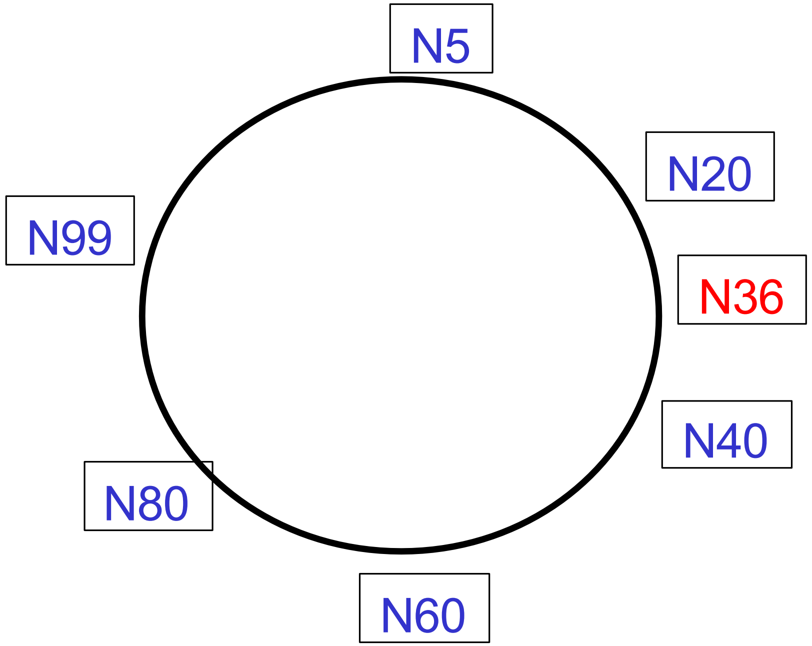
Joining the Ring - Step 1

- Initialize the new node finger table
  - Locate any node *p* in the ring
  - Ask node *p* to lookup fingers of new node N36
  - Return results to new node



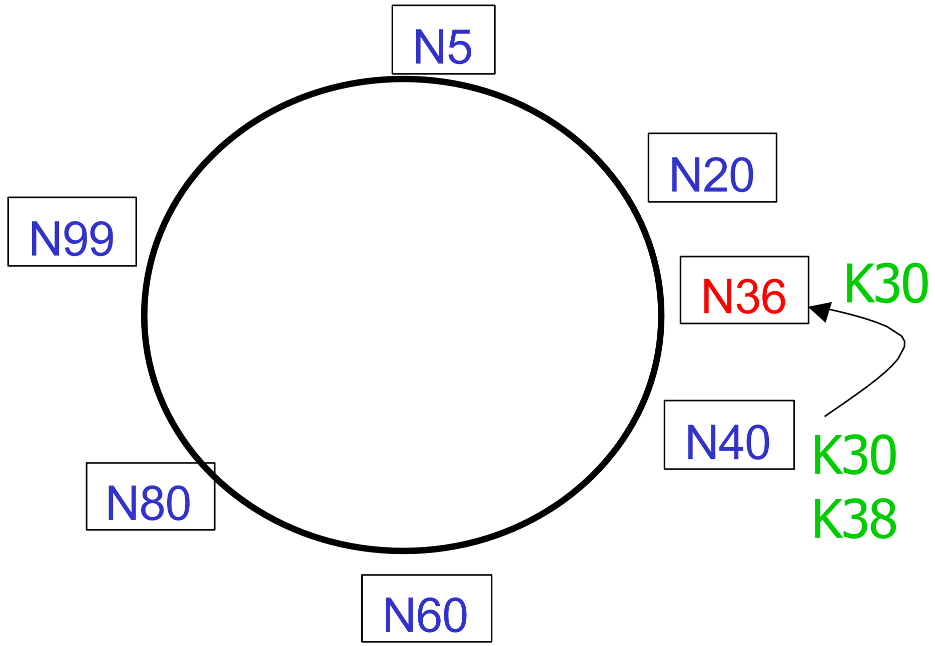
Joining the Ring - Step 2

- Updating fingers of existing nodes
  - new node calls update function on existing nodes
  - existing nodes can recursively update fingers of other nodes



Joining the Ring - Step 3

- Transfer keys from successor node to new node
  - only keys in the range are transferred



Copy keys 21..36
from N40 to N36

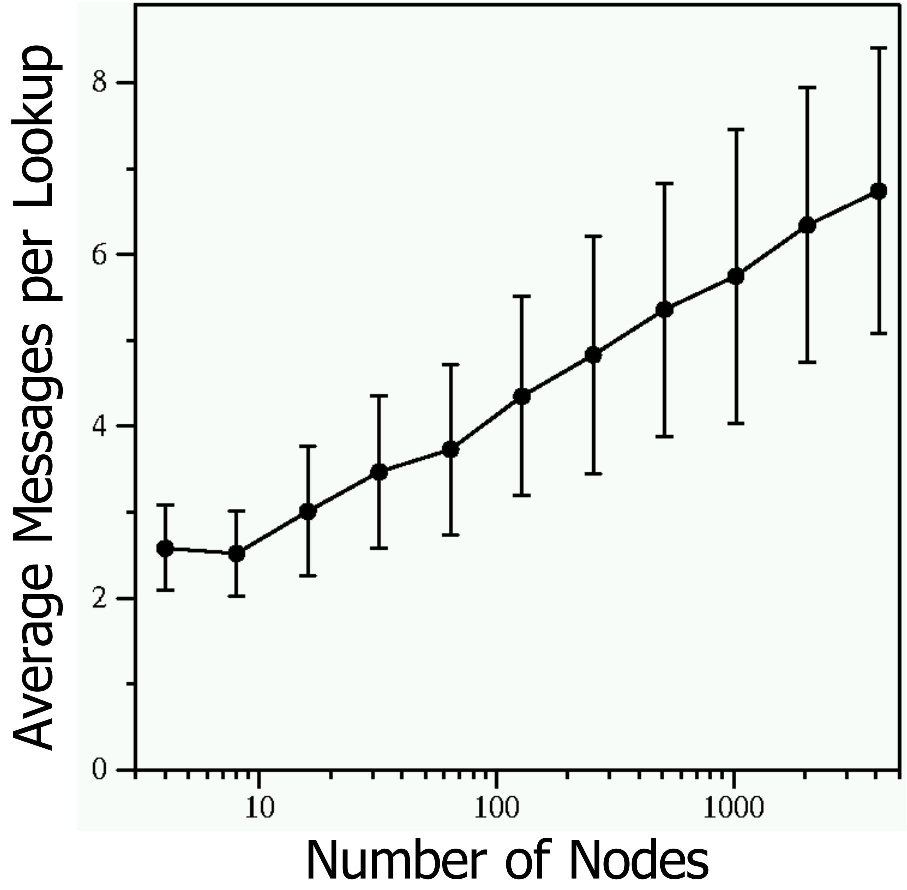
Evaluation Overview

- Quick lookup in large systems
- Low variation in lookup costs
- Robust despite massive failure

- Experiments confirm theoretical results

Cost of lookup

- Cost is O(Log N) as predicted by theory



Summary

- Pioneering work in peer-to-peer networks
- Elegant solution that bridges theory and practice
- Scalability with theoretical guarantees

- Later developments
  - DHTs → Key-value stores, e.g., Amazon Dynamo
  - Distributed applications → Blockchain, Bitcoin, Ethereum, etc.