

The RAMCloud Storage System

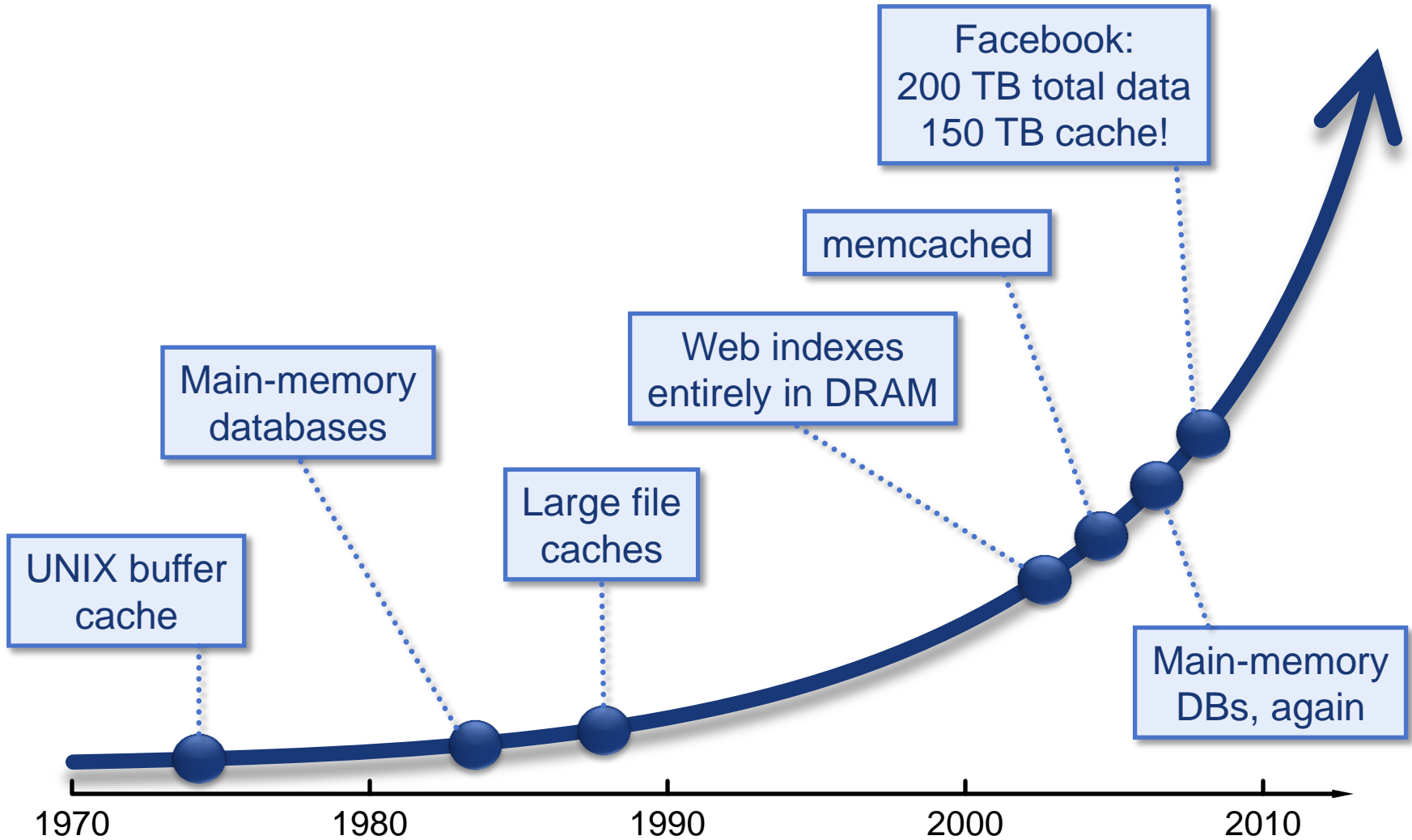
John Ousterhout

**Platform Lab
Stanford University**

Slides available for
download at
<http://goo.gl/13zote>

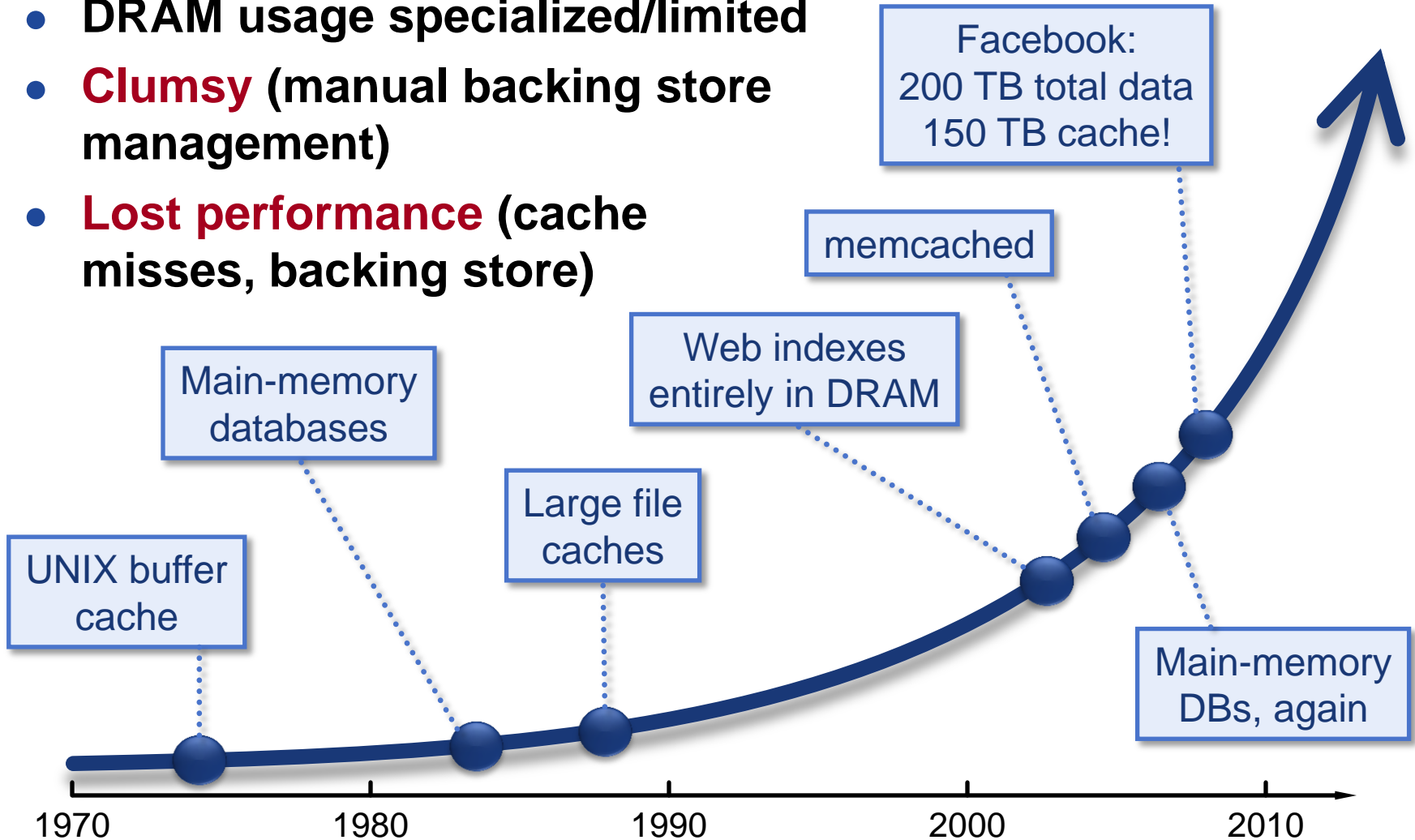


DRAM in Storage Systems



DRAM in Storage Systems

- DRAM usage specialized/limited
- **Clumsy** (manual backing store management)
- **Lost performance** (cache misses, backing store)



RAMCloud

General-purpose DRAM-based storage for large-scale applications:

- All data is stored in DRAM at all times
- As durable and available as disk
- Simple key-value data model
- **Large scale:** 1000+ servers, 100+ TB
- **Low latency:** 5-10 μ s remote access time

Potential impact: enable new class of applications

Performance (Infiniband)

Read 100B object	4.7 μ s
Read bandwidth (large objects, 1 client)	2.7 GB/s
Write 100B object (3x replication)	13.5 μ s
Write bandwidth (large objects, 1 client)	430 MB/s
Single-server throughput:	
Read 100B objects	900 Kobj/s
Multi-read 100B objects	6 Mobj/s
Multi-write 100B objects	450 Kobj/s
Log replay for crash recovery	800 MB/s or 2.3 Mobj/s
Crash recovery time (40 GB data, 80 servers)	1.9 s

Tutorial Outline

Part I: Motivation, Potential Impact

Part II: Overall Architecture

Part III: Log-Structured Storage

Part IV: Low-Latency RPCs

Part V: Crash Recovery

Part VI: Status and Limitations

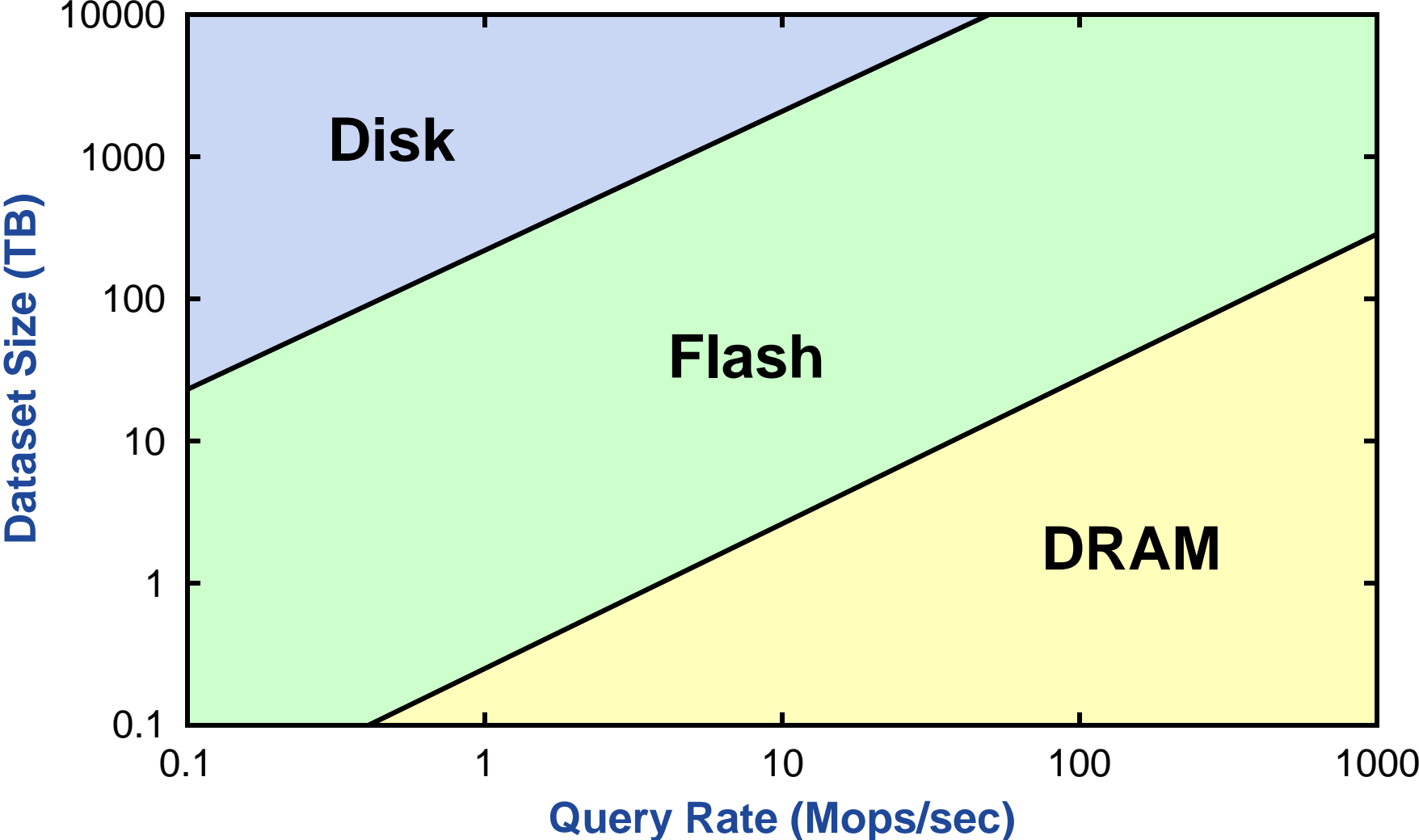
Part VII: Application Experience

Part VIII: Lessons Learned

Part I: Motivation, Potential Impact

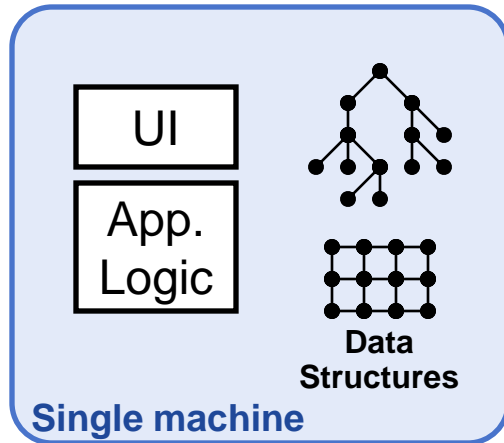


Lowest TCO



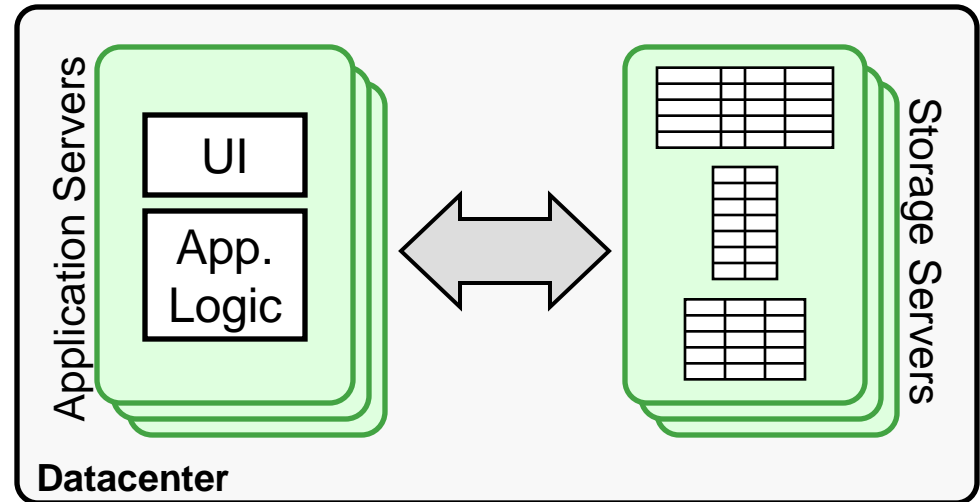
Why Does Latency Matter?

Traditional Application



$\ll 1\mu\text{s}$ latency

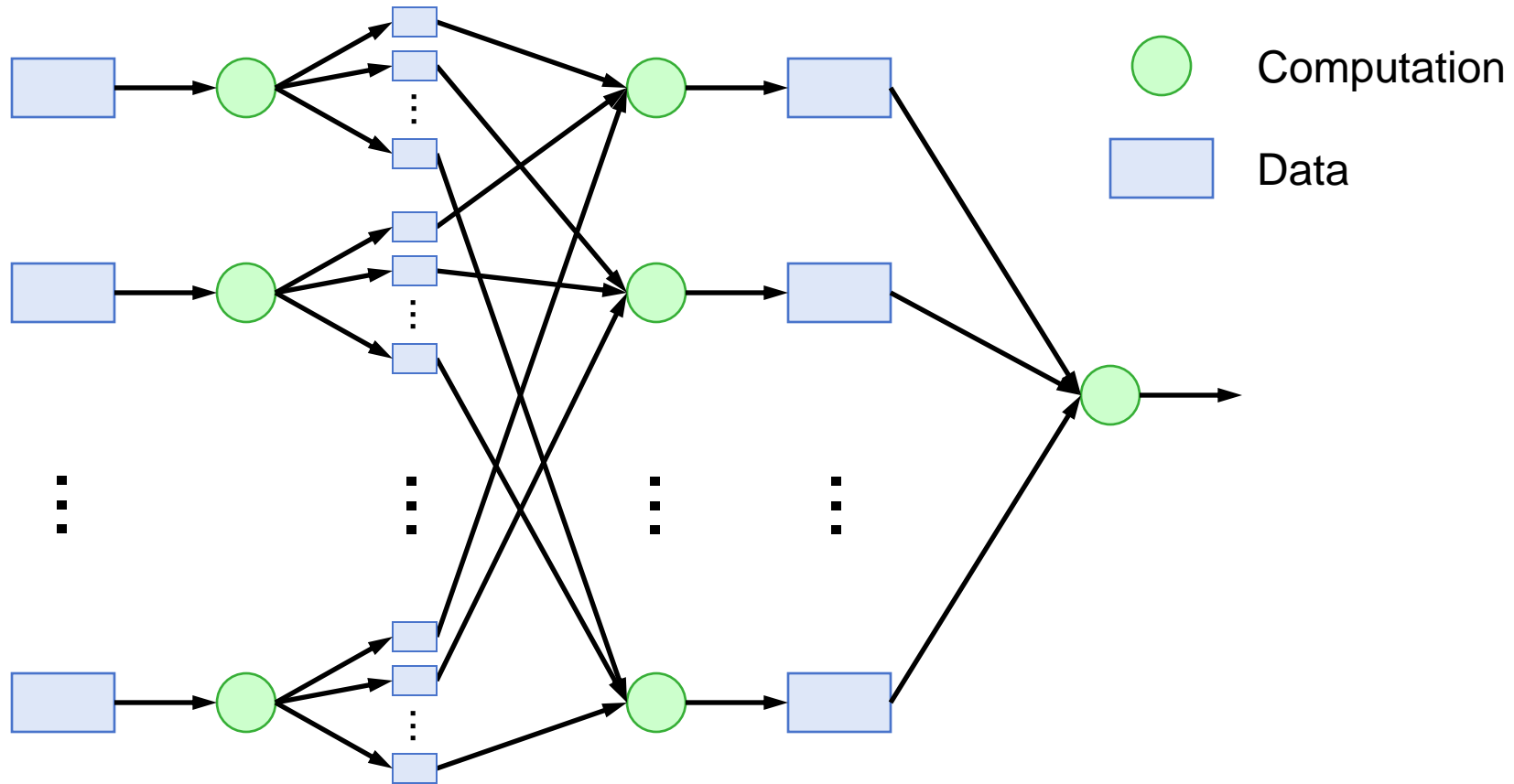
Web Application



0.5-10ms latency

- **Large-scale apps struggle with high latency**
 - Random access data rate has not scaled!
 - Facebook: can only make 100-150 internal requests per page

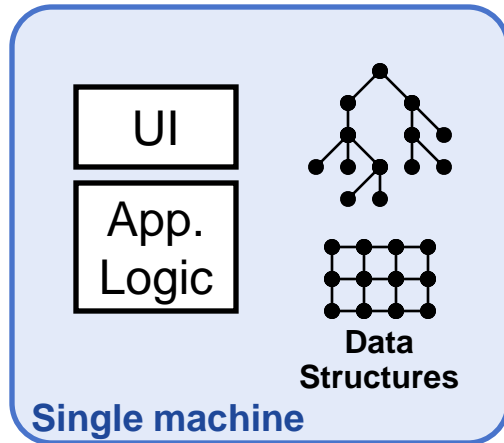
MapReduce



- ✓ **Sequential data access** → high data access rate
- ✗ **Not all applications fit this model**
- ✗ **Offline**

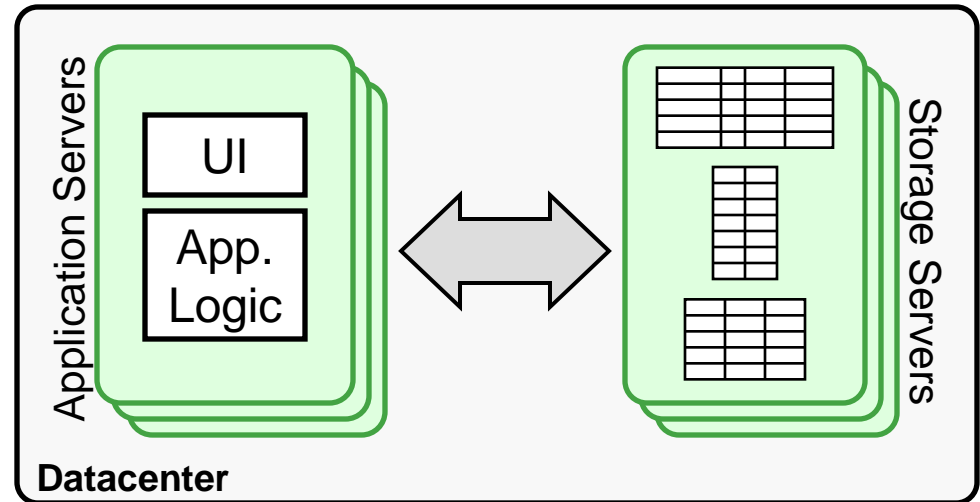
Goal: Scale and Latency

Traditional Application



$\ll 1\mu\text{s}$ latency

Web Application

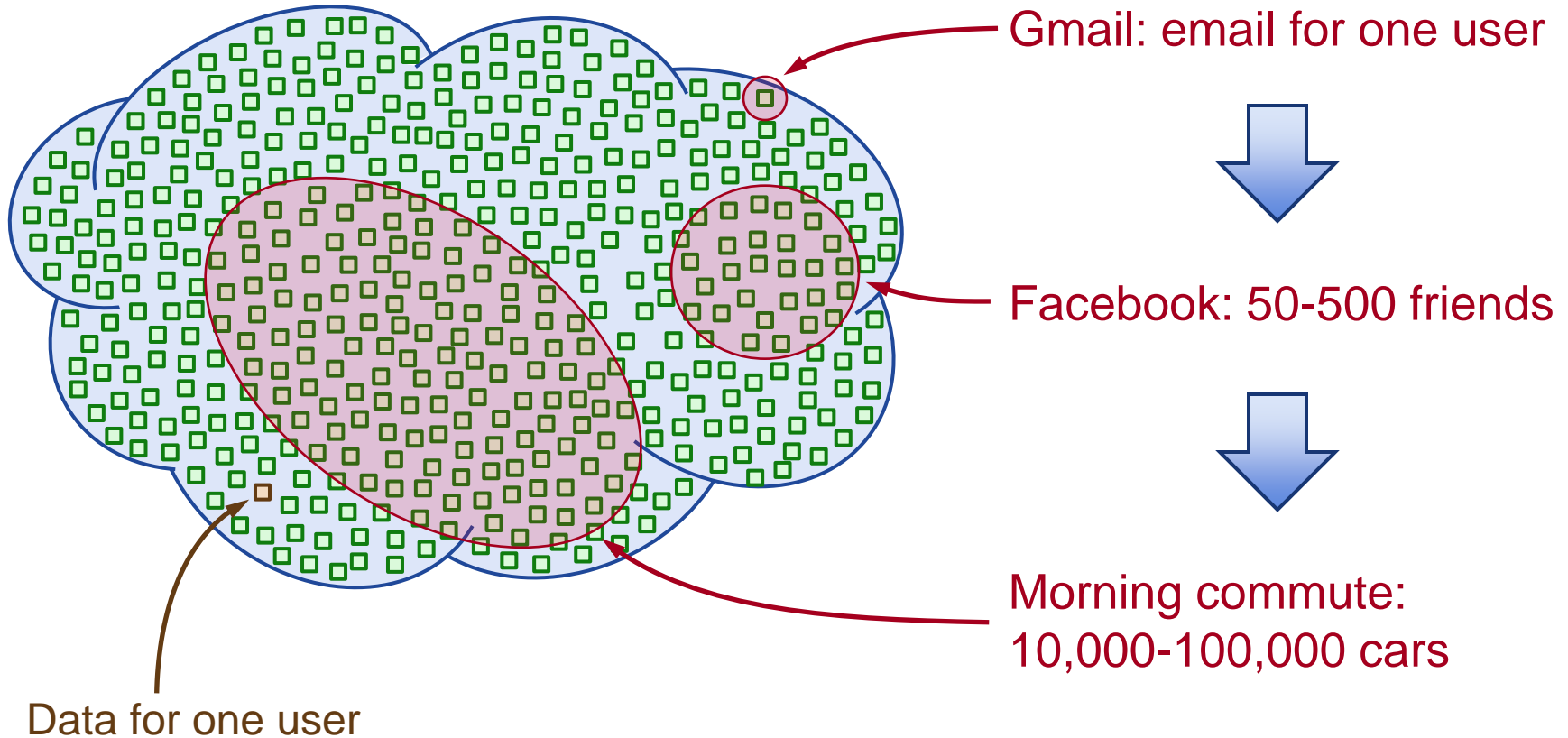


~~0.5-10ms latency~~
5-10 μs

- **Enable new class of applications:**
 - Large-scale graph algorithms (machine learning?)
 - Collaboration at scale?

Large-Scale Collaboration

“Region of Consciousness”



Internet of Things?

Part II: Overall Architecture



Data Model: Key-Value Store

TABLE OPERATIONS

createTable(*name*) → *id*
getTableId(*name*) → *id*
dropTable(*name*)

BASIC OPERATIONS

read(*tableId*, *key*) → *value*, *version*
write(*tableId*, *key*, *value*) → *version*
delete(*tableId*, *key*)

BULK OPERATIONS

multiRead([*tableId*, *key*]*) → [*value*, *version*]*
multiWrite([*tableId*, *key*, *value*]*) → [*version*]*
multiDelete([*tableId*, *key*]*)
enumerateTable(*tableId*) → [*key*, *value*, *version*]*

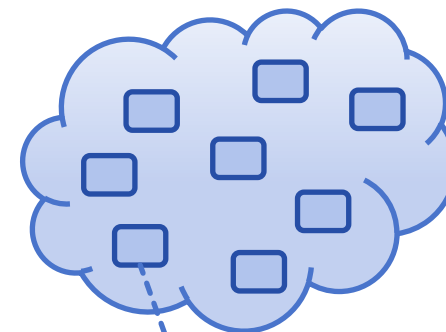
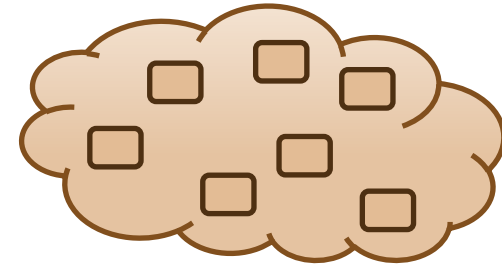
ATOMIC OPERATIONS

increment(*tableId*, *key*, *amount*) → *value*, *version*
conditionalWrite(*tableId*, *key*, *value*, *version*) → *version*

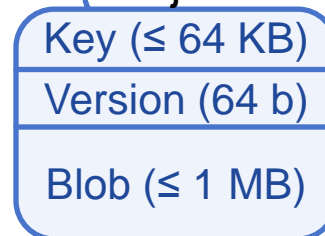
MANAGEMENT OPERATIONS

splitTablet(*tableId*, *keyHash*)
migrateTablet(*tableId*, *keyHash*, *newMaster*)

Tables



Object

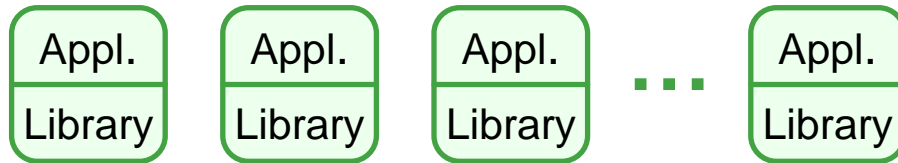


RAMCloud Data Model, cont'd

- **Goal: strong consistency (linearizability)**
 - Not yet fully implemented
- **Secondary indexes and multi-object transactions:**
 - Useful for developers
 - Not implemented in RAMCloud 1.0
 - Currently under development

RAMCloud Architecture

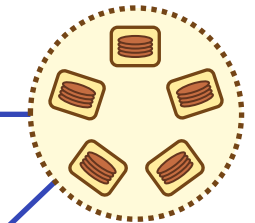
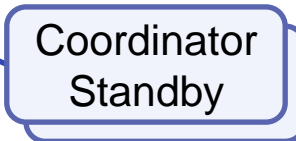
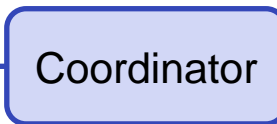
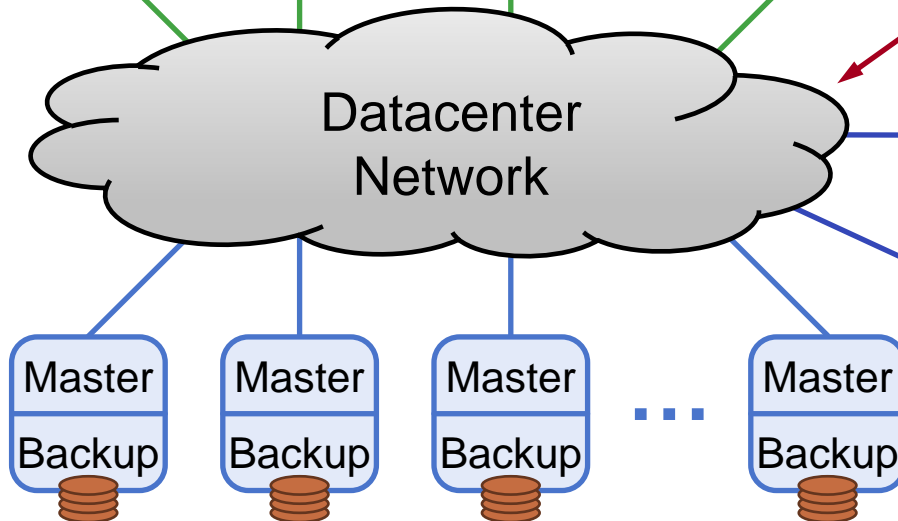
1000 – 100,000 Application Servers



High-speed networking:

- 5 μ s round-trip
- Full bisection bandwidth

Commodity Servers



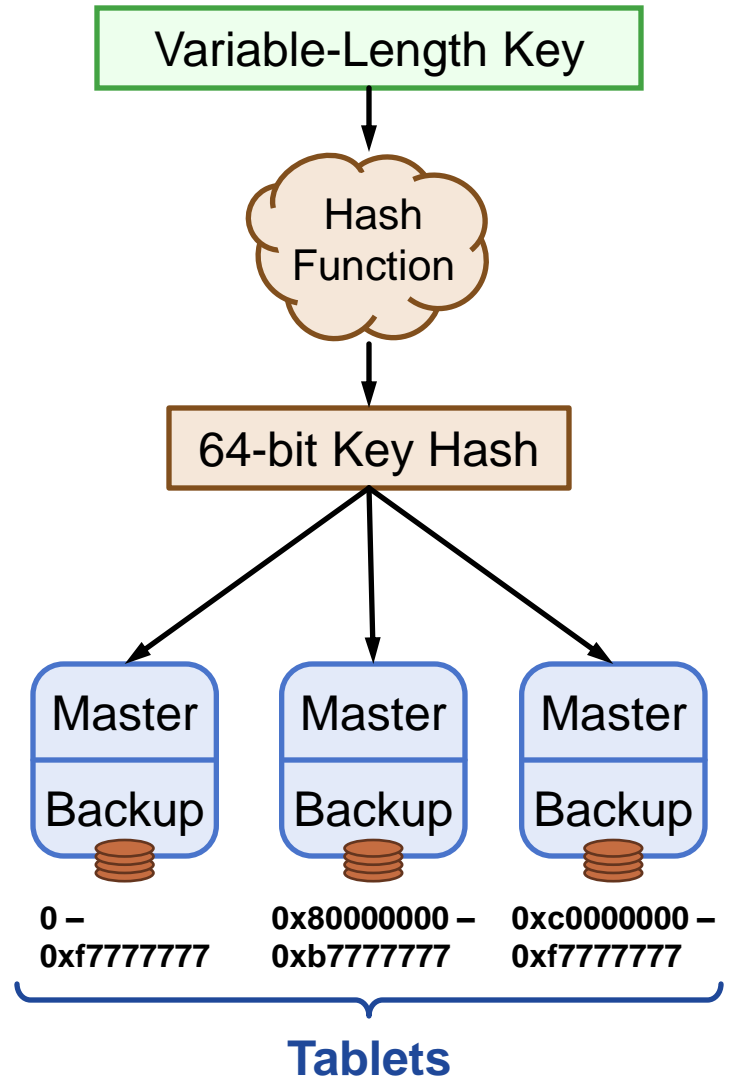
External Storage (ZooKeeper)

64-256 GB per server

1000 – 10,000 Storage Servers

Hash Partitioning

- Tables divided into **tablets** by key hash
- **Tablet**: unit of allocation to servers
- **Small tables**: single tablet
- **Large tables**: multiple tablets on different servers
- Each server stores multiple tablets
- **Currently no automatic reconfiguration**



Example Configurations

	2010	2015–2020
# servers	2000	4000
GB/server	24GB	256GB
Total capacity	48TB	1PB
Total server cost	\$3.1M	\$6M
\$/GB	\$65	\$6

For \$100-200K today:

- One year of Amazon customer orders
- One year of United flight reservations

Part III: Log-Structured Storage



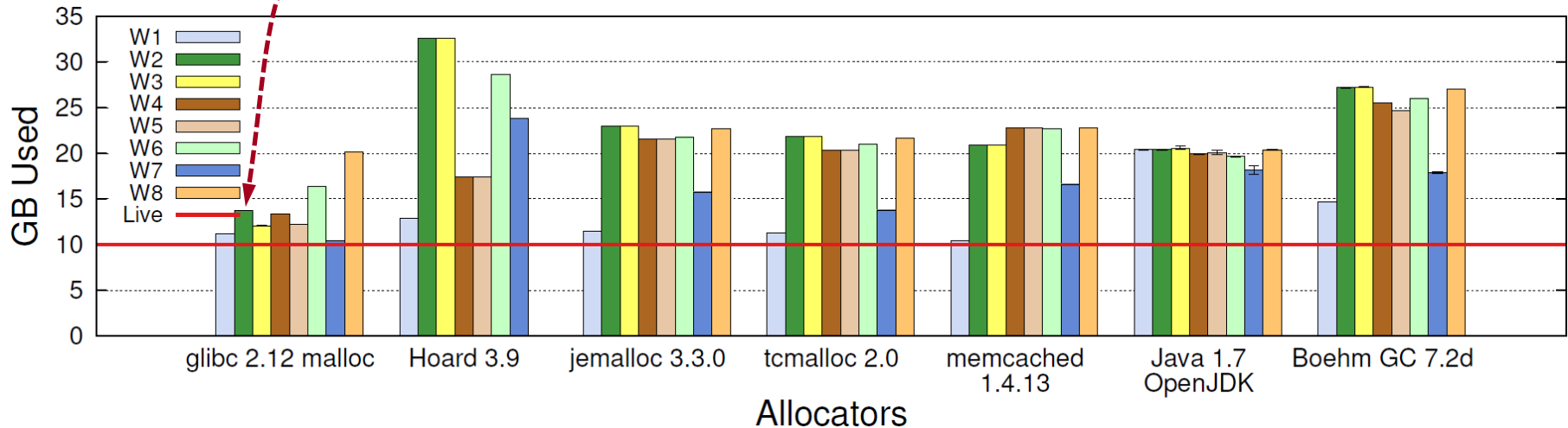
Storage System Requirements

- **High performance**
 - Read/write latency not impacted by secondary storage speed
- **Efficient use of DRAM**
 - DRAM \approx 50% of system cost
 - Goal: 80-90% DRAM utilization
- **Durability/availability \geq replicated disk**
- **Scalable**
 - Increase capacity/performance by adding servers
 - Minimize centralized functionality

Existing Allocators Waste Memory

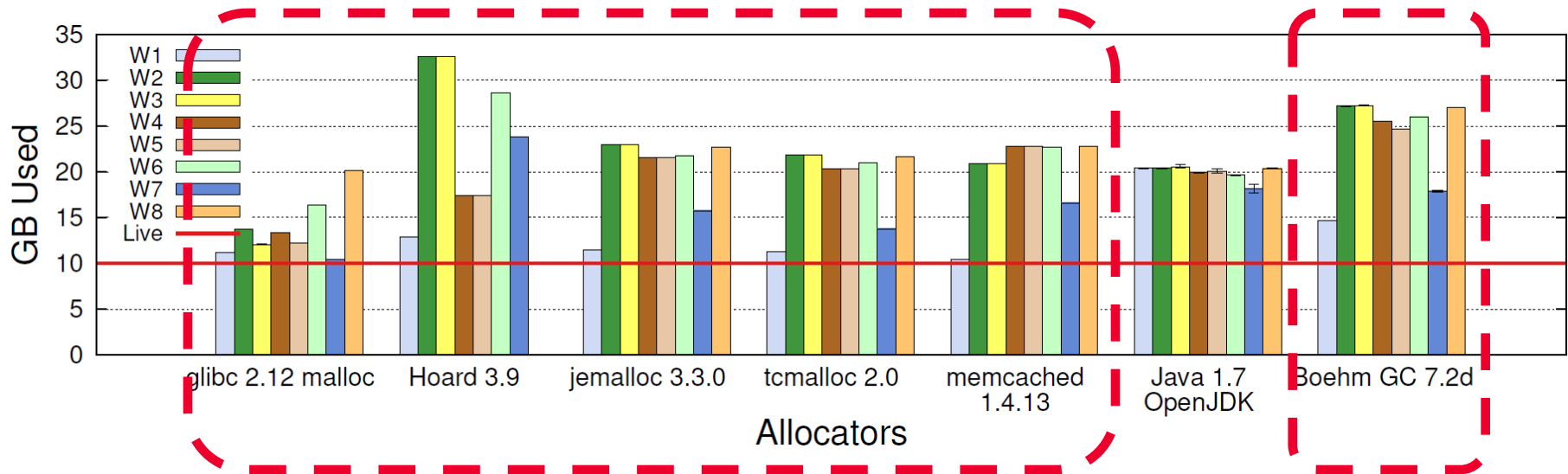
glibc malloc: 13.7 GB memory to hold 10 GB data
under workload W2:

- Allocate many 100B objects
- Gradually overwrite with 130B objects

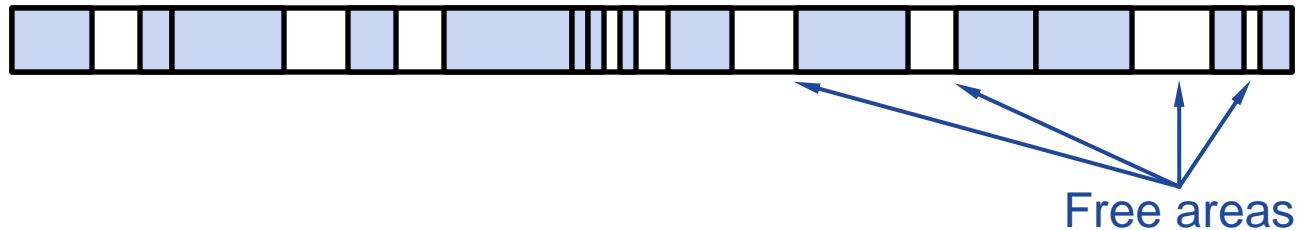


- **7 memory allocators, 8 synthetic workloads**
 - Total live data constant (10 GB)
 - But **workload changes** (except W1)
- **All allocators waste at least 50% of memory in some situations**

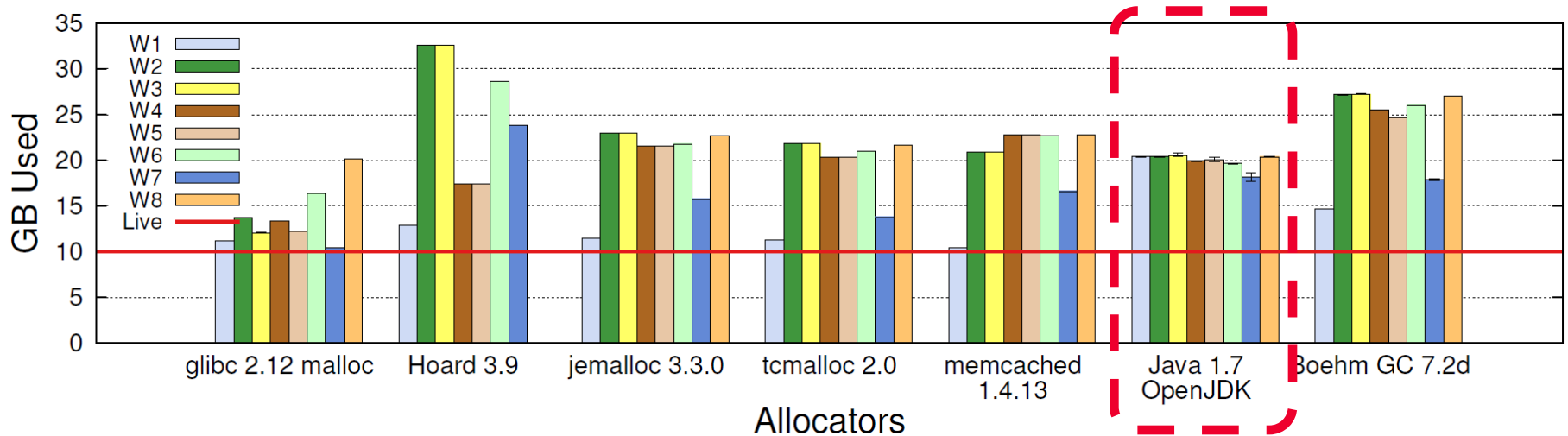
Non-Copying Allocators



- **Blocks cannot be moved once allocated**
- **Result: fragmentation**



Copying Garbage Collectors



Before collection:



After collection:



- **Must scan all memory to update pointers**
 - Expensive, scales poorly
 - Wait for lots of free space before running GC
- **State of the art: 3-5x overallocation of memory**
- **Long pauses: 3+ seconds for full GC**

Allocator for RAMCloud

- **Requirements:**
 - Must use copying approach
 - Must collect free space incrementally
- **Storage system advantage: pointers restricted**
 - Pointers stored in index structures
 - Easy to locate pointers for a given memory block
 - **Enables incremental copying**
- **Solution: log-structured storage**

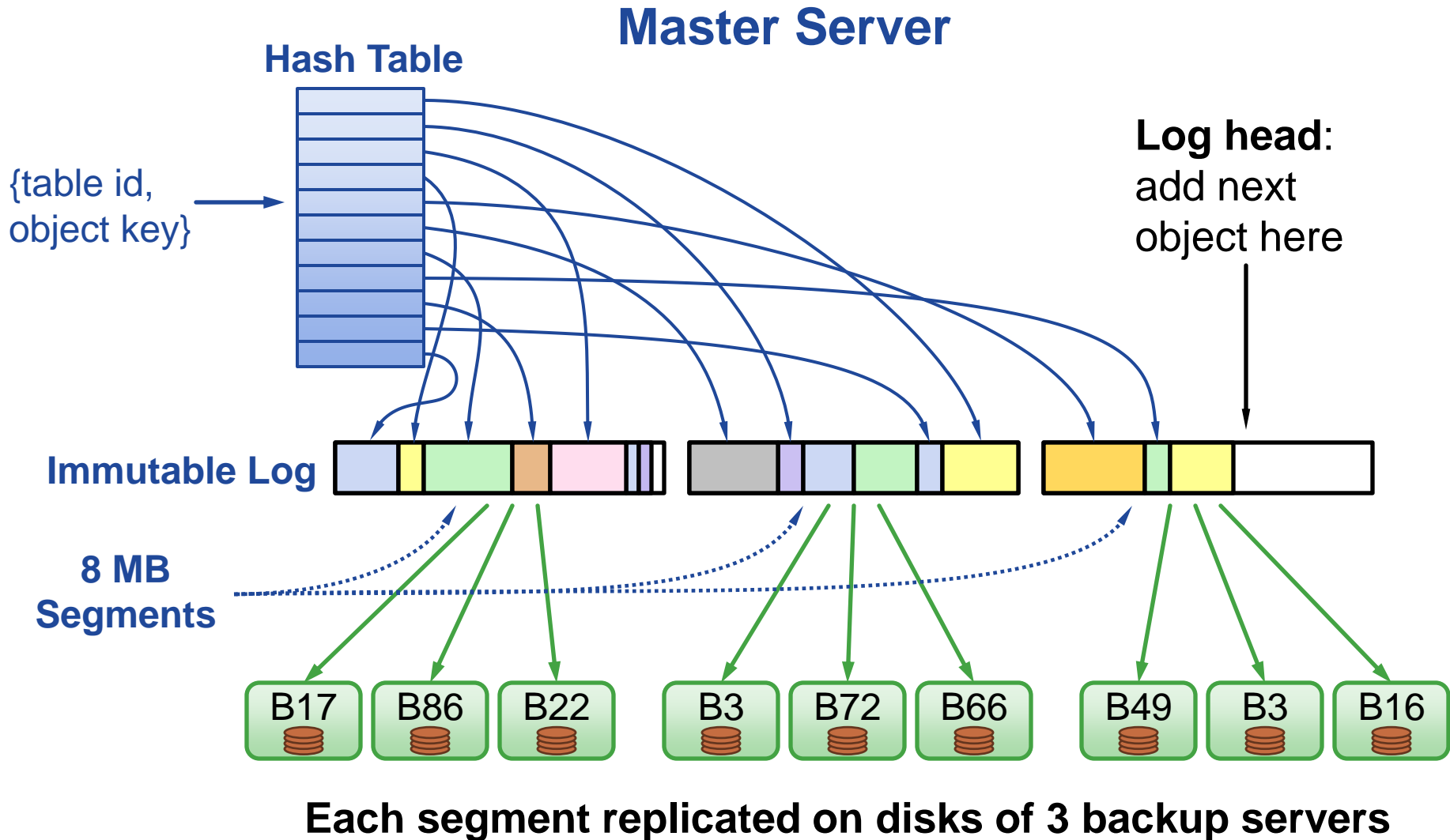
Durability/Availability

- **All data must be replicated**
- **Replication in DRAM?**
 - Expensive
 - Insufficient (power failures)
- **RAMCloud: primary-backup approach:**
 - One copy in DRAM
 - Multiple copies on secondary storage (disk/flash)
 - Must **recover quickly** after crashes
- **Challenge: secondary storage latency**
 - Must not affect RAMCloud access times

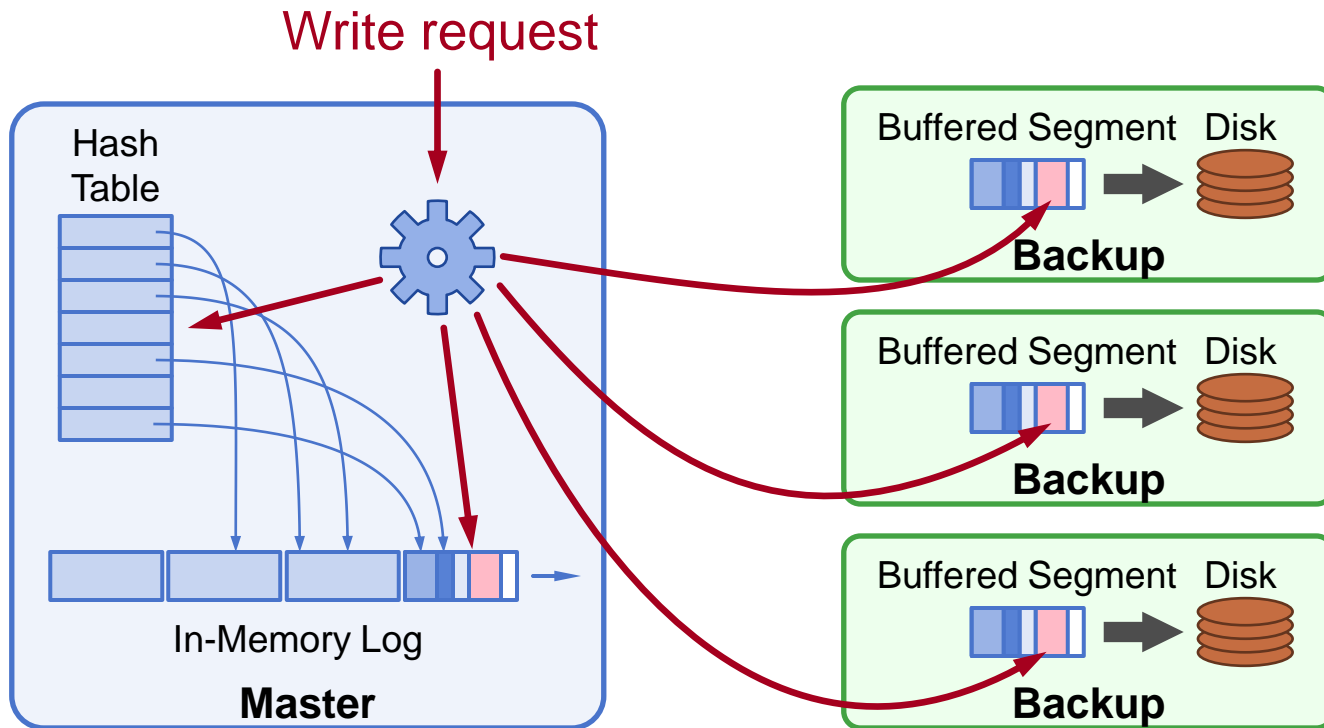
Log-Structured Storage

- **Store all data in append-only logs:**
 - One log per master
 - Both DRAM and secondary storage
 - Similar to log-structured file systems
- **Benefits:**
 - Fast allocation
 - High throughput: batched updates to secondary storage
 - 80-90% memory utilization
 - Insensitive to workload changes
 - Crash recovery: replay log
 - Consistency: serializes operations

Log-Structured Storage



Durable Writes



- **No disk I/O during write requests**
- **Backups perform I/O in background**
- **Buffer memory must be non-volatile (NVDIMMs?)**

Logs on Secondary Storage

- **Log on disk/fla**
- **Except during crash recovery**
- **During recovery, read entire log for a master**

Log Entry Types

Object

Table Id	Key	Version	Timestamp	Value
----------	-----	---------	-----------	-------

Tombstone (identifies dead object)

Table Id	Key	Version	Segment Id
----------	-----	---------	------------

Segment Header

Master Id	Segment Id
-----------	------------

Log Digest (identifies all segments in log)

Segment Id	Segment Id	...	Segment Id
------------	------------	-----	------------

Tablet Statistics

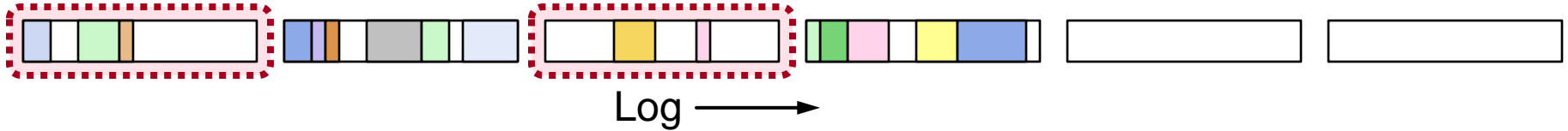
For each tablet: # log entries, log bytes (compressed)
--

Safe Version

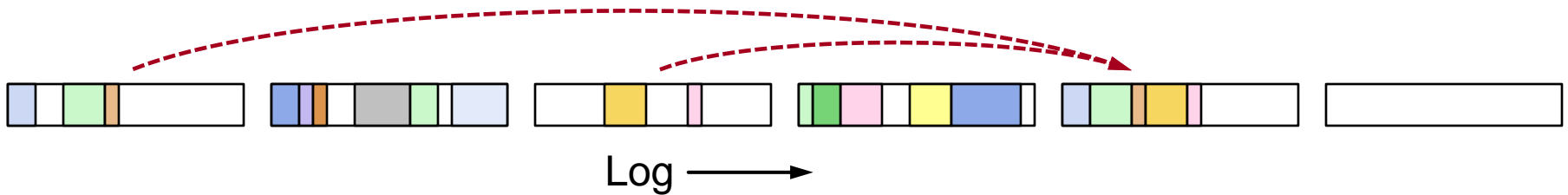
Version # larger than any used on master
--

Log Cleaning

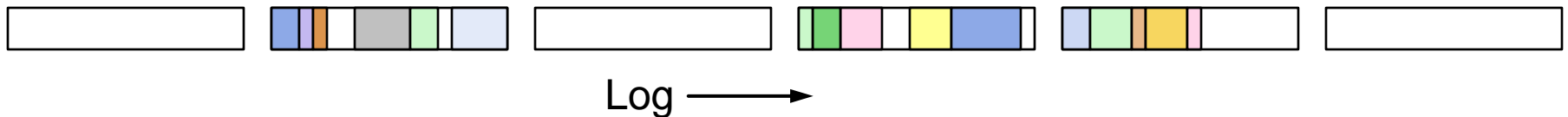
1. Pick segments with lots of free space:



2. Copy live objects (survivors):



3. Free cleaned segments (and backup replicas)



Cleaning is incremental

Tombstones

- **How to prevent reincarnation during crash recovery?**
- **Tombstones:**
 - Written into log when object deleted or overwritten:
 - Table id
 - Object key
 - Version of dead object
 - Id of segment where object stored
- **When can tombstone be cleaned?**
 - After segment containing object has been cleaned (and replicas deleted on backups)
- **Note: tombstones are a mixed blessing**

Cleaning Cost

U: fraction of live bytes
in cleaned segments

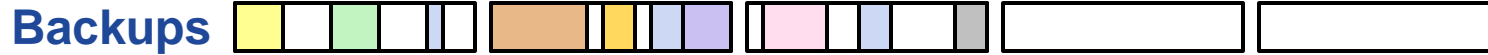
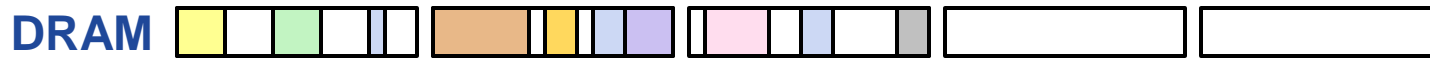
	0.5	0.9	0.99
Bytes copied by cleaner (U)	0.5	0.9	0.99
Bytes freed (1-U)	0.5	0.1	0.01
Bytes copied/byte freed (U/(1-U))	1.0	9.0	99.0

.....

		Capacity	Bandwidth
Conflicting Needs:	Memory	expensive	cheap
	Disk	cheap	expensive

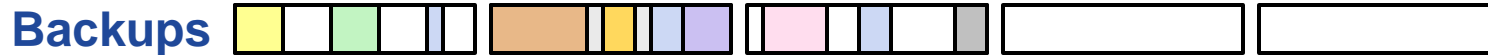
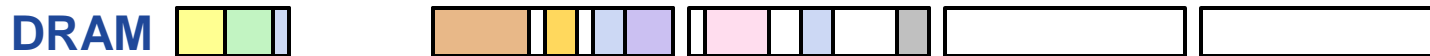
Need different policies for cleaning disk and memory

Two-Level Cleaning



Compaction:

- Clean single segment in memory
- No change to replicas on backups



Combined Cleaning:

- Clean multiple segments
- Free old segments (disk & memory)

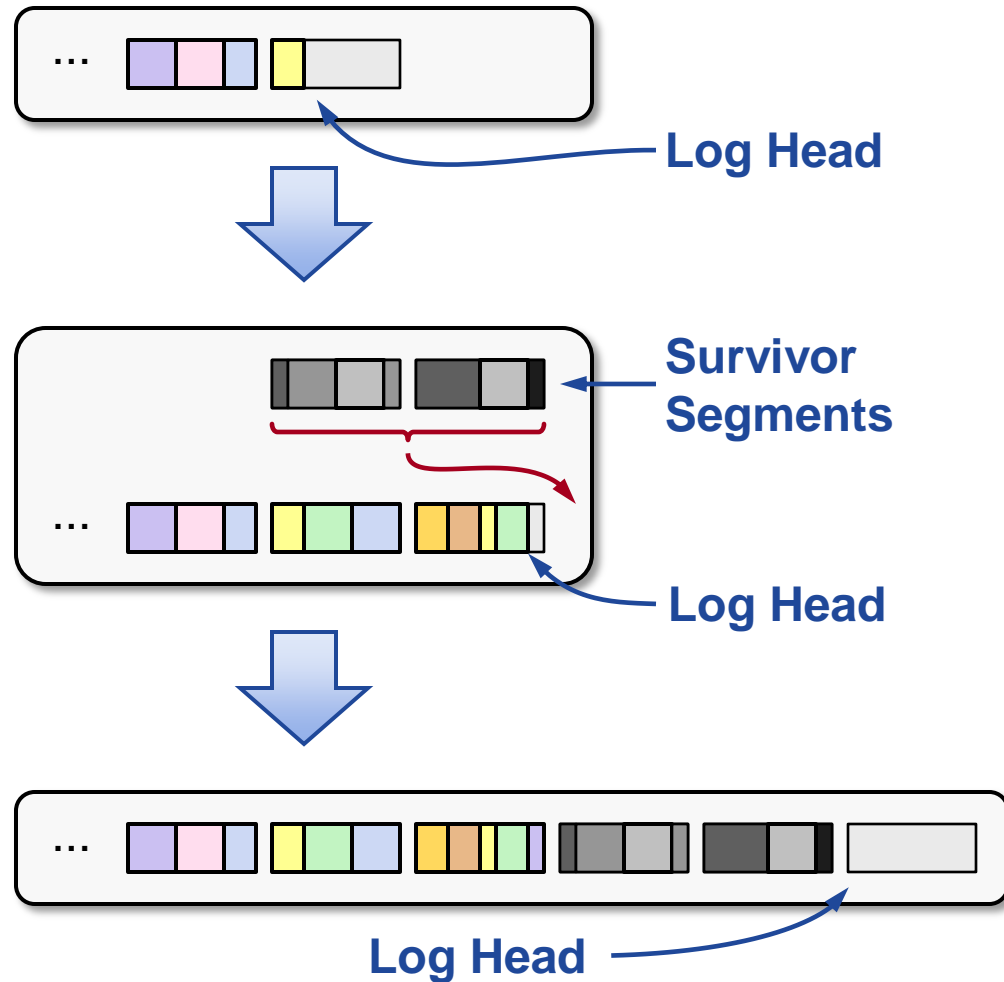


Two-Level Cleaning, cont'd

- **Best of both worlds:**
 - Optimize utilization of memory
(can afford high bandwidth cost for compaction)
 - Optimize disk bandwidth
(can afford extra disk space to reduce cleaning cost)
- **But:**
 - Segments in DRAM no longer fixed-size
(implement with 128 KB [seglets](#))
 - Compaction cannot clean tombstones
(must eventually perform combined cleaning)

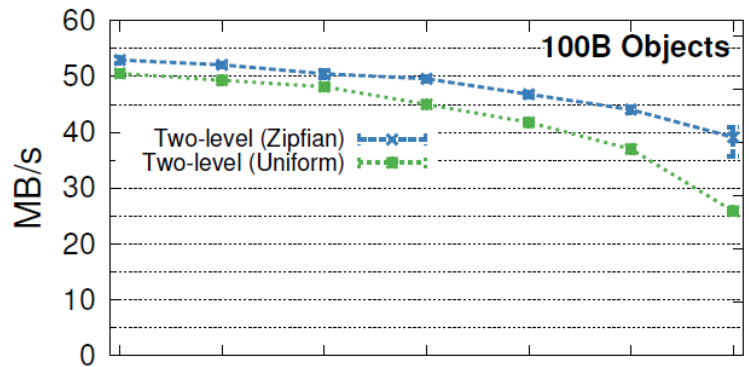
Parallel Cleaning

- **Survivor data written to “side log”**
 - No competition for log head
 - Different backups for replicas
- **Synchronization points:**
 - Updates to hash table
 - Adding survivor segments to log
 - Freeing cleaned segments

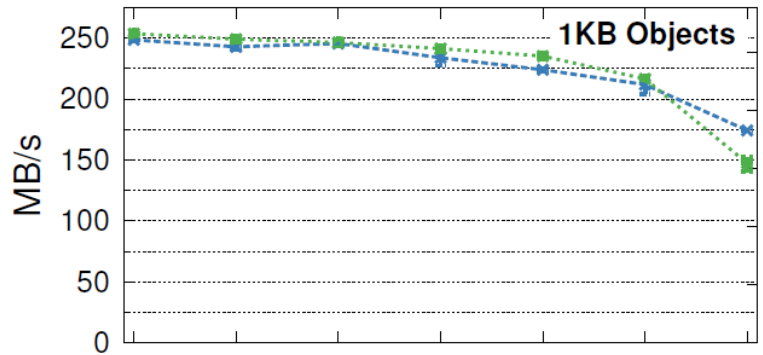


Throughput vs. Memory Utilization

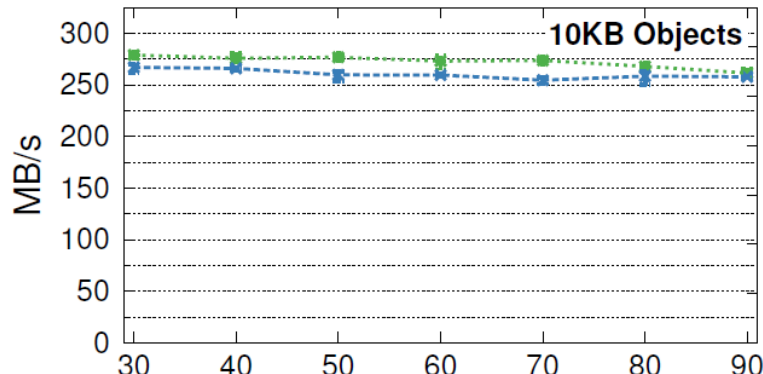
1 master,
3 backups,
1 client,
concurrent
multi-writes



Memory Utilization	Performance Degradation
80%	17-27%
90%	26-49%

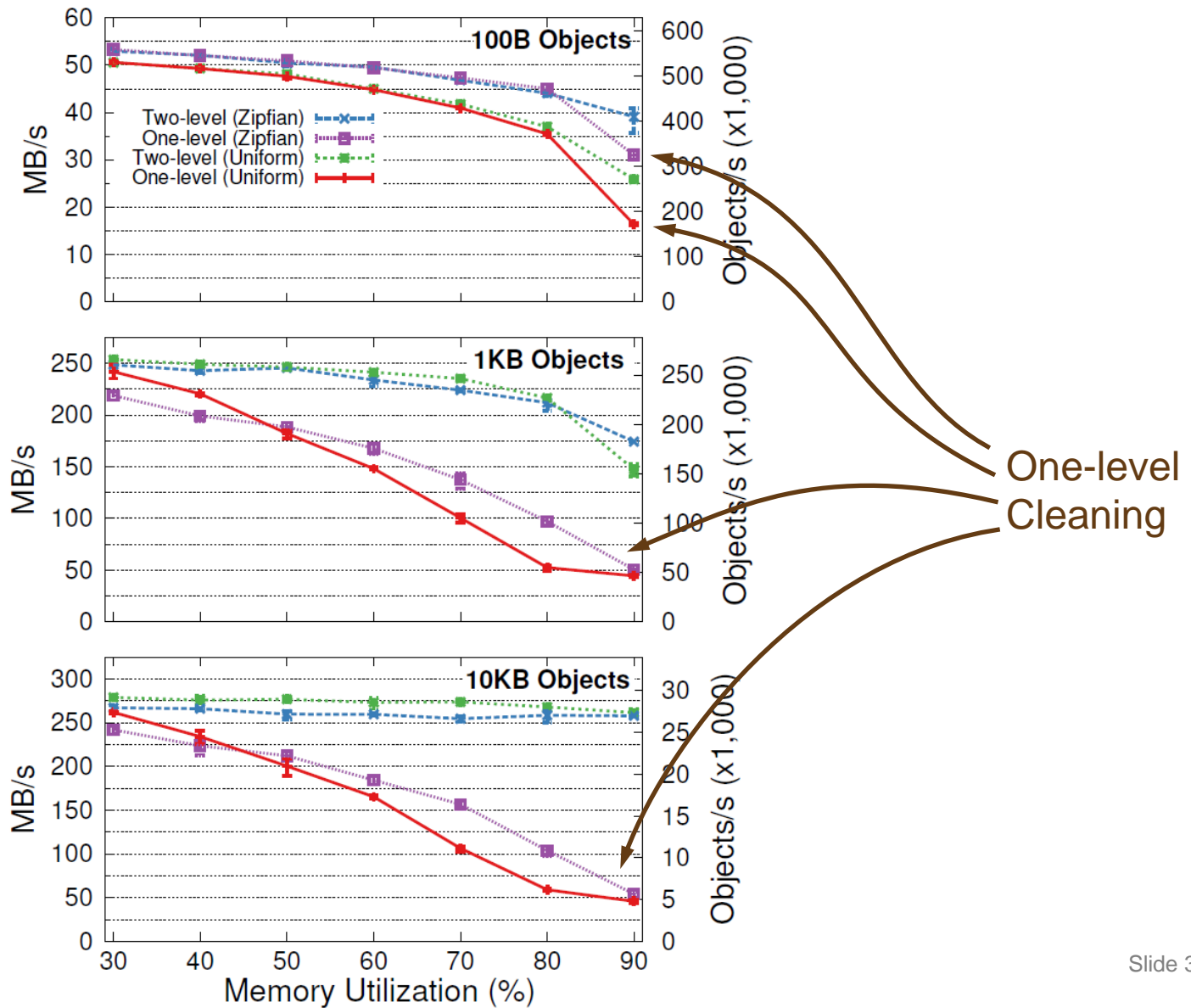


Memory Utilization	Performance Degradation
80%	14-15%
90%	30-42%



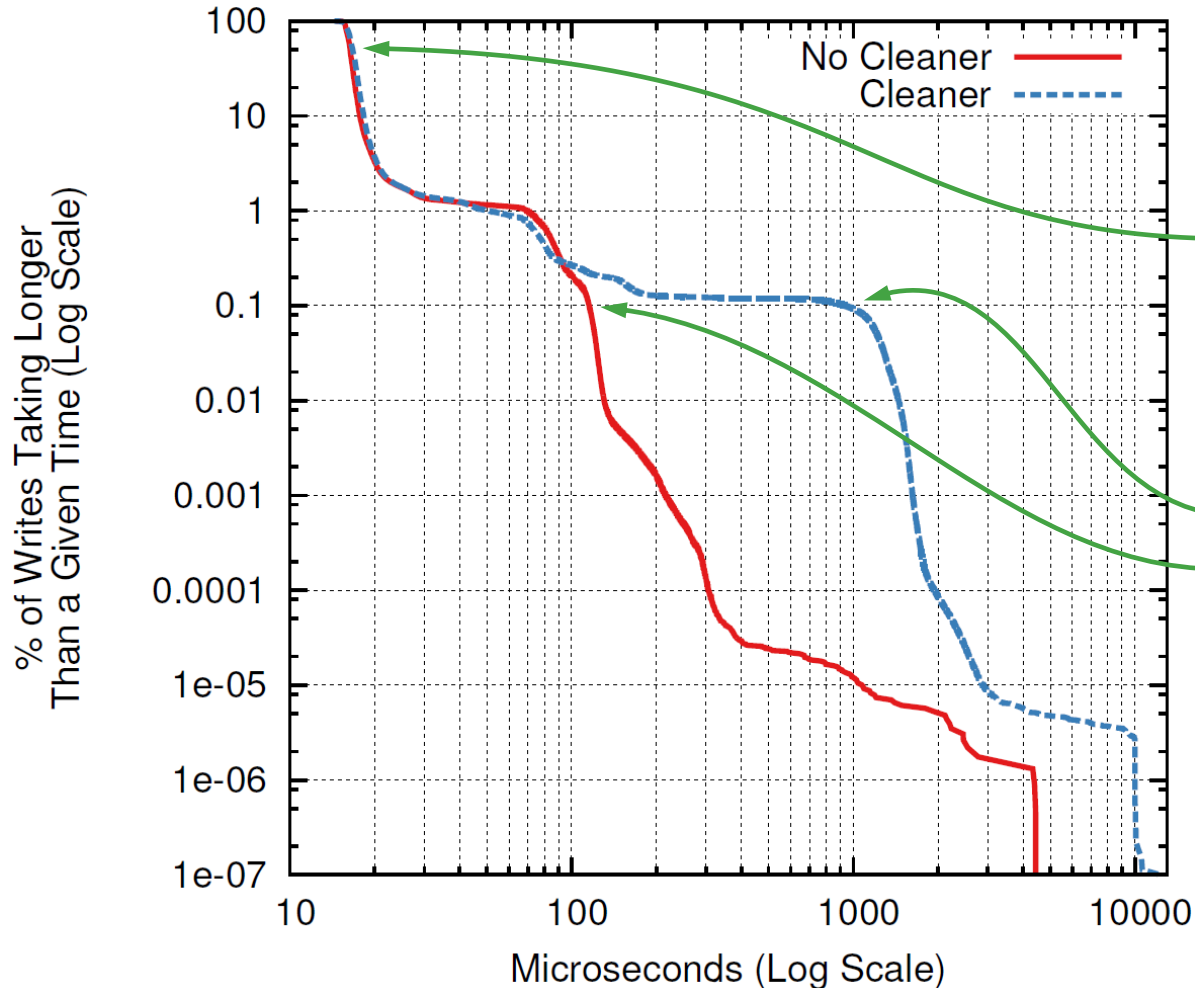
Memory Utilization	Performance Degradation
80%	3-4%
90%	3-6%

1-Level vs. 2-Level Cleaning



Cleaner's Impact on Latency

1 client, sequential 100B overwrites, no locality, 90% utilization



Median:

- With cleaning: 16.70 μ s
- No cleaner: 16.35 μ s

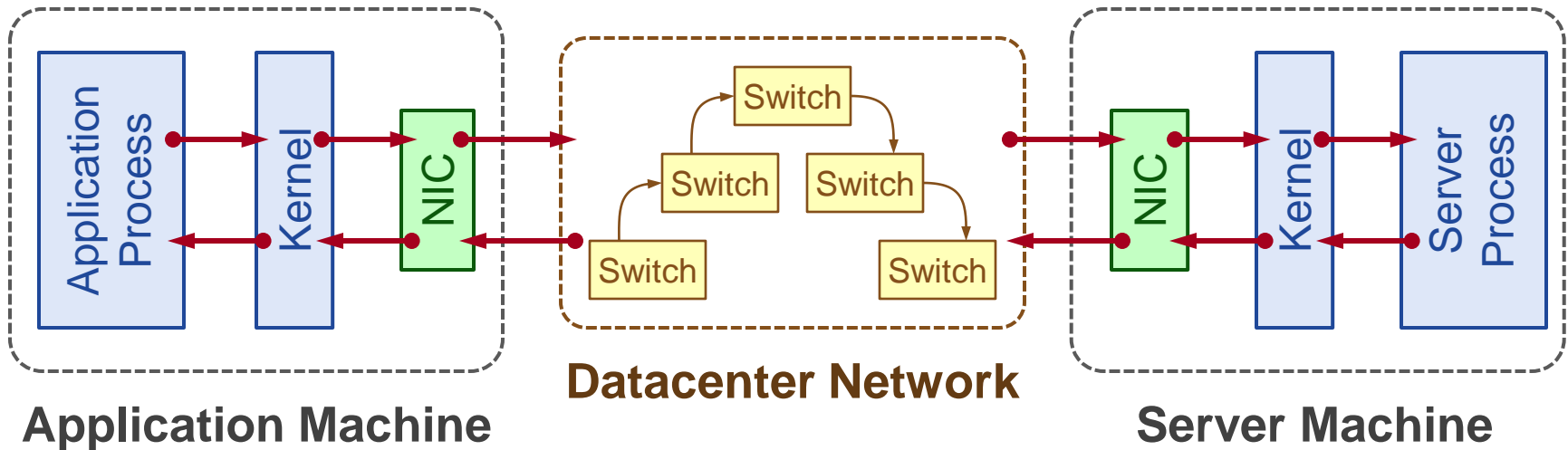
99.9th %ile:

- With cleaning: 900 μ s
- No cleaner: 115 μ s

Part IV: Low-Latency RPCs



Datacenter Latency in 2009



Component	Delay	Round-trip
Network switch	10-30 μs	100-300 μs
OS protocol stack	15 μs	60 μs
Network interface controller (NIC)	2.5-32 μs	2-128 μs
Propagation delay	0.5 μs	1.0 μs

Typical in 2009: **200-400 μs** RAMCloud goal: **5-10 μs**

How to Improve Latency

- **Network switches (10-30 μ s per switch in 2009):**
 - 10Gbit switches: 500 ns per switch
 - **Radical redesign: 30 ns per switch**
 - Must eliminate buffering
- **Software (60 μ s total in 2009):**
 - Kernel bypass: 2 μ s
 - Direct NIC access from applications
 - Polling instead of interrupts
 - **New protocols, threading architectures: 1 μ s**
- **NIC (2-32 μ s per transit in 2009):**
 - Optimize current architectures: 0.75 μ s per transit
 - **Radical NIC CPU integration: 50 ns per transit**

Round-Trip Delay, Revisited

Component	2009	2015	Limit
Switching fabric	100-300 μs	5 μs	0.2 μs
Operating system	60 μs	0 μs	0 μs
Application/server	2 μs	2 μs	1 μs
NIC	8-128 μs	3 μs	0.2 μs
Propagation delay	1 μs	1 μs	1 μs
Total	200-400 μs	11 μs	2.4 μs

- **Biggest remaining hurdles:**
 - Software
 - Speed of light

RAMCloud Goal: 1 μ s Service Time

- **Can't afford many L3 cache misses (< 10?)**
- **Can't afford much synchronization**
 - Acquire-release spin lock (no cache misses): 16 ns
- **Can't afford kernel calls**
- **Can't afford batching**
 - Trade-off between bandwidth and latency

Low Latency in RAMCloud

- **Kernel bypass:**

- Map virtual NIC into application address space
- Originally developed for Infiniband (Mellanox)
- Now becoming available for 10 GigE (Intel, SolarFlare, etc.)
 - Driven by demand for faster virtual machines
 - Newer Mellanox NICs also support 10 GigE
 - Latency unimpressive for many NICs (RPC round-trip 2x Mellanox)

- **Polling:**

- Client spins while waiting for RPC response
 - Response time < context switch time
 - Condition variable wakeup takes 2 μ s
- Server spins while waiting for incoming request
 - Burns 1 core even when idle

Transports

- **Encapsulate different approaches to networking**

- Service naming
- Reliable delivery of request & response messages

- **Client APIs:**

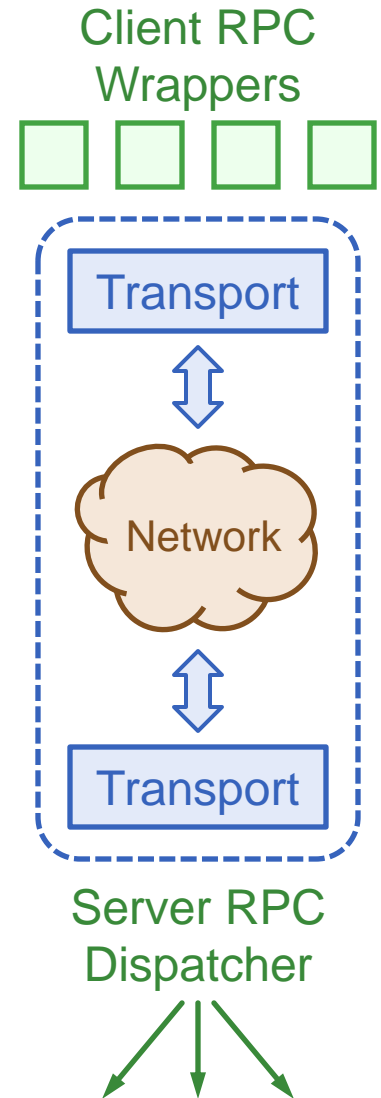
```
session = transport->getSession(  
    serviceLocator);
```

```
session->sendRequest(request,  
    response);
```

```
response->isReady();
```

- **Server API (callout):**

```
handleRpc(request) → response
```



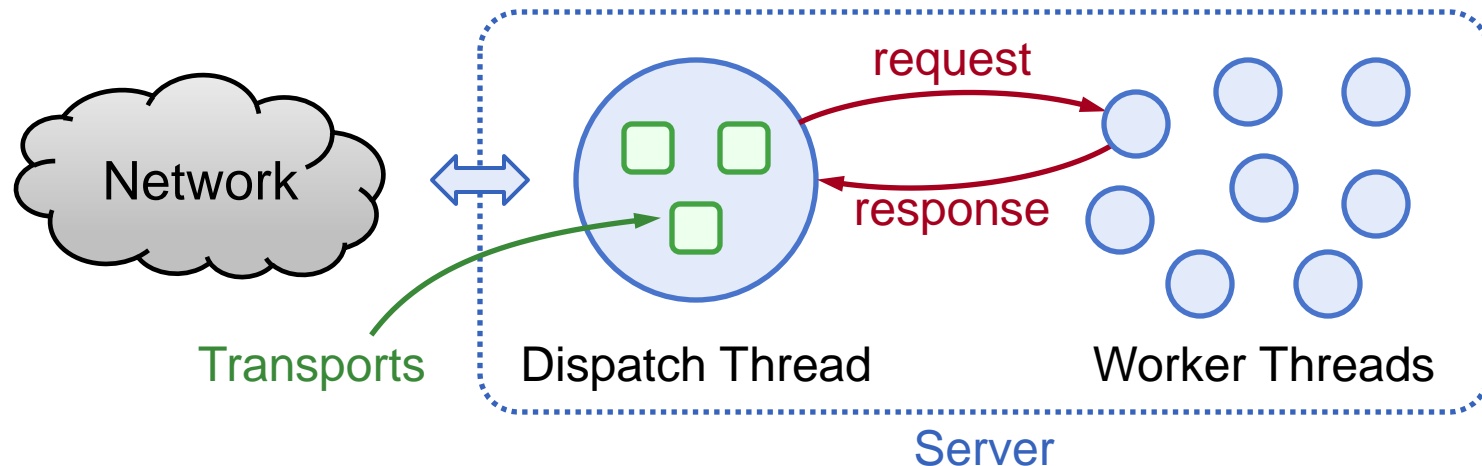
Current Transports

- **InfRcTransport**
 - Uses Infiniband Verbs APIs (reliable connected queue pairs)
 - Supports kernel bypass
 - Our workhorse transport (4.7 μ s for 100B reads)
- **TcpTransport**
 - Uses kernel TCP sockets
 - Slow (50-150 μ s for 100B reads)
- **FastTransport**
 - Custom protocol (reliable, flow-controlled, in-order delivery)
 - Layered on unreliable datagram drivers
 - Current drivers:
 - Kernel UDP
 - Infiniband unreliable datagrams (kernel bypass)
 - SolarFlare (10 GigE with kernel bypass)
 - Not yet as fast as InfRcTransport....

Threading Architecture

- **Initial implementation: single-threaded**
 - No synchronization overhead
 - Minimizes latency
- **Fragile:**
 - Can't process heartbeats during long-running requests
 - Callers will assume server crashed
 - "Crashes" cascade
- **Vulnerable to distributed deadlock:**
 - Nested RPCs sometimes needed:
 - E.g, replication during writes
 - All resources can be consumed with top-level requests

Dispatch Thread and Workers



- **Dispatch thread:**

- Runs all transports
- Polls network for input; never sleeps
- Dispatches requests to workers
- Thread limits for different request classes: prevent deadlock

- **Worker thread:**

- Processes RPC requests
- Returns responses to dispatch thread
- Polls to wait for next request; eventually sleeps

Threads are Expensive!

- **Latency for thread handoffs:**
 - 100ns in each direction
- **Shared state between dispatch and worker threads:**
 - Request/response buffers, etc.
 - >20 L2 additional cache misses to migrate state
- **Total cost of threading: ~450 ns in latency**
- **Dispatch thread is also throughput bottleneck**

We are still looking for better alternatives...

Infiniband Latency (μs)

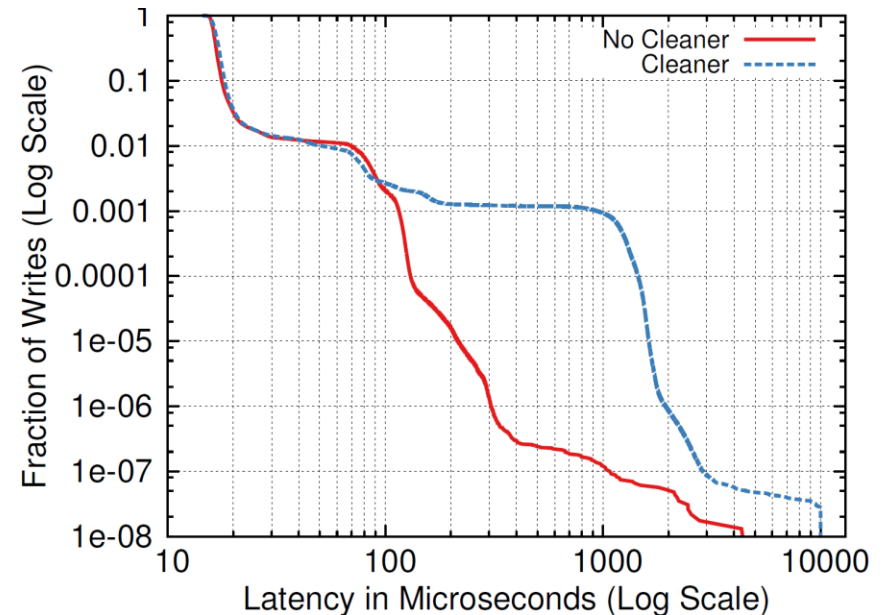
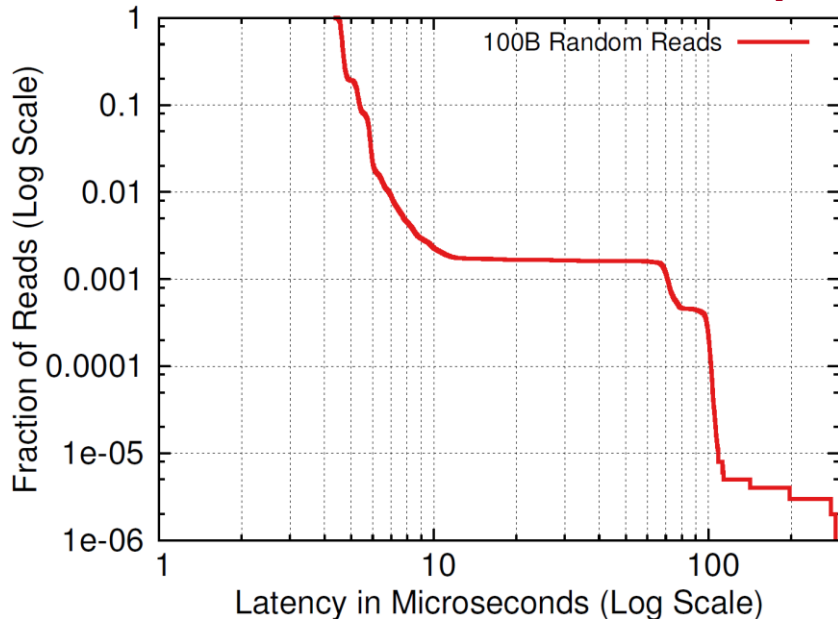
Reads

Object Size	Median	90%	99%	99.9%
100	4.7	5.4	6.4	9.2
1000	7.0	7.7	8.9	12.0
10000	10.1	11.1	12.3	28.5
100000	42.8	44.0	45.3	85.6
1000000	358	364	367	401

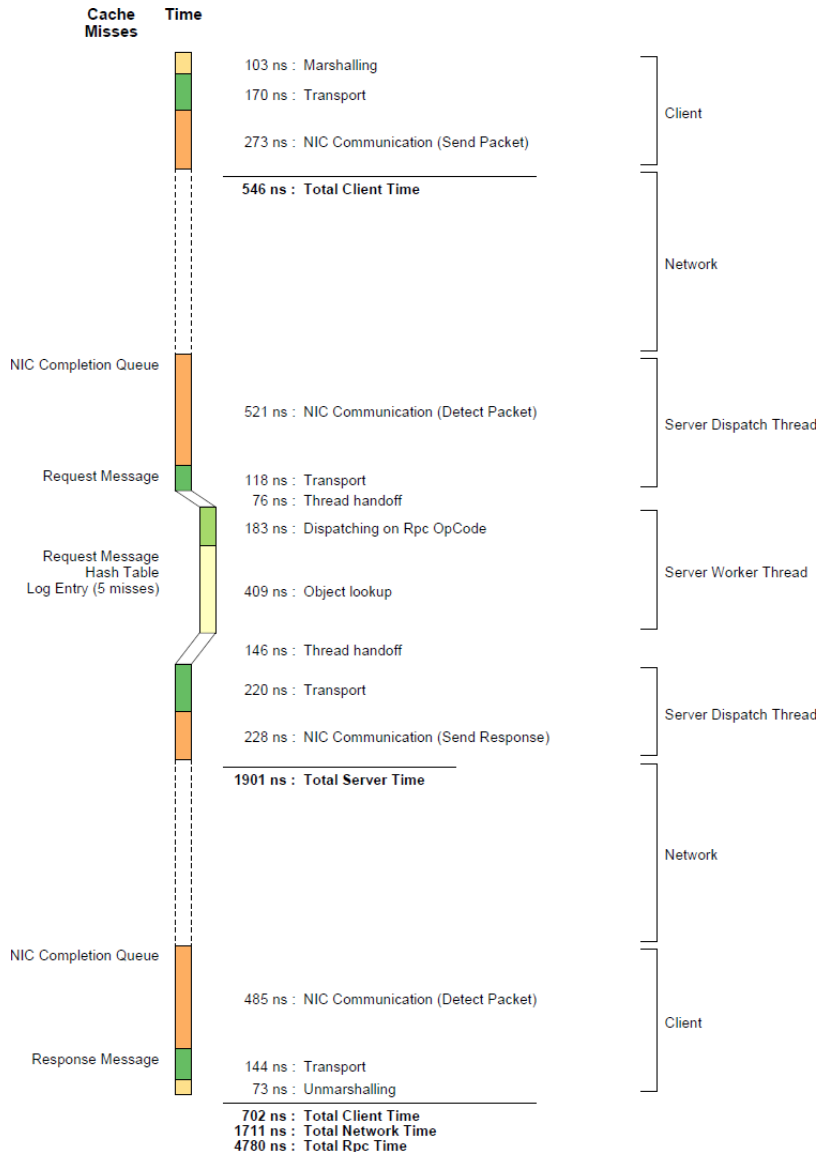
Writes

Median	90%	99%	99.9%
13.4	14.7	75.6	148
18.5	20.8	105	176
35.3	37.7	209	287
228	311	426	489
2200	2300	2400	2700

2.8 Gbytes/sec



Infiniband Read Timeline (100B)

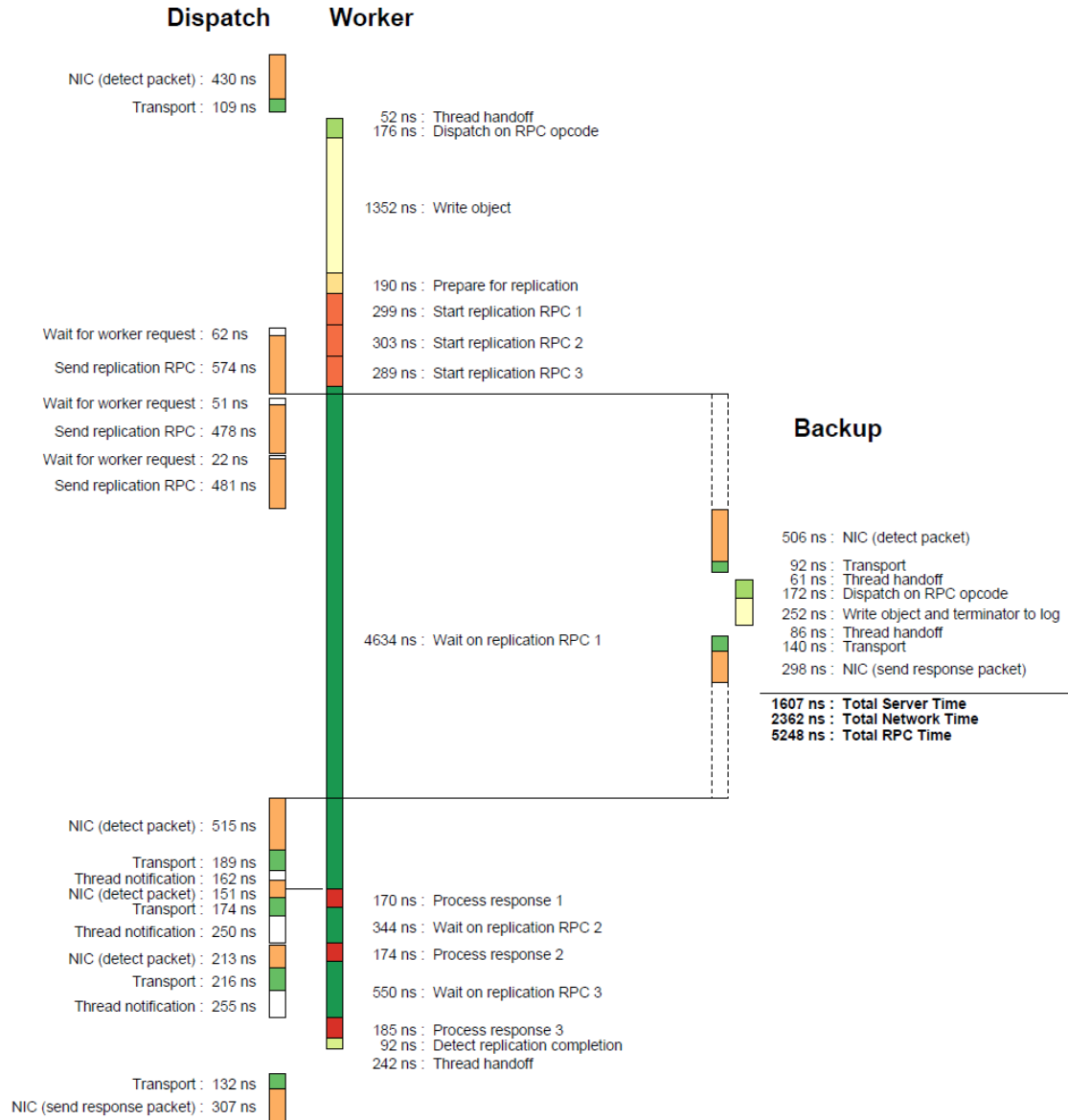


- 3.2 μ s in network and NICs
- 9 L3 cache misses on server (up to 86 ns each)

Time on server:

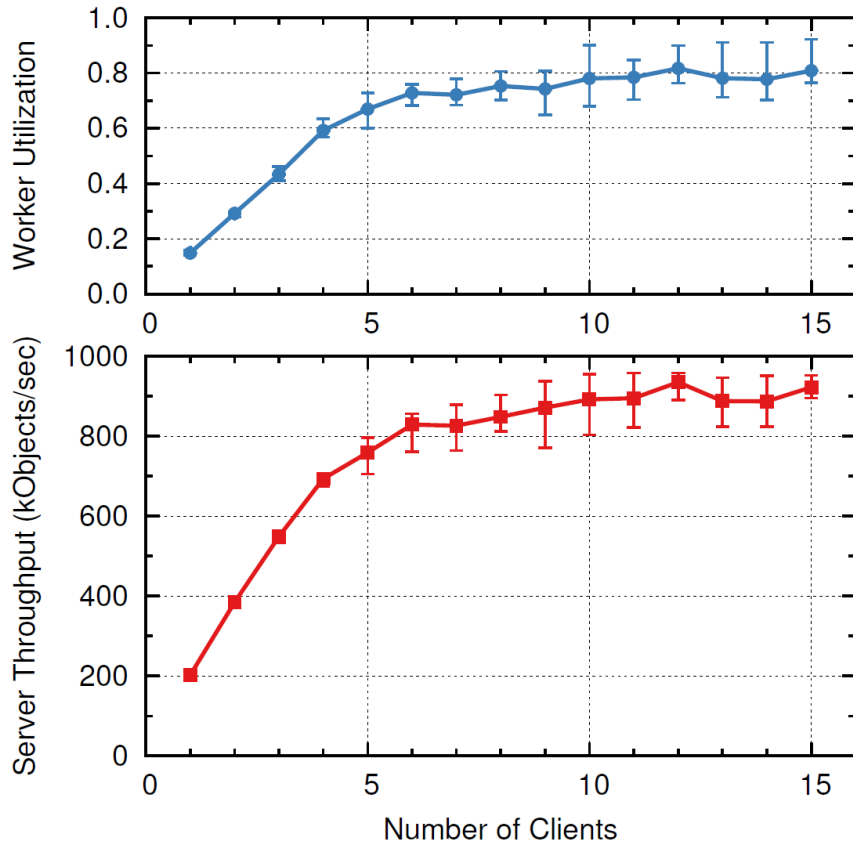
NIC communication:	749 ns	39%
Thread handoffs:	470 ns	25%
Cache misses (est.):	300 ns	16%
Other:	382 ns	20%
Total:	1901 ns	100%

Infiniband Write Timeline (100B)

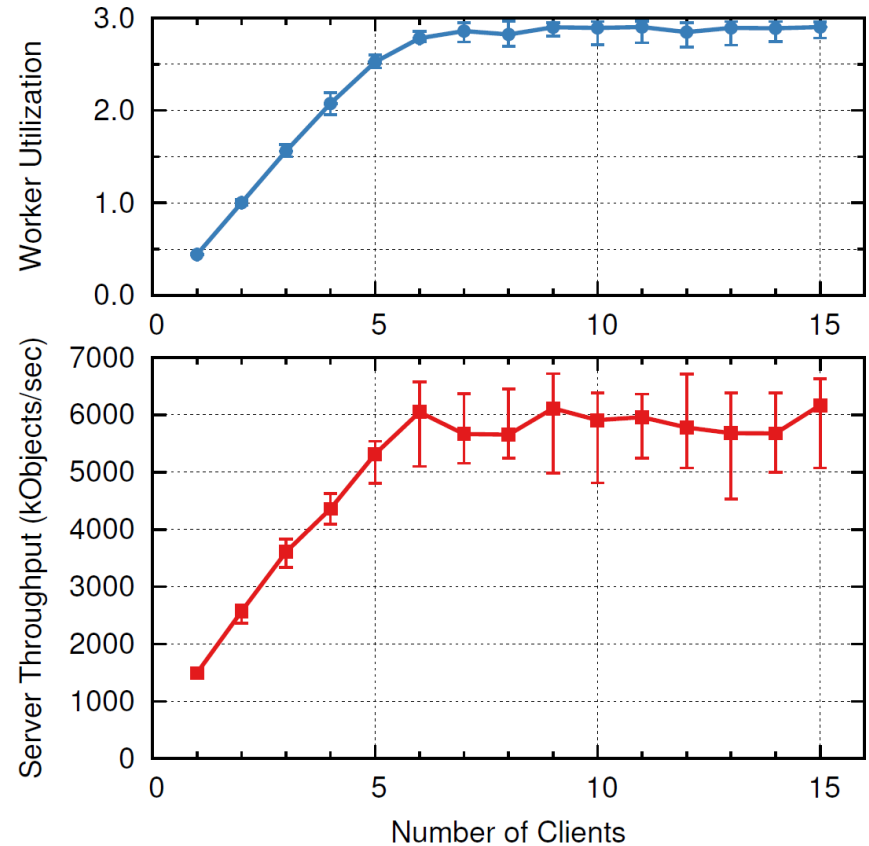


Single-Server Read Throughput

Individual Reads (100B)



Multi-reads (70 × 100B)



Part V: Crash Recovery



Fault Tolerance Introduction

- **Failures to handle:**

- Networking failures (e.g. packet loss, partitions)
- Storage server crashes (masters/backups)
- Coordinator crashes
- Corruption of segments (DRAM and disk/flash)
- Multiple failures
- Zombies: “dead” server keeps operating

- **Assumptions:**

- Fail-stop (no Byzantine failures)
- Secondary storage survives crashes
- Asynchronous network

Fault Tolerance Goals

- **Individual server failures? Continue normal operation:**
 - Near-continuous availability
 - High performance
 - Correct operation
 - No data loss
- **Multiple failures also OK if:**
 - Only a small fraction of servers fail
 - Failures randomly distributed
- **Large-scale outages:**
 - May cause unavailability
 - No data loss (assuming sufficient replication)

Error Handling Philosophy

- **Error handling: huge source of complexity**
 - Must write code 3 times
 - Must handle secondary/simultaneous failures
 - Hard to test
 - Rarely exercised

} May not work
when needed
- **Goal: minimize distinct cases to handle**
- **Technique #1: masking**
 - Deal with errors at a low level
- **Technique #2: failure promotion**
 - E.g., promote all internal server errors to “server failure”

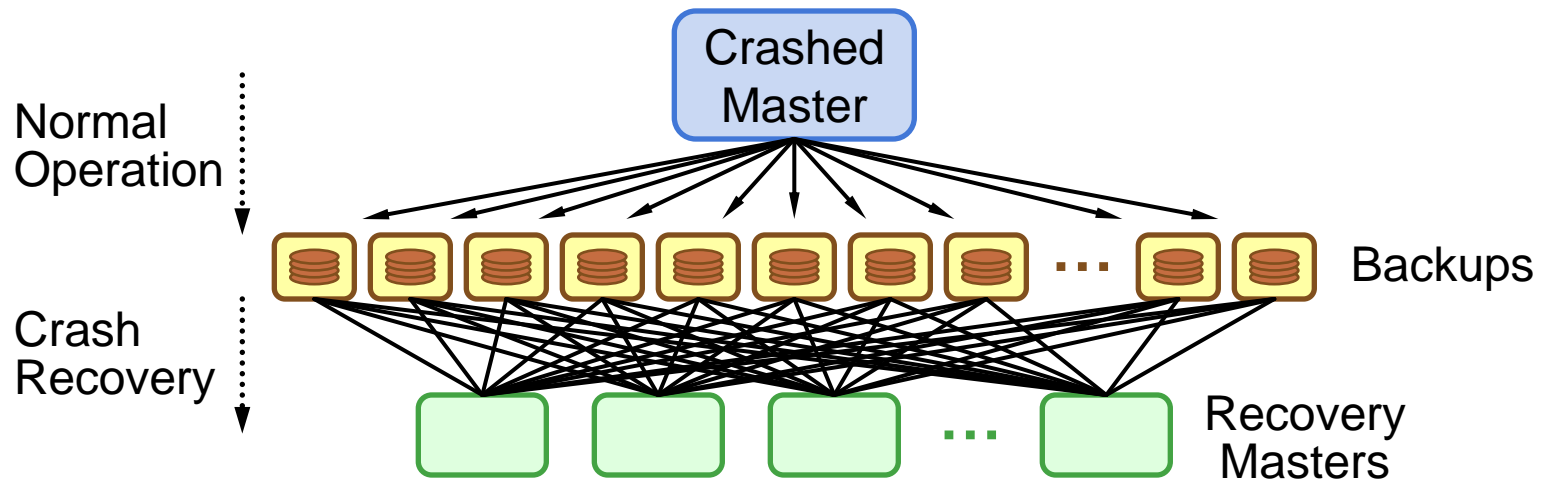
Master Crash Recovery

Additional challenges:

- **Speed: must recover in 1-2 seconds**
 - Data unavailable during recovery
- **Avoid creating scalability bottlenecks**
 - Distributed operations

Fast Master Recovery

- **Goal: recover 256 GB data in 1-2 seconds:**
 - Read from one flash drive? 1000 seconds
 - Transmit over 10 GigE connection? 250 seconds
 - Replay log on one CPU? 500 seconds
- **Solution: concurrency**
(take advantage of cluster scale)

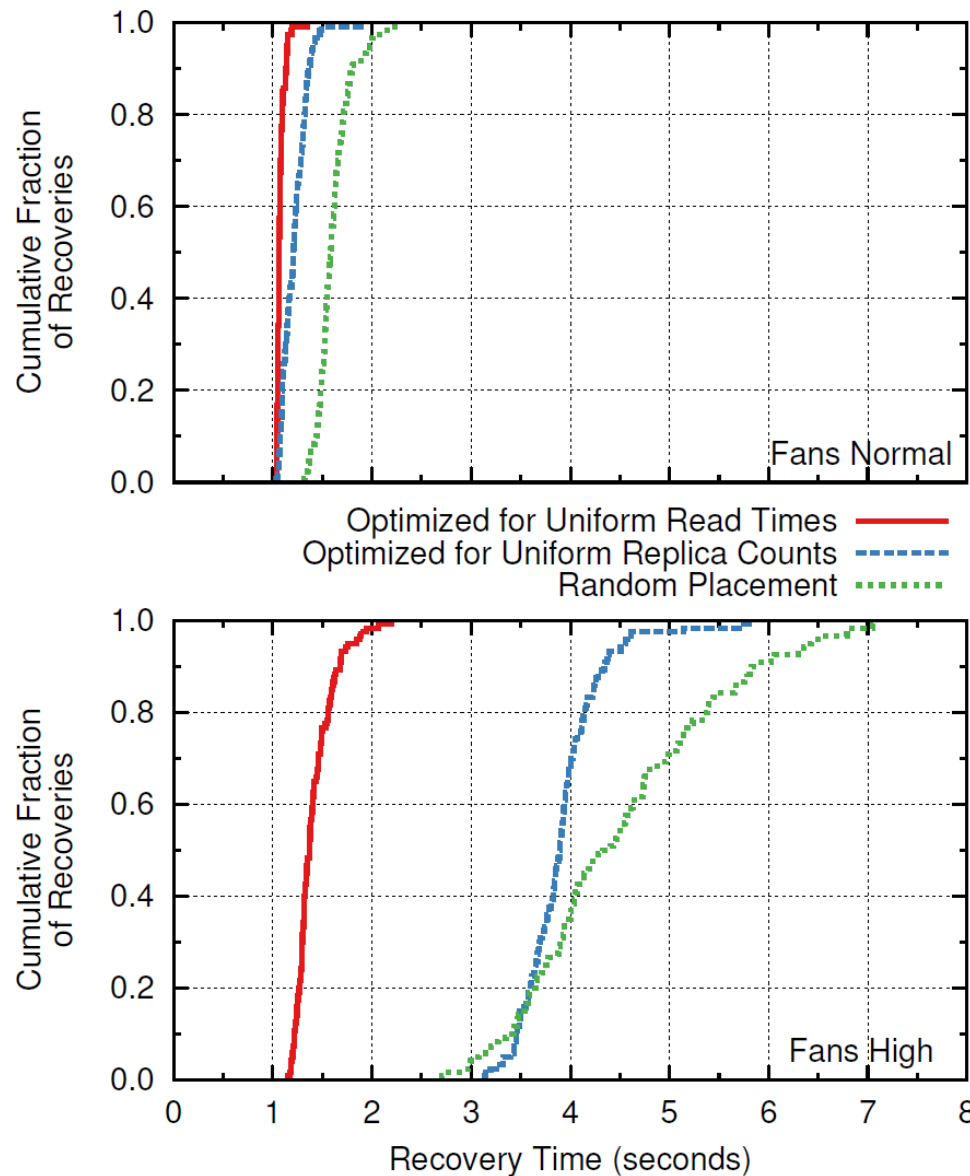


Scattering Replicas

- **Requirements for replica placement:**
 - Distribute replicas for each master uniformly
 - Use backup bandwidth and space evenly
 - Reflect failure modes (replicas in different racks)
 - Backups may have different device capacities/speeds
 - Backups enter and leave cluster
 - Each master must place its replicas independently
- **Solution: randomization with refinement**
 - Mitzenmacher's "power of two choices"
 - Pick several candidate backups at random
 - Select best choice(s)
(minimize worst-case read time for a backup)

Placement Effectiveness

- 120 recoveries per graph
- Replicas stored on disk



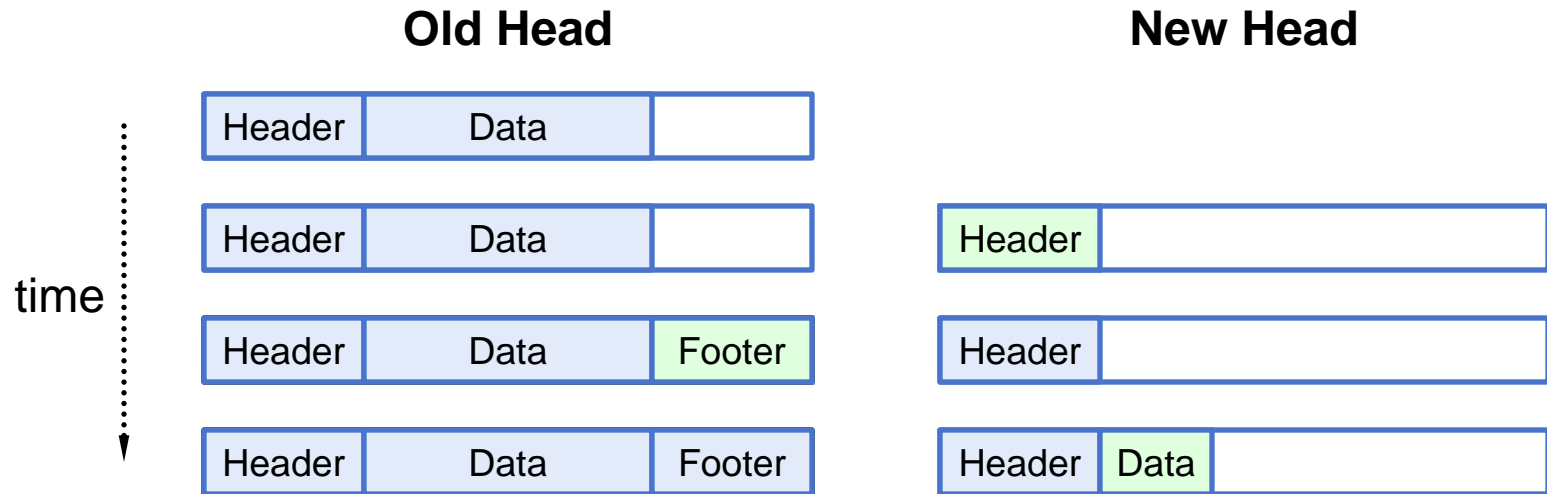
Fast Failure Detection

- **Goal: detect failures in a few hundred ms**
- **Distributed randomized approach:**
 - Every 100ms each server pings another at random
 - No response in 10-20ms? Report to coordinator
 - Coordinator pings again before declaring death
- **Probability of detecting crashed server:**
 - 63% in first round
 - 99% after 5 rounds
- **Problems:**
 - Performance glitches may be treated as failures (overloaded servers)
 - Protocol interactions (200 ms retry interval in TCP)

Master Recovery Overview

- 1. Coordinator collects log metadata from all backups**
- 2. Coordinator divides recovery work (tablet partitions)**
- 3. Coordinator chooses recovery masters, assigns partitions**
- 4. Recovery masters, backups replay log entries**
 - Recovery masters incorporate data into their logs
- 5. Coordinator updates tablet configuration info to make tablets available again**

Ensuring Log Completeness



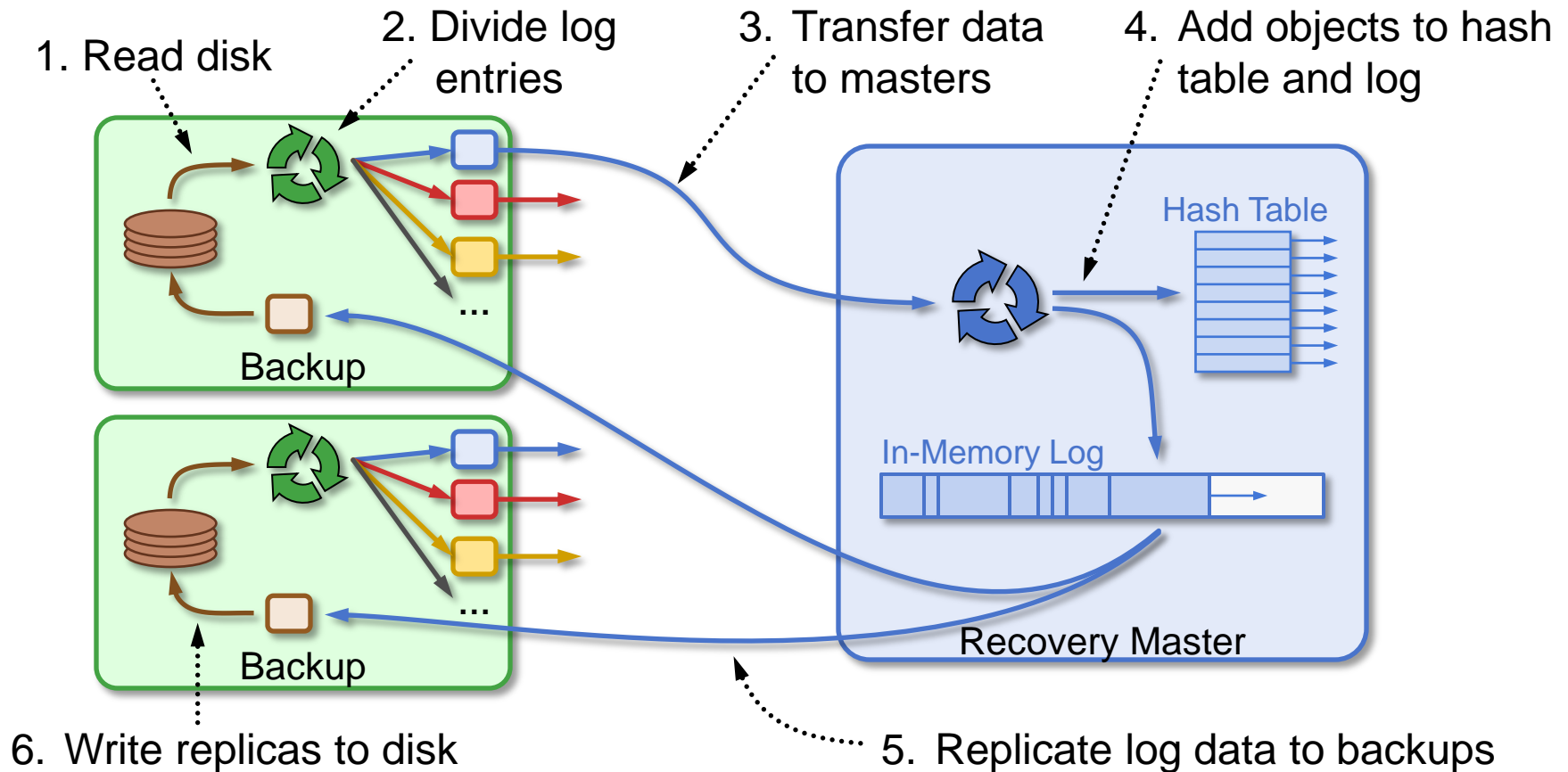
- **Invariants:**

- Header names all other segments in log (**log digest**)
- At least one **open segment** (header but no footer)
- If multiple open segments, only oldest contains data

- **Defer recovery until log complete:**

- Open segment available
- One replica available for each segment in log digest

Log Replay



- **Concurrency in two dimensions:**

- Pipelining
- Data parallelism

Segment Replay Order

- **Backups and masters work independently**
 - Backups read segments, divide log entries
 - Masters fetch partitioned data, replay
- **To avoid pipeline stalls:**
 - Backups publish read order
 - Masters fetch in order of expected availability
 - Masters maintain multiple outstanding fetches
- **Log data replayed out of order:**
 - Version numbers identify most up-to-date information

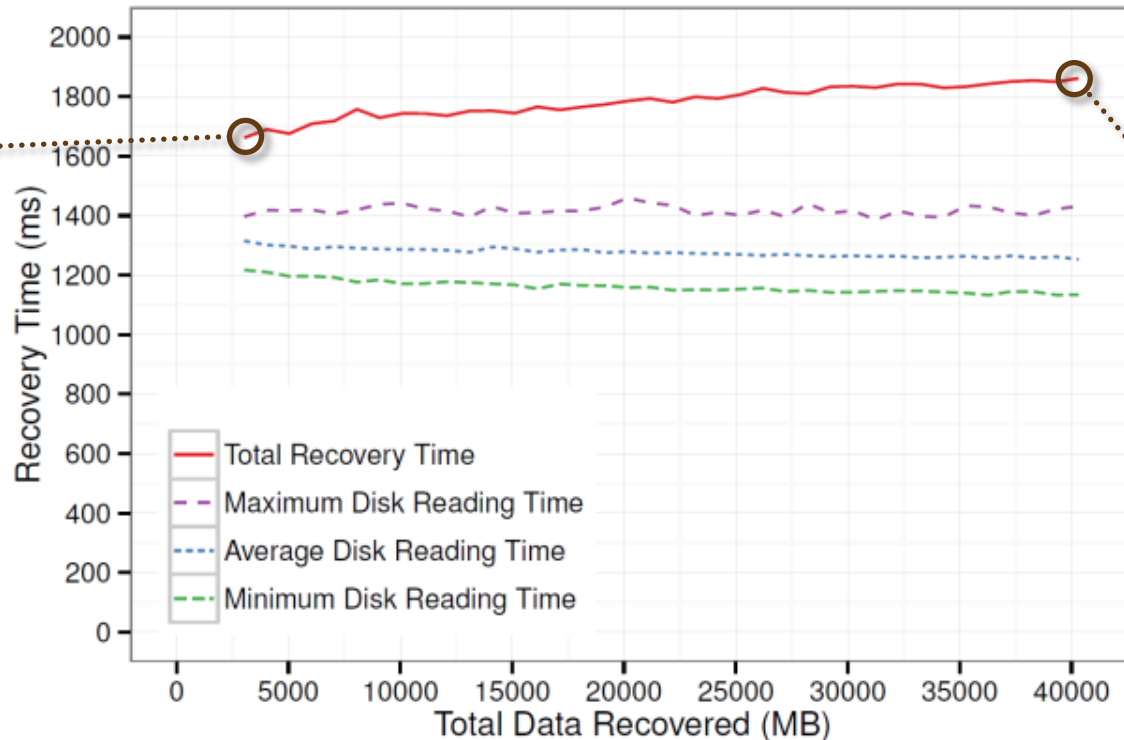
Replay Throughput

Single recovery master (Infiniband):

Object Size (bytes)	Throughput	
	(Mobj/s)	(MB/s)
1	2.32	84
64	2.18	210
128	2.03	319
256	1.71	478
1024	0.81	824
2048	0.39	781
4096	0.19	754

Recovery Scalability

1 master
2 backups
2 SSDs
500 MB



80 masters
160 backups
160 SSDs
40 GB

- **Will improve with newer machines**

- Need more cores (our nodes: 4 cores)
- Need more memory bandwidth (our nodes: 11 GB/sec)

Secondary Failures

Recovery complications:

- **Multiple master failures**
- **Recovery masters:**
 - Crash during recovery
 - Insufficient memory
 - Not enough recovery masters available
- **Backup crashes:**
 - Before recovery
 - During recovery

} Replicas not available
- **Coordinator crashes**

Handling Multiple Failures

- **Recovery is organized incrementally:**
 - Make progress in small independent pieces (one partition for one crashed master)
 - Retry until done
- **Coordinator recovery loop:**
 - Pick a dead master
 - Collect replica info from backups, see if complete log available
 - Choose (some) partitions, assign to recovery masters
 - For recovery masters that complete, update tablet assignments
 - If dead master has no tablets assigned, remove it from cluster
- **This approach also handles cold start, partitions**

Zombies

- **“Dead” servers may not be dead!**
 - Temporary network partition causes ping timeouts
 - RAMCloud recovers “dead” server: tablets reconstructed elsewhere
 - Partition resolved, “dead” server continues to serve requests
 - Some clients use zombie, some use new servers: **inconsistency!**
- **Preventing writes to zombies:**
 - Coordinator must contact backups for head segment during recovery
 - Backups reject replication writes from zombie; zombie suicides
- **Preventing reads from zombies:**
 - Zombie learns of its status during pings for failure detection
 - Only probabilistically safe...

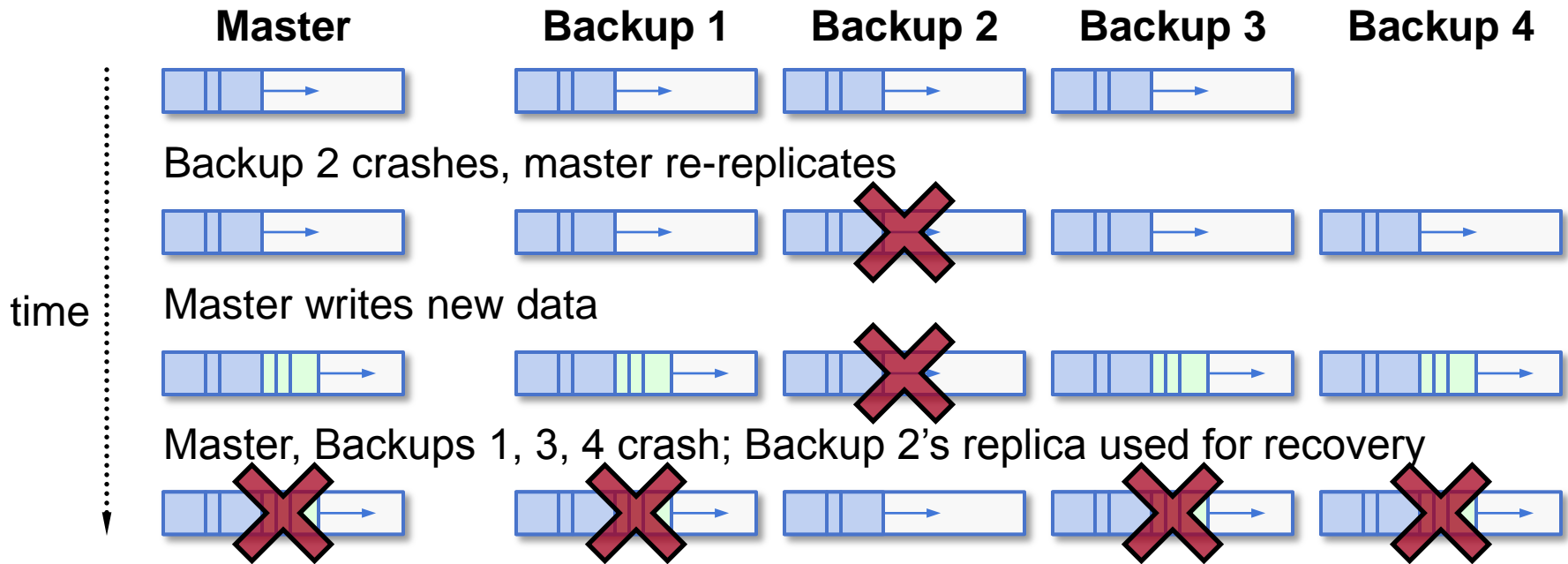
Backup Crashes

- **Basic mechanism:**
 - Coordinator notifies masters of crashes
 - Each master independently re-replicates lost segments
 - Mechanism not time-critical (no loss of availability)
- **Complications:**
 - Backup restart: replica garbage collection
 - Write-all-read-any approach requires replica consistency
 - Replica consistency problems:
 - When backup for head segment crashes
 - When master crashes during re-replication

Replica Garbage Collection

- **Backup restart:**
 - Normal case: can discard existing replicas (all masters have re-replicated)
 - But, sometimes need replicas (e.g. cold start, master crash)
- **For each replica, check state of master**
 - Not in cluster: free replica (master crashed, was recovered)
 - Crashed: retain replica
 - Master up: check with master (“do you still need this replica?”)
 - Repeat until all replicas freed

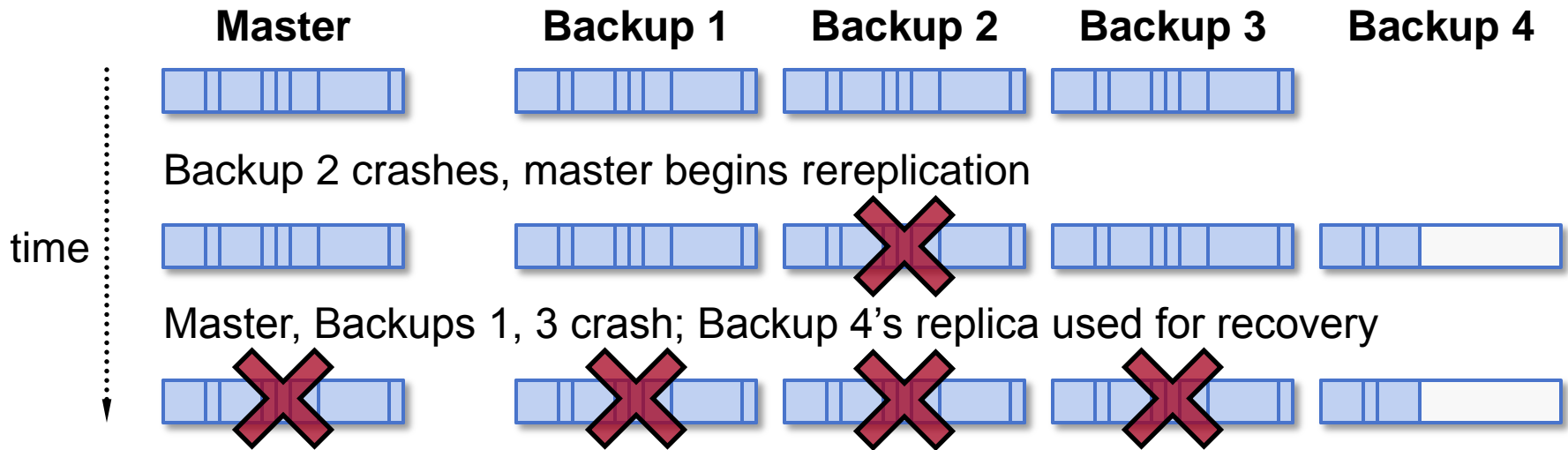
Head Segment Consistency



- **Must prevent use of out-of-date replicas**

- Master sends info to coordinator after crash recovery (new log epoch number)
- Coordinator ignores out-of-date replica during recovery

Crash During Rereplication



- **Must prevent use of incomplete replicas**

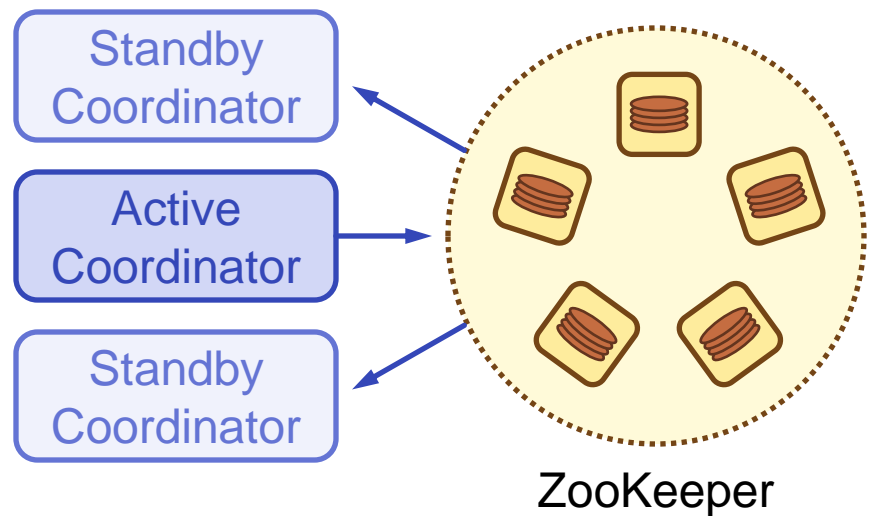
- During rereplication, new replica marked “incomplete”
- Once rereplication complete, new replica marked “complete”
- During recovery, backup doesn't report incomplete replicas

Coordinator Crash Recovery

- **Must protect coordinator metadata:**
 - Server list (active/crashed storage servers)
 - Information for each table:
 - Name
 - Identifier
 - Mapping of tablets to storage servers
- **Store metadata in RAMCloud?**
 - Need server list before recovery
- **Instead, use separate external storage:**
 - Key-value data model
 - Must be highly reliable
 - Doesn't need to be very large or very fast
 - Currently using ZooKeeper

Active/Standby Model

- **One active coordinator:**
 - Record state on external storage
- **Multiple standbys:**
 - Watch activity of active coordinator
 - If active coordinator stops making progress, compete to become new leader
- **New leader:**
 - Read state from external storage
 - Cleanup incomplete operations

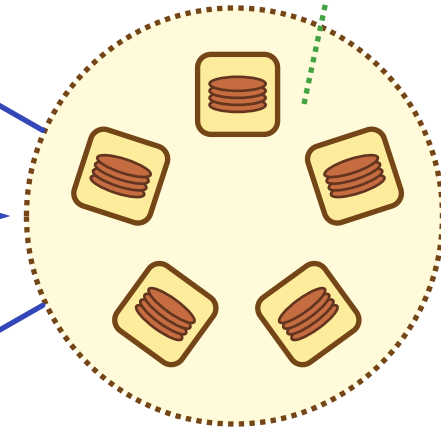
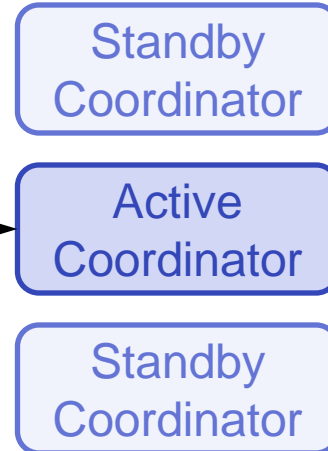


Leader Election & Lease

- Identifies active coordinator
- Version must change within lease time

Leader Object
Service Locator
Version

- Update leader object to maintain leadership (conditional write based on version)
- If update fails, stop acting as coordinator



ZooKeeper

- Check leader object occasionally
- If lease time elapses with no version change, conditional write to become leader

Distributed Updates

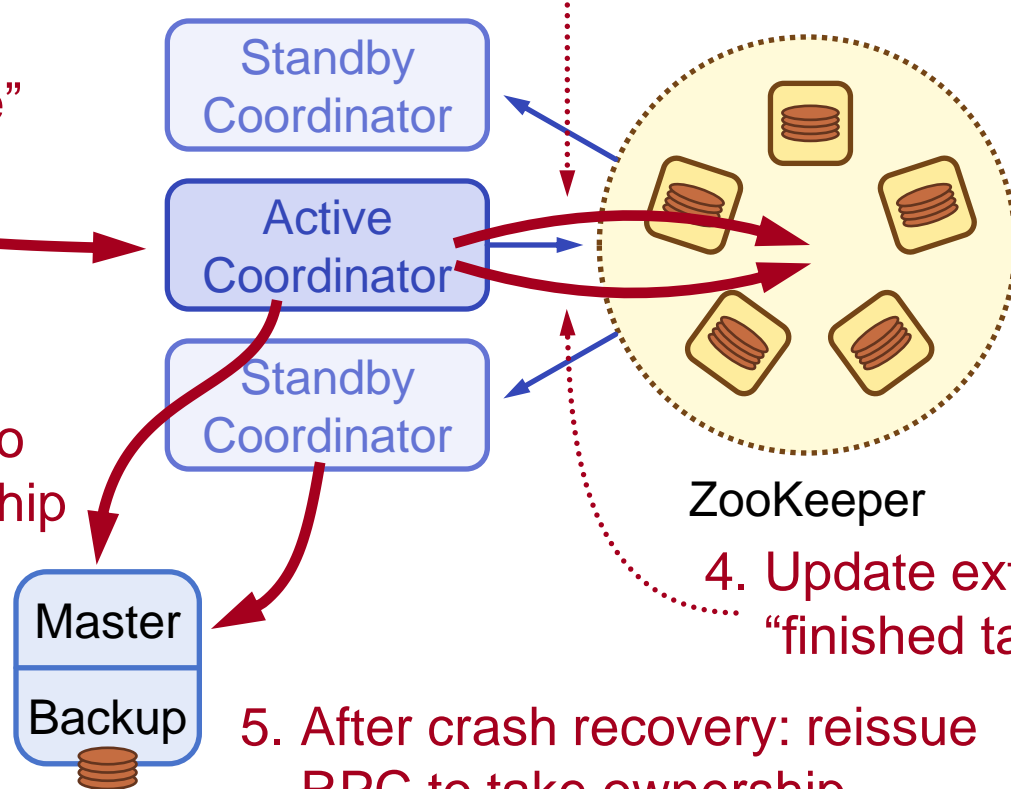
Must maintain consistency between coordinator, other servers, external storage

2. Create external storage object for table: "intend to place on server X"

1. Client request: "create new table"

3. Tell master to take ownership

Must be idempotent!



5. After crash recovery: reissue RPC to take ownership

Part VI: Status and Limitations



RAMCloud History

- **First design discussions: Spring 2009**
- **Began serious coding: Spring 2010**
- **Version 1.0 in January 2014**
 - Includes all features described here
 - Usable for applications
- **Available in open-source form at RAMCloud Wiki:**
<https://ramcloud.stanford.edu/>
- **Goal: ~~research prototype~~ production-quality system**

Limitations

- **No geo-replication**
- **Key-value data model**
- **Linearizability support incomplete**
- **No protection**
- **Incomplete configuration management (mechanisms but no policies)**

Current Work

- **Higher-level data model:**
 - Secondary indexes
 - Multi-object transactions
 - Full linearizability
 - Research question: achievable at low latency and large scale??
- **New transport layer:**
 - New protocol for low-latency datacenter RPC (replace TCP)
 - New threading architecture
 - Better scalability

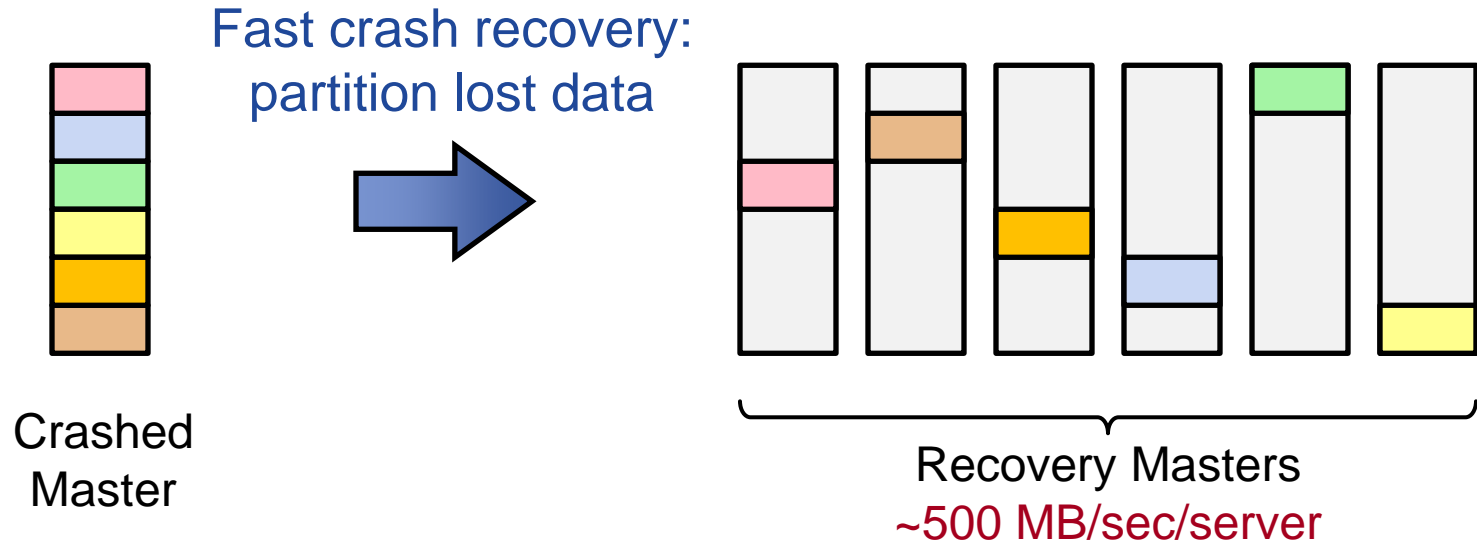
Part VII: Application Experience



Applications?

- **No applications in production, but several experiments:**
 - Stanford: natural language processing, graph algorithms
 - Open Networking Laboratory: ONOS (operating system for software defined networks)
 - CERN: high energy physics (visiting scientist, summer 2014)
 - Huawei: real-time device management
- **Challenges**
 - Low-latency networking not yet commonplace
 - RAMCloud not cost-effective at small scale
 - RAMCloud is too slow (!!)

Scale and Recovery

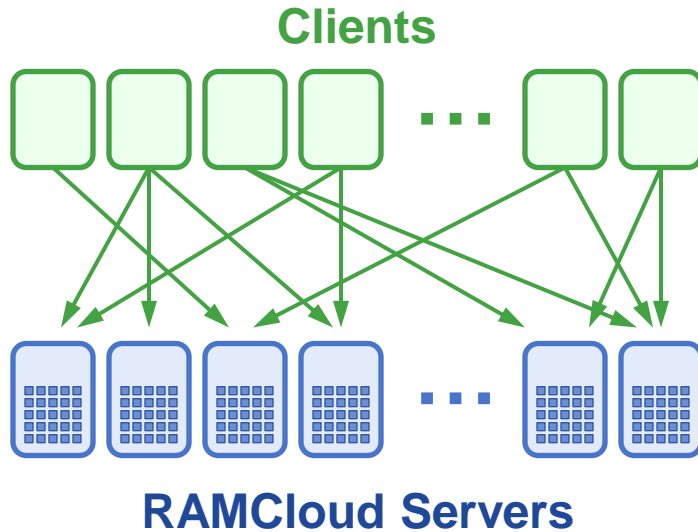


Cluster Size	Server Capacity	Cluster Capacity	Recovery Time
101 servers	50 GB	5 TB	1 sec
201 servers	100 GB	20 TB	1 sec
6 servers	100 GB	600 GB	40 sec
6 servers	2.5 GB	15 GB	1 sec
11 servers	5 GB	55 GB	1 sec

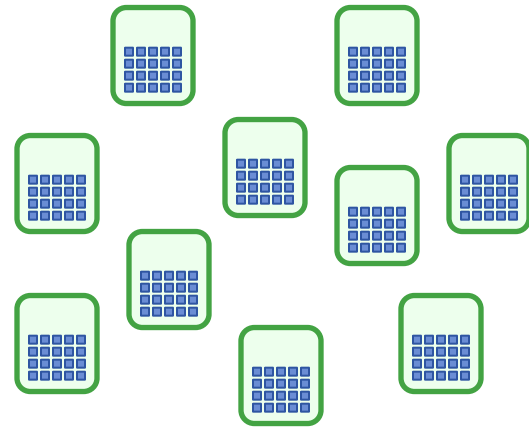
Small clusters can't have both fast recovery and large capacity/server

Fast But Not Fastest

Choice #1:
5-10 μ s remote access



Choice #2:
50-100ns local access



- **Choice #2 is 100x faster than RAMCloud**
 - And, can store data in application-specific fashion
 - But, data must partition
 - What about persistence?

Application Philosophy

- **Technology transfer is a numbers game:**
 - Must try many experiments to find the right fit
- **Our goals:**
 - Learn something from every test case
 - Keep improving RAMCloud
- **Application issues suggest new research opportunities**

Part VIII: Lessons Learned



Logging

- **Initially chosen for performance (batch writes to disk/flash)**
- **Many other advantages:**
 - Crash recovery: self-identifying records that can be replayed
 - Convenient place for additional metadata (log digest, tablet usage stats)
 - Consistent replication: mark consistent points
 - Immutable: simplifies concurrent access
 - Neutralize zombies (disable head segment)
 - **Manages memory quite efficiently**
- **Disadvantage:**
 - Only one insertion point per master: limits throughput

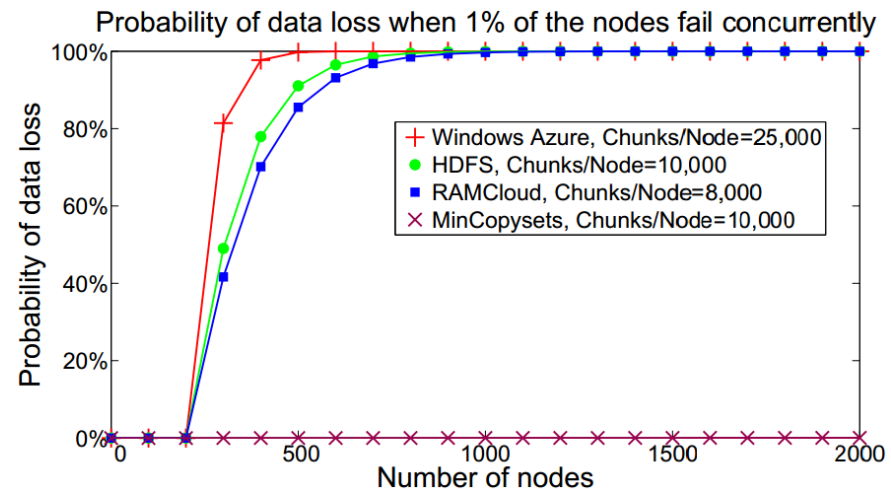
Randomization

Essential tool for large-scale systems:

- **Replace centralized decisions with distributed ones:**
 - Choosing backups for replicas
 - Failure detection
- **Simple and efficient algorithms for managing large numbers of objects**
 - Coordinator dividing tablets among partitions during recovery
- **Many “pretty good” decisions produces nearly optimal result**

Sometimes Randomization is Bad!

- **Select 3 backups for segment at random?**
- **Problem:**
 - In large-scale system, any 3 machine failures results in data loss
 - After power outage, ~1% of servers don't restart
 - Every power outage loses a few segments!
- **Solution: derandomize backup selection**
 - Pick first backup at random (for load balancing)
 - Other backups deterministic (**replication groups**)
 - Result: data safe for hundreds of years
 - (but, lose more data in each loss)



Ubiquitous Retry

Assume operations may not succeed at first: provide mechanism for retries

- **Fault tolerance:**

- After crash, reconstruct data and retry
- Incomplete recovery

- **Configuration changes (e.g., tablet moved)**

- **Blocking:**

- Don't block operations on servers (resource exhaustion, deadlock)
- Return STATUS_RETRY error; client retries later

- **Retries now built into RPC system**

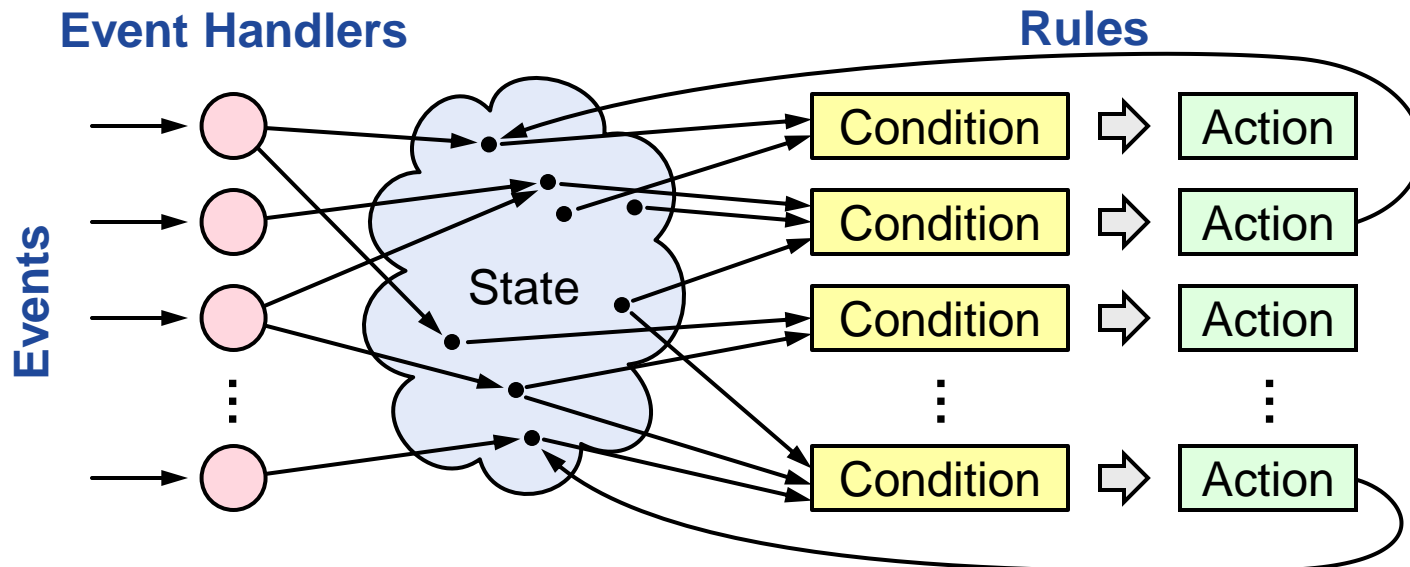
- All RPCs transparently retry-able
- Can define reusable **retry modules** (e.g. for “tablet moved”)

Rules-Based Programming

- **RAMCloud contains many DCFT modules (Distributed, Concurrent, Fault-Tolerant)**
 - Segment replica manager
 - Cluster membership notifier
 - Main loop of recovery masters
 - Multi-read dispatcher
 - ...
- **Very hard to implement! (nondeterminism)**

Rules-Based Programming, cont'd

- **Solution: decompose code into rules**
 - Rule = condition to check against state, action to execute
 - Each rule makes incremental progress towards a goal
 - DCFT module = retry loop
 - Execute rules until goal reached



Layering Conflicts With Latency

- **Layering:**
 - Essential for decomposing large systems
 - Each crossing adds delay
 - Many layers → high latency
 - Granular interfaces especially problematic
- **For low latency, must rethink system architecture**
 - Minimize layer crossings
 - **Thick interfaces:** lots of useful work for each crossing
 - **Fast paths** that bypass layers (e.g., kernel bypass for NICs)

Conclusion

- **RAMCloud: general-purpose DRAM-based storage**
 - Scale
 - Latency
- **Goals:**
 - Harness full performance potential of DRAM-based storage
 - Enable new applications: intensive manipulation of large-scale data
- **What could you do with:**
 - 1M cores
 - 1 petabyte data
 - 5-10 μ s flat access time

References

- [1] RAMCloud Wiki: <https://ramcloud.atlassian.net/wiki/display/RAM/RAMCloud>
- [2] J. Ousterhout et al., “The RAMCloud Storage System,” under submission, <https://ramcloud.atlassian.net/wiki/display/RAM/RAMCloud?preview=/6848571/6947168/RAMCloudPaper.pdf>
- [3] D. Ongaro, S. Rumble, R. Stutsman, J. Ousterhout and M. Rosenblum, “Fast Crash Recovery in RAMCloud,” *Proc. 23rd ACM Symposium on Operating Systems Principles*, October 2011, pp. 29-41.
- [4] S. Rumble, A. Kejriwal, and J. Ousterhout, “Log-Structured Memory for DRAM-based Storage,” *12th USENIX Conference on File and Storage Technology (FAST '14)*, February 2014, pp. 1-16.
- [5] Ryan Stutsman’s Ph.D. dissertation: *Durability and Crash Recovery in Distributed In-memory Storage Systems, 2013*
- [6] Steve Rumble’s Ph.D. dissertation: *Memory and Object Management in RAMCloud, 2014*
- [7] R. Stutsman, C. Lee, and J. Ousterhout, “Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code,” Stanford technical report, <https://ramcloud.atlassian.net/wiki/display/RAM/RAMCloud+Papers?preview=/6848671/12058674/dcft.pdf>

Palette

