The Dissertation Committee for Yang Wang
certifies that this is the approved version of the following dissertation:

Separating Data from Metadata for Robustness and Scalability

Committee:

Lorenzo Alvisi, Supervisor

Michael Dahlin, Co-Supervisor

Keshav Pingali

Michael Walfish

Remzi Arpaci-Dusseau

Emmett Witchel

Separating Data from Metadata for Robustness and Scalability

by

Yang Wang, B.E.; M.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2014

Acknowledgments

The six years of Ph.D. life in Austin have been one of the most enjoyable experiences in my life, not only because I have done work that I believe is interesting, but also because I had the opportunity to work with and learn from a group of smart, supportive, and kind people.

My advisors, Lorenzo Alvisi and Mike Dahlin, have guided me with their insights, patience, and encouragement during the six years. What I have learnt from them is far beyond the research work I have done and such experience finally motivates me to follow their paths.

The other members of my committee (Remzi Arpaci-Dusseau, Keshav Pingali, Michael Walfish, and Emmett Witchel) not only helped improve this document, but also provided insightful suggestions for my work and great supports for my career decision. I also want to express my special thanks to Calvin Lin, who made my defense possible by taking this time-consuming job as an observer.

My work would not exist without the help from other fellow graduate students in the LASR group: Allen Clement, Lakshmi Ganesh, Manos Kapritsos, Jeevitha Kirubanandam, Sangmin Lee, Prince Mahajan, Taylor Riche, Lara Schmidt, Mark Silberstein, Chunzhi Su, Chao Xie, and Navid Yaghmazadeh. Other students have also provided great advice during my work: Sebastian Angel, Trinabh Gupta, Owen Hofmann, Sangman Kim, Joshua Leners, Don Porter, Srinath Setty, and Ed Wong. Thank you all for bringing passion and joy to my Ph.D. life.

I would like to thank the staff of TACC and Emulab. It would have been impossible to finish my work without their help.

Finally, I want to thank my parents and my wife Yi Huang for their patience and understanding: six years is short for people who enjoyed it, like me; but six years is long for people who were waiting, like my parents and Yi. I would not be the person that I am today without them.

Yang Wang

The University of Texas at Austin
December 2014

Separating Data from Metadata for Robustness and Scalability

Publication No. _____

Yang Wang, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Lorenzo Alvisi
Co-Supervisor: Michael Dahlin

When building storage systems that aim to simultaneously provide robustness, scalability, and efficiency, one faces a fundamental tension, as higher robustness typically incurs higher costs and thus hurts both efficiency and scalability. My research shows that an approach to storage system design based on a simple principle—separating data from metadata—can yield systems that address elegantly and effectively that tension in a variety of settings. One observation motivates our approach: much of the cost paid by many strong protection techniques is incurred to detect errors. This observation suggests an opportunity: if we can build a low-

cost oracle to detect errors and identify correct data, it may be possible to reduce the cost of protection without weakening its guarantees. This dissertation shows that metadata, if carefully designed, can serve as such an oracle and help a storage system protect its data with minimal cost.

This dissertation shows how to effectively apply this idea in three very different systems: Gnothi—a storage replication protocol that combines the high availability of asynchronous replication and the low cost of synchronous replication for a small-scale block storage; Salus—a large-scale block storage with unprecedented guarantees in terms of consistency, availability, and durability in the face of a wide range of server failures; and Exalt—a tool to emulate a large storage system with 100 times fewer machines.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The primary directive of storage—not to lose data—is hard to carry out: disks can fail in unpredictable ways [13, 14, 42, 55, 86, 92], and so can CPUs and memory [81, 93]. Concerns about robustness become even more pressing as scalable storage systems like Google's GFS [44], Bigtable [27], Megastore [15], and Spanner [32], Facebook's Haystack [17], and Amazon's DynamoDB [36] become more complex. For example, Google observes one corruption for every 5.4 petabytes of data scanned in Bigtable [35]. What is worse, the consequences of such corruptions are hard to predict: in the infamous 2008 Amazon outage, a single bit flip caused the entire Amazon S3 service to be down for 8 hours [1].

Strong protection techniques, such as Byzantine Fault Tolerance (BFT) [26, 31], can shield the system from unexpected and uncommon errors, but are usually expensive: the scale of today's large storage systems magnifies the cost of these strong protection techniques, severely reducing their applicability in practice. Therefore, developers today are facing a painful tradeoff between robustness and scalability and, in practice, they usually choose affordable solutions that can tolerate several types of common errors, leaving the system vulnerable to uncommon errors with large impact. To resolve the tension between robustness and scalability, my re-

search explores new ways to provide modern storage systems with extremely high levels of reliability at reasonable cost.

My approach to building robust and scalable storage systems utilizes an old but powerful idea—*separating data from metadata*—in new ways. Many previous systems protect metadata more aggressively than data because metadata in storage systems is usually smaller and more important than data [2,5,23,88]. Consequently, these systems usually offer stronger guarantees for metadata than for data. The dissertation, however, shows that many distributed storage systems can actually achieve strong guarantees for *both* metadata *and* data by applying strong protection to metadata and minimal protection to data. This perhaps surprising result is based on the observation that much of the cost paid by many strong protection techniques is incurred to *detect* errors. Taking simple data replication as an example, if the system only aims at tolerating machine crashes, then two replicas are enough to cover one failure because the system can detect whether a machine has crashed or not by using complementary mechanisms, such as timeouts. If the aim is to tolerate arbitrary errors, however, then there is no obvious way to detect whether a machine is faulty, and the system needs at least three replicas to outvote a single faulty replica.

This observation suggests an opportunity: since the high cost of strong protection usually comes from error detection, if we can build a low-cost oracle to detect errors and identify correct data, it may be possible to reduce the cost of protection without weakening its guarantees. This dissertation shows that metadata, if carefully designed, can serve as such an oracle. Our approach involves three steps: first, we design metadata so that it can be used to validate data integrity; we then apply those strong and expensive techniques *only* to metadata, with little effect on scalability; finally, we use such strongly protected metadata to identify correct data. Despite its simplicity, we show that this approach yields data protec-

tion with strong guarantees at minimal cost: in particular, we are able to employ powerful fault tolerance techniques such as Paxos [61,62] and end-to-end Byzantine Fault Tolerance (BFT) [26,31] at little additional cost over weaker alternatives such as, respectively, synchronous primary backup and piecemeal checksums. In fact, in some cases, providing strong end-to-end guarantees opens up new optimization opportunities that allow our hardened systems to significantly outperform the original systems on which they are based.

I have applied this approach to build three different systems: Gnothi, a small-scale storage system that can tolerate data loss and timing errors cheaply; Salus, a large-scale block store that provides strong end-to-end guarantees for read operations, strict ordering guarantees for write operations, and strong durability and availability guarantees despite a wide range of server failures (including memory corruptions, disk corruptions, firmware bugs, etc); and Exalt, an emulator that allows researchers to test the scalability of today's large storage systems.

- *Gnothi: Efficient and available storage replication [Chapter 2].* Replication is the key technique to guarantee data durability and availability in storage systems and multiple replication protocols have been proposed to provide different guarantees with different costs: synchronous primary backup uses $f+1$ replicas to tolerate f crash failures but it usually employes a conservative timeout to perform accurate failure detection, which hurts the availability of the system; asynchronous replication (e.g. Paxos) does not rely on accurate failure detection, but it increases the replication cost to $2f+1$. My work targets the following question: can one write data to only $f+1$ nodes and still use a short and potentially inaccurate timeout without risking correctness? This is well-known to be impossible in the general case, but in storage systems, leveraging the key idea of separating data from metadata allows me to closely approximate this goal: by replicating metadata with Paxos and using

metadata to identify correct data during failure and recovery, I show that it is sufficient to replicate data on only $f + 1$ nodes. I have built a small-scale storage system, Gnothi, based on this insight.

- *Salus: A robust and scalable block store [Chapter 3].* Salus provides functionalities similar to those of Amazon's popular Elastic Block Store (EBS), but with unprecedented guarantees in terms of consistency, availability, and durability in the face of a wide range of server failures (including memory corruptions, disk corruptions, CPU errors, etc.).

  Existing scalable storage systems usually give up certain robustness properties for scalability, but Salus demonstrates that such trade-offs may not be necessary. For example, scalable systems shard data and write data to different shards in parallel to achieve scalability. This approach, however, does not provide ordering guarantees between writes, and such guarantees are essential to the correctness of certain applications, e.g. a block store. Salus addresses this problem by separating data transfer from metadata transfer: data is processed in parallel, while metadata, which carries information about which data can be committed, is processed sequentially. If failures occur, Salus utilizes metadata to identify data that can be committed. Salus addresses a second key challenge: large-scale storage systems are usually composed of multiple layers, with data replication performed at the lowest layer. In such systems, using approaches similar to Gnothi to enhance the robustness of the replication layer is not enough, since middle layers are not replicated and can become single points of failure. Salus shows that replicating such middle layers can improve not only the robustness of the system, but also its efficiency when disk bandwidth exceeds network bandwidth.

- *Exalt: An emulator for evaluating large-scale storage systems on small-to-medium infrastructures [Chapter 4].* A basic tenet of sound systems research

is to validate a design by implementing a prototype and running experiments on it. Abiding by this precept when designing highly scalable storage systems, however, is prohibitively hard: for example, Salus targets systems with thousands of machines and tens of thousands of disks, but the largest affordable experimental infrastructure I could use to validate my design included only 200 machines. The lack of large testbeds presents a fundamental challenge to almost all researchers working on large-scale systems: even industrial researchers who are within reach of clusters of the necessary size may not be able to reserve them for large-scale experiments, since these clusters are a primary source of revenue.

To solve this problem, I have designed an emulator, Exalt, that uses data compression to reduce by two orders of magnitude the number of physical machines needed to validate a storage system of a given size. To achieve efficient compression, I leverage the observation that the behavior of storage systems often does not depend on the actual data being stored: this insight is at the core of Tardis, a new synthetic data format that allows applications to quickly separate data from metadata and achieve high rates of data compression.

By applying Exalt to existing large-scale storage systems, I improve the scalability of a mature storage system by an order of magnitude compared to its default configuration and unearth several performance issues that are not observable at small scale.

Gnothi [104], Salus [106], and Exalt [105] have each been the subject of conference publications: this dissertation not only expands on the original papers, but also improves Salus and Exalt in both design and evaluation.

Chapter 2

Gnothi: Robust and Efficient Storage Replication

Replication, one of the core techniques to provide fault tolerance in storage systems, is sensitive to the tension between robustness and efficiency: 1) synchronous primary-backup systems [27, 34, 44] require $f + 1$ replicas to tolerate f crash faults, but they risk data loss if there are timing errors; 2) asynchronous full replication systems [15, 19, 22, 54] use asynchronous agreement [61, 62] to ensure correctness despite timing errors, but send data to $2f + 1$ replicas to tolerate f crash failures and thus have higher costs than synchronous primary-backup systems; 3) asynchronous partial replication systems [63, 108] still require $2f + 1$ replicas, but they only activate $f + 1$ of the replicas in the failure-free case; the spare replicas are activated only if some of the active ones fail. Although existing partial replication approaches work well for small-state services, they are not well-suited for replicating a storage system because, after a failure, the system becomes unavailable until it activates a spare replica, which requires copying all of the state from available replicas. If the copying can be done at, say, 100MB/s, then the fail-over time would exceed 2.7 hours per terabyte of storage capacity.

This dissertation presents Gnothi,[1] a new block storage system that simultaneously achieves robustness (correctness despite timing errors and availability despite failures) and efficiency ($f + 1$ data replication). Gnothi replicates data to guarantee availability and durability when replicas fail. To guarantee correctness despite timing errors, Gnothi uses $2f + 1$ replicas to perform asynchronous state machine replication [61, 62, 91]. To reduce network bandwidth, disk arm overhead, and storage cost, Gnothi executes updates to different blocks on different subsets of replicas. The key challenge is to perform partial replication while not hurting availability or durability. Gnothi meets this challenge by using two key ideas.

First, to ensure availability during failure and recovery, Gnothi *separates data from metadata* so that metadata is replicated on all replicas while data for a given block is replicated only to a preferred subset for that block. A replica's metadata keeps the status of each block in the system, including whether the replica holds the block's current version. Replicating metadata to all replicas allows a replica to always process a request correctly, even while it is recovering after having missed some updates.

Second, to ensure durability during failures, Gnothi *reserves a small fraction (e.g. 10%) of storage on each replica* to buffer writes to unavailable replicas. While up to f of a block's *preferred replicas* are unresponsive, Gnothi buffers writes in the reserve storage of up to f of the block's available *reserve replicas*. Directing writes to a reserved replica when a block's preferred replica is unavailable guarantees that each new update is always written to $f+1$ replicas even if some replicas fail. Gnothi allows a tradeoff between availability and space cost: data is writeable in the face of f failures as long as failed nodes are repaired before the reserve space is exhausted. To guarantee write availability regardless of failure duration or repair time, conservative users can configure the system with the same space as asynchronous full replication

[1] "Gnothi S'auton" (Γνῶθι σ'αυτόν) is the ancient Greek aphorism "Know thyself".

($2f+1$ actual storage blocks per logical block). Given that in Gnothi replicas recover quickly, analysis of several traces shows that a 10% reserve is enough to guarantee write availability for many workloads.

Gnothi combines these ideas to ensure availability and durability during failures and to make recovery fast despite partial replication. In summary, Gnothi provides the following guarantees: when an update completes, data is stored on $f + 1$ disks; all reads and writes are linearizable [52]; reads always return the most current data even though some replicas may have stale versions of some blocks; the system is available for reads as long as there are at most f failures; and the system is available for writes as long as there are at most f failures and failed replicas recover or are replaced before the reserve buffer is fully consumed by new updates.

We implement Gnothi by modifying the ZooKeeper server [54]. Gnothi provides a block store API, and it can be used like a disk: users can mount it as a block device and create and use a filesystem on it. We evaluate Gnothi's performance both in the common case and during failure recovery and compare it with Gaios, a state-of-the-art Paxos-based block replication system [19]. The evaluation shows that Gnothi's write throughput can be 40%-64% higher than our implementation of a Gaios-like system while retaining Gaios's excellent read scalability. We also find that for systems with large amounts of state, separating data and metadata significantly improves recovery compared to traditional state machine replication. Unlike standard Paxos-based systems, Gnothi ensures that a recovering replica will eventually catch up regardless of the rate at which new requests are processed, and unlike previous partial replicated systems, Gnothi remains available even while large amounts of state are rebuilt on recovering replicas.

2.1 Design

2.1.1 Interface and Model

Gnothi targets disk storage systems within small clusters of tens of machines. Because linearizability is composable, it is possible to scale Gnothi by composing multiple small clusters: this is discussed in Chapter 3.

Gnothi provides an interface similar to a disk drive: there is a fixed number of blocks with the same size, and applications can read or write a whole block. Block size is configurable. Our experiments use sizes ranging from 4KB to 1MB, but smaller or larger sizes are possible.

Gnothi provides linearizable reads and writes across different clients. Furthermore, if a client has multiple outstanding requests, Gnothi can be configured so that they will be executed in the order they were issued.

Gnothi is designed to be safe under the asynchronous model: the network can drop, reorder, modify, or arbitrarily delay messages. Therefore, Gnothi makes no assumption about the maximum communication delay between nodes, and thus it is impossible to detect whether a node has failed or it is just slow. Gnothi provides the same guarantees as previous asynchronous replicated state machines (RSMs): the system is always safe (all correct replicas process the same sequence of updates), but it is only live (the system guarantees progress) during periods when the network is available and message delivery is timely. Gnothi uses $2f + 1$ replicas to tolerate f omission/crash failures. Commission/Byzantine failures are not considered.

2.1.2 Architecture

As shown in Figure 2.1, Gnothi uses the Replicated State Machine (RSM) approach [91]: agreement modules on different replicas work together to guarantee that all replicas process the same client update requests in the same order. Re-

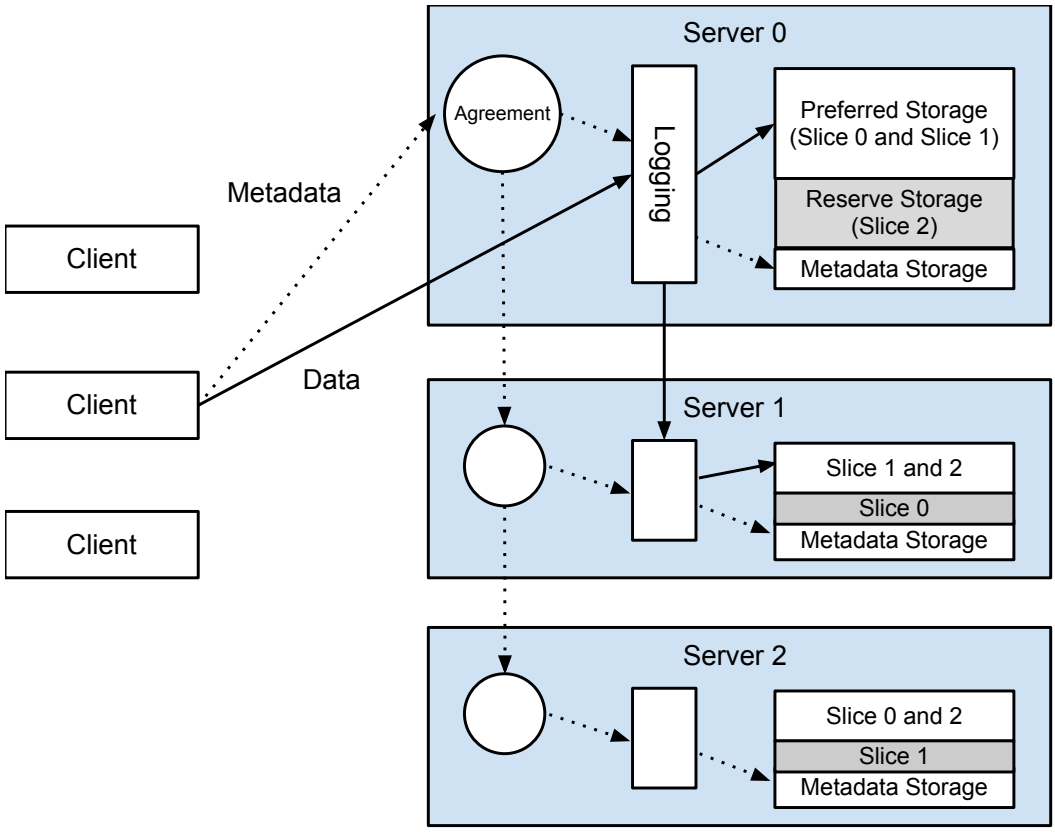

Figure 2.1: Data and metadata flow for a request to update a block in slice 1.

quests are then logged and executed, and replies are sent to the client.

Gnothi splits metadata and data. Metadata is updated using state machine replication and is replicated at all $2f + 1$ replicas, but data is replicated to just $f + 1$ replicas. A replica marks a data block as *COMPLETE* or *INCOMPLETE* depending on whether or not the replica holds what it believes to be the block's current version.

- A block is *COMPLETE* at a replica if the replica stores a version of the block's data that corresponds to the latest update to the block recorded in that replica's metadata.

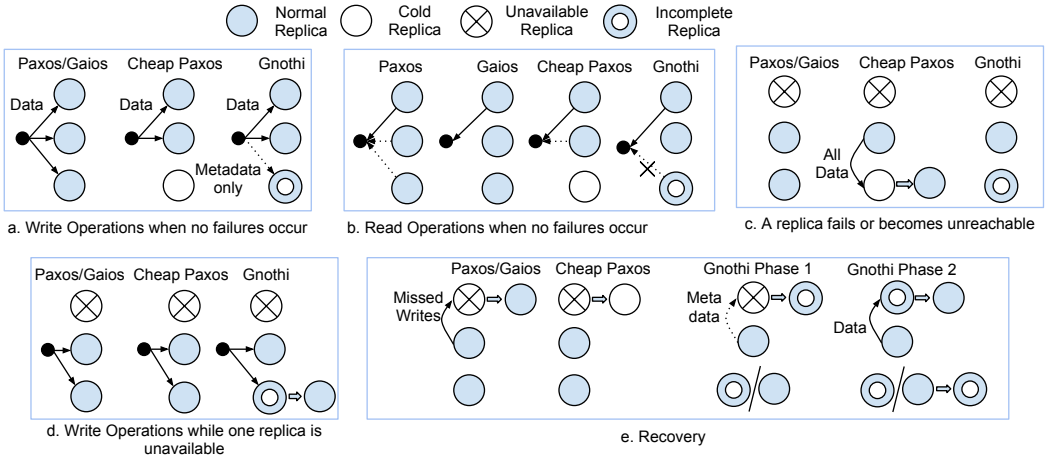

Figure 2.2: Gnothi protocols. We only show the logical flow of data and metadata in the figure, not actual messages. Consensus messages among servers are omitted, and we only show the flow and state for a single block. Multiple blocks are distributed among servers, so that each replica holds both *COMPLETE* and *INCOMPLETE* blocks.

- A block is *INCOMPLETE* at a replica if the replica's metadata records a version of the block that is more recent than the latest data stored at the replica for that block.

Note that the concepts of *COMPLETE* and *INCOMPLETE* are different from those of *Fresh* and *Stale*. A block is *Fresh* if it contains the data of the latest update to that block and is *Stale* if it contains a previous version. In Gnothi, a *COMPLETE* block can be *Stale*. For example, this can happen when a node becomes disconnected and misses both the data and metadata update. Section 2.2.3 discusses how to avoid reading a *Stale* block.

When no failures or timeouts occur, Gnothi maps each block n to one of $2f+1$ *slices* and stores each slice on $f+1$ preferred replicas, from replica $n\%(2f+1)$ to replica $(n-f)\%(2f+1)$. This ensures that the $2f+1$ slices are evenly distributed among different replicas, and that each replica is in the preferred quorum of $f+1$ different slices, which are the *PREFERRED* slices for that replica.

When failures or timeouts occur, a data block might be pushed to reserve storage on replicas out of its preferred quorum. We will show the detailed protocol later.

To simplify the description, we say that a block is *PREFERRED* at a replica if the replica is a member of the block's preferred quorum. Otherwise, we say that the block is *RESERVED* at that replica. We similarly say a request (read, write) is *PREFERRED*/*RESERVED* at a replica if it accesses a *PREFERRED*/*RESERVED* block at the replica.

Each replica allocates a preferred storage to store the data of *PREFERRED* writes, a reserve storage to store the data of *RESERVED* writes, and a relatively small metadata storage for each block's version and status.

2.1.3 Protocol Overview

This section presents an overview of Gnothi's protocol and compares Gnothi with the asynchronous full replication used by Paxos [61, 62], the asynchronous partial replication used by Cheap Paxos [63], and the state-of-the-art Paxos-based block replication system Gaios [19].

Figure 2.2.a shows a write operation when no failures occur. In Paxos and Gaios, a write operation is sent to, and executed on, all correct replicas. This seems redundant if our goal is to tolerate one failure: a natural idea is to send the write requests to two replicas first, and if they do not respond in time, try the third one [4]. Cheap Paxos adopts this idea by activating two replicas and leaving the other one as a cold backup [63, 108]. Gnothi incorporates a similar idea, but it still sends the metadata to the third replica, which executes the request by marking the corresponding data block as *INCOMPLETE*. Later, we will see that this metadata is critical to reducing the cost of failure and recovery.

Figure 2.2.b shows a read operation when no failures occur. In Paxos, the

read is sent to all replicas and the client waits for two replies. The figure shows a common optimization that lets one replica send back the full reply and lets the others send back a hash or version number [26]. By using similar optimizations for its writes, Cheap Paxos executes the read on only two replicas. Gaios introduces a protocol that allows reads to execute on only one replica while still ensuring linearizability, and Gnothi uses Gaios's read protocol, with a slight modification to avoid reading *INCOMPLETE* blocks.

Figure 2.2.c shows what happens when one replica fails. Paxos and Gaios do not need special handling since the remaining two replicas hold all data. Cheap Paxos brings online the cold backup, which needs to fetch the data from the live replica: the system is unavailable until this transfer finishes, possibly for a long time if the system stores a large amount of data. Gnothi too needs no special handling, the third replica knows whether a block it stores is *COMPLETE* or not, so it can safely continue processing read requests by serving reads of *COMPLETE* blocks and redirecting reads of *INCOMPLETE* ones to the other replica. And it can also continue processing writes whose block belongs to the failed replica by storing data in its reserve storage.

Figure 2.2.d shows a write operation when a replica is unavailable. Paxos, Gaios, and Cheap Paxos do not need any special handling. For Gnothi, a replica may receive a *RESERVED* write and store it in its reserve storage to ensure that writes only complete when at least two nodes store their data. Read operations in this case are not different from those when no failure occurs.

Figure 2.2.e shows how recovery works when a replica that has missed some writes recovers or when a new replica replaces a lost one. Paxos and Gaios both need to fetch all missing data before processing new requests at the recovered replica. Cheap Paxos can just leave the recovered replica as the cold backup and does not need any special handling. Gnothi performs a two-phase recovery when a failed

replica recovers.

In the first phase, the recovering replica fetches missing metadata from the other replicas. Since metadata is updated on all replicas, this phase of recovery proceeds as in a traditional RSM. The recovering replica then proceeds to store and mark as *COMPLETE* all the data blocks it has received; all remaining blocks referred in the received metadata are marked as *INCOMPLETE*. By the end of the first phase, the recovering replica can serve write requests even though full recovery is not complete yet: at this point the system stops consuming additional reserve storage on other replicas. Since the size of metadata is small, this phase is fast, and thus it is often not necessary to allocate a large reserve storage.

In the second phase, the recovering replica re-replicates all missing or stale *PREFERRED* blocks. Gnothi performs this step asynchronously, so it can balance recovery bandwidth and execution bandwidth while still guaranteeing progress. Depending on the status of the recovering replica, there are two possible cases here: if all data on disk is lost, the recovering replica needs to rebuild its whole disk; if the data on disk is preserved, the recovering replica just needs to fetch the updates it missed during its failure. Note that Gnothi can continue processing reads and writes to all blocks during the second phase. If a node receives a read request for an *INCOMPLETE* block, it rejects the request, and the client retries with another replica.

2.1.4 Summary

Tables 2.1 and 2.2 summarize the costs of Gnothi and of previous work. In read cost, write cost, and space, Gnothi dominates Paxos, Gaios, and Cheap Paxos, improving on each in at least one dimension and approximating most in the others. For recovery and availability, Gnothi can perform the heavy data transfer in the background concurrently with serving new requests, while in Paxos and Gaios, the

recovering replica must wait for the transfer to finish before serving requests, and in Cheap Paxos, the whole system must halt until the transfer completes.

Protocol	Write	Read	Space
Paxos	2f+1	2f+1	2f+1
Gaios	2f+1	1	2f+1
Cheap Paxos	f+1	f+1	f+1+f (Cold)
Gnothi	f+1	1	f+1+Δf $0 < \Delta \leq 1$

Table 2.1: Cost of Gnothi and previous work when there are no failures.

Protocol	Failure	Recovery (Disk survived)	Recovery (Disk replaced)
Paxos	0	O(NB)	O(S)
Gaios	0	O(NB)	O(S)
Cheap Paxos	O(S) (Blocking)	0	0
Gnothi	0	O(Nb)+O(NB)	O(Nb)+O($\frac{f+1}{2f+1}$S)

Table 2.2: Cost of Gnothi and previous work during failure and recovery. S is the total storage space, N is the number of unique updated blocks missed by the recovering replica, B is the block size, and b is the metadata size for each block.

2.2 Detailed Design

This section presents in detail how Gnothi stores and accesses data and metadata, and how it performs recovery after a replica fails.

2.2.1 Data and Metadata

Gnothi splits the storage space into $2f + 1$ slices, with each replica in the preferred quorum of $f + 1$ slices. A replica stores the data of its $f + 1$ *PREFERRED* slices in its preferred storage, and allocates space for f *RESERVED* slices in its reserve storage. When all replicas are available, blocks are always written to preferred storage, but when some replicas are not available, blocks are stored in the reserve storage of replicas outside the block's preferred quorums.

If the per-slice size of preferred and reserve storage are the same, then the system can remain available indefinitely even if f replicas fail, but at the cost of $2f + 1$ physical blocks for each logical block. In Section 2.2.5, we will show that, given Gnothi's fast recovery, a much smaller reserve storage is likely to suffice for many workloads. For now, let us assume that preferred and reserve storage have the same per-slice size.

In processing updates, Gnothi separates data and metadata. The data is carried in a PrepareData message, while the corresponding metadata is carried in a WriteData message; we will detail the messages' format in the following subsections. A client first sends PrepareData; upon receiving the message, a replica first logs it to disk and then stores it in a buffer until it receives the corresponding WriteData and can perform the actual write. We call the buffer the PrepareData buffer in the following sections. To avoid overflowing a replica's PrepareData buffer, Gnothi sets an upper bound on how many outstanding PrepareData requests a single client can buffer. If a replica finds its PrepareData buffer for a client is full, it stops receiving messages from that client until the buffer has room. Once it knows that the PrepareData has been stored by enough replicas, the client proceeds to send the WriteData message. Replicas run an agreement protocol to guarantee that WriteData messages are processed in the same order by all correct replicas.

It would be tempting for the clients to send PrepareData and WriteData messages in parallel. However, this optimization does not work, because a client failure may leave the system in a state where metadata (WriteData) is fully replicated while data (PrepareData) is completely lost: each replica would mark the corresponding block as *INCOMPLETE*, and the system would be unable to process read requests for this block. Sending PrepareData before WriteData—the approach used in Gnothi—eliminates this problem, but it introduces the possibility of a new problem of its own: a buffered PrepareData on a replica may never be written

to its target location. For example, a client could fail after sending the PrepareData message but before sending the WriteData message. To garbage-collect such buffered PrepareDatas, a client includes a client sequence number with each PrepareData message and WriteData message it sends, and a replica discards a buffered PrepareData if it processes a WriteData message with a higher sequence number. When the client fails and recovers, it sends to all replicas a special "new epoch" command. Replicas process the new epoch command using the same agreement protocol used to order WriteData messages: hence, by the time replicas enter a new epoch, they have processed the same sequence of WriteData messages. Once the new epoch command completes, all replicas can discard all buffered PrepareDatas in the previous epoch. Notice that if the failed client does not recover, the replica cannot discard buffered PrepareDatas; in asynchronous replication, it is impossible to know whether a client has permanently failed or is just slow. If the cost of a few megabytes per permanently-failed client is too high, the system can rely on an administrator or on a very long timeout (say, one day) to detect and remove the permanently failed client and clear its buffer.

Gnothi keeps metadata for each block: an 8-byte version number assigned by agreement to identify the block's last update, and an 8-byte requestID to connect the block to the PrepareData message. The version number and requestID are primarily used in failure recovery: we will show why Gnothi needs them and how to use them later. In addition, each replica keeps one bit for each block to identify whether or not the block is *COMPLETE* on the replica.

2.2.2 Write Protocol

The write protocol is illustrated in Figure 2.3:

① A client sends PrepareData(requestID, block data) to $f+1$ replicas, using the pair (clientID, clientSeqNo) to achieve a unique requestID. At first, the client

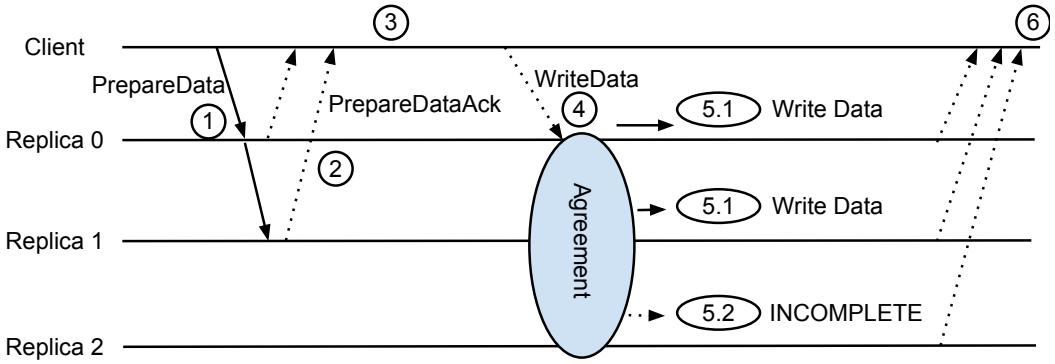

Figure 2.3: Write Protocol

targets the block's preferred quorum, but if a timeout occurs, the client tries other replicas. To prevent the client's network from becoming a bottleneck for sequential access we use chain replication [47, 102]: the client sends the data to one replica which forwards it to the next, and so on.

② A replica receiving the PrepareData puts it into a PrepareData buffer, logs it to disk, and sends a PrepareDataAck(requestID) to the client.

③ The client waits for $f + 1$ PrepareDataAcks. If there is a timeout, the client repeats Step ①, choosing some other replicas. When the network is available and message delivery is timely, this step is guaranteed to terminate as long as at least $f + 1$ replicas are capable of processing requests.

④ The client sends WriteData(requestID, block number) through the agreement protocol so that all replicas receive the same sequence of write commands. Gnothi uses code from ZooKeeper for agreement, but other Paxos-like protocols [19, 31] could be used.

⑤ After agreement, when a replica receives a WriteData message it updates its metadata storage and tries to find the corresponding PrepareData in the PrepareData buffer by using the requestID as the identifier. There are three possible cases:

⑤.① The replica has both the WriteData and the corresponding PrepareData messages, and this is a *PREFERRED* write for the replica. The replica then writes the data to its preferred storage and marks the corresponding block as *COMPLETE*. Note that the write operation can be performed asynchronously, because the PrepareData message is already logged to disk in Step ②: as shown in Section 2.4.2, writing asynchronously can improve the throughput of the system under random workloads because it gives the disks more opportunities to reorder requests.

⑤.② The replica has the WriteData message but no corresponding PrepareData message. The replica then marks the corresponding block as *INCOMPLETE*.

⑤.③ The replica has both the WriteData and the corresponding PrepareData messages, and this is a *RESERVED* write for the replica. The replica then writes the data to the reserve storage and marks the corresponding block as *COMPLETE*. This case happens only when there are unavailable or slow replicas, so it is not shown in Figure 2.3.

In all cases, the replica sends a WriteAck(requestID) back to the client.

⑥ The client waits for $f + 1$ WriteAcks. If there is a timeout, the client repeats Step ④. If the WriteData has already been processed, the replicas send a WriteAck reply [26]; otherwise, they process the write request. Assuming there are at least $f + 1$ functioning replicas, the client is guaranteed to get enough WriteAcks eventually.

To argue correctness, we observe that Step ③ guarantees that PrepareData is received by at least $f+1$ replicas and thus will not be lost; the agreement protocol guarantees that WriteData is eventually received by all correct replicas and thus will not be lost; and the agreement protocol provides the write linearizability guarantee. Notice that in Step ⑥, WriteAcks may not always come from the same nodes that stored data and sent PrepareDataAcks in Step ②, but this is not a problem since the system is still protected against f failures. If one of the nodes storing data is

slow, temporarily unavailable, or crashed but can recover locally, it will catch up with others using standard techniques [61, 62] and process the WriteData, so that the write will survive even if another node fails. Conversely, if a node permanently loses its data, the recovery protocol must restore full redundancy by fetching the failed node's state from the remaining replicas, but this case is no different whether the node that received the PrepareData and then permanently crashed did so before or after sending a WriteAck.

2.2.3 Read Protocol

For reads, we use the Gaios read protocol [19], modified slightly to handle *INCOMPLETE* blocks. (Steps ②-④ below are the same as described for Gaios):

① A client sends a Read (block number, replica ID) to the current agreement leader node, stating that it wants to read that block from a specific target replica. Usually, the target is the first replica in the block's preferred quorum.

② The leader buffers the Read request and queries all other replicas: "Am I still the leader?"

③ If the leader receives at least f "Yes" responses, it continues. Otherwise, it does nothing. This can happen if a slow replica still believes to be the leader, while enough other replicas have already moved on. In this case, the client will timeout, restart from Step ①, and try another replica as the leader.

④ The leader attaches a version number to the Read and sends it to the target replica specified in the request. The version number is set to the number of write requests already agreed. This number is used in Step ⑤ to ensure that the target replica does not read stale data.

⑤ The target replica waits until the write with the specified version number is executed, and then it executes the Read. This synchronization prevents a slow replica from sending stale data to the client. There are two cases to consider:

⑤.① The corresponding block is *COMPLETE* : the target replica then sends the data to the client.

⑤.② The corresponding block is *INCOMPLETE*: the target replica sends an INCOMPLETE reply to the client. This allows the client to move to the next replica quickly, instead of waiting for a timeout.

⑥ If the client receives the data, it finishes the Read. If it receives INCOMPLETE or times out, it chooses another replica and restarts from Step ①. The client chooses the target replica in round-robin fashion starting with the preferred quorum, so that all replicas will be tried.

When no failures or timeouts occur, Step ⑤.① will always happen, since the client chooses a node from the preferred quorum as the target. When failures or timeouts occur, the client may try some other replicas, but during a period with timely message delivery, it will eventually succeed since some replica must hold the data.

Note that if the client issues a read and then a write to the same block before the read returns, the read can return the result of the later write. Gnothi assumes this is an acceptable behavior for block drivers [19], but a client can prevent it by blocking the later write when there is an outstanding read operation to the same block.

2.2.4 Failure and Recovery

Gnothi performs no special operations when replicas fail. A client may timeout in the read or write protocol and retry using some other replicas, or write data to some replicas not in the preferred quorum, which will store these *RESERVED* writes in their reserve storage.

Recovering a failed replica begins with replaying the replica's log. If the disk is damaged or the machine is entirely replaced, this step may fail but correctness

is not affected. What cannot be recovered from the log is fetched from the other replicas in two phases: first to be restored is the metadata, and then any data missing from the failed replica's preferred slices. The recovering replica can process new requests once the first phase is complete, and it is fully recovered and no longer counts against our f threshold when the second phase is complete.

Phase 1: Metadata recovery

Gnothi replicates metadata on each node, so metadata recovery proceeds as it would in traditional RSMs: the recovering replica sends to the primary the last version number it is aware of, to which the primary replies with a list of metadata records, if any, with higher version numbers. Besides the version number, each of these records includes a block number and a requestID.

For each received record, the replica then checks if it holds in its buffer a PrepareData with the same requestID: if so, it executes the write request and marks the block as *COMPLETE*. This check handles the case when a replica receives a PrepareData but fails before receiving the corresponding WriteData. In this case, the recovering replica should finish executing the write request, and the requestID is necessary to connect a PrepareData to its block. If there is no PrepareData in the buffer with the same requestID, the replica simply marks the block as *INCOMPLETE*.

When metadata recovery is complete, it is safe for the replica to process new requests, even though it may have some *INCOMPLETE* blocks. An update will overwrite the *INCOMPLETE* block, and a read will be redirected to other replicas with the *COMPLETE* block.

Gnothi transfers 24 bytes of metadata for each block during this phase. This is 6GB per terabyte of data using 4KB blocks and 24MB per terabyte for 1MB blocks, so the first phase typically takes a few seconds to a few minutes to complete.

Note that during this metadata transfer, the other replicas continue to process new reads and writes.

Phase 2: Re-replicate

In the second phase, the recovering replica retrieves from the others the data for all the *INCOMPLETE* blocks in its preferred storage, thus freeing those replicas' reserve storage. If a replica retains its data on its local disk, it just needs to fetch the modified blocks. This case typically occurs when a replica crashes and recovers, becomes temporarily disconnected from the network, or becomes temporarily slow. If a replica loses its on-disk data as a result of a hardware fault, it needs to rebuild its storage by fetching all blocks in its slices' preferred storage.

This phase can take a long time, depending on how many blocks that need to be fetched, but it is needed only to free the reserve space of other nodes so that they are better equipped to mask future failures: once the replica recovers its metadata, it can process all writes to its slices, and it can process reads to the subset of blocks that are locally *COMPLETE*. Gnothi performs re-replication as a background task that can be throttled to balance the resources used for re-replication and for processing new client requests. Even if new client requests are processed at a high rate and re-replication proceeds at a low rate, re-replication will still eventually complete because the recovering replica's metadata allows it to process new requests while it is still catching up re-replicating missed old updates.

Every replica periodically checks its reserve storage: for every *RESERVED* block, the reserved replica communicates with the block's preferred replicas to check whether the block is *COMPLETE*, and if a *RESERVED* block is *COMPLETE* on all its preferred replicas, then the reserved replica can safely delete the block from its reserve storage.

2.2.5 Reducing replication state

Each replica needs to reserve space for f *RESERVED* slices. It is always safe to set the size of reserve storage to be f times a slice size, so that it can absorb any number of writes to each slice. This approach needs to pay a storage cost of $2f + 1$ blocks for one block, since a data block is stored on a preferred quorum of $f + 1$ replicas, and the other f replicas must reserve space for this block in the reserve storage: when $f = 1$, a replica must then allocate one third of its storage space for reserve storage, and more when f is larger. This is the same space overhead of the standard approach of Paxos or Gaios, which may be acceptable. When reducing replication costs is a concern, however, Gnothi also enables allocating less space for reserve storage. The risk of this thriftier approach is that if a failed replica does not recover or is not replaced soon, the reserve storage can fill, preventing the system from processing additional writes. However, filling the reserve storage does not put safety at risk, since data is always written to $f + 1$ replicas. In general, Gnothi can allocate less space for reserve storage in any of the following cases: 1) the workload is read-heavy; 2) the workload is write-heavy but dominated by random writes so that the throughput is low; 3) the workload is write-heavy but has good locality. Our analysis of several disk traces suggests that, as long as the metadata is recovered quickly, allocating 10% of disk space as reserve storage is enough to guarantee write availability for many workloads.

Specifically, we analyze two sets of traces from Microsoft: one is collected by Microsoft Research Cambridge [78] and it consists of 23 one-week disk traces under different workloads; the other is collected on Microsoft's production servers [58] and consists of 44 disk traces, whose lengths vary from six hours to one day. We choose these two sets of traces because they are recent and because they contain a variety of workloads including compiling, MSN Storage, SQL Server, computation, etc. We calculate the maximum usage ratio for each trace. To be precise, $MaxUsage(T)$ is

the maximum number of different sectors written during any time interval of length T, divided by the total number of sectors.

In the Microsoft Cambridge Traces, only two of the 23 traces write to more than 10% of the disk space in a week. For the heaviest one, reserving 10% always allows at least 10 minutes to finish Phase 1 and recover all metadata before the system becomes unavailable to writes. A conservative administrator may reserve more for this workload.

In the Microsoft Production Server Traces, 38 of the 44 disk traces write to less than 10% of the space in their traces. For the heaviest one, reserving 10% always allows at least 10 minutes to complete Phase 1.

2.2.6 Metadata

Each replica stores both local and replicated metadata for every block. The local metadata consists of the *COMPLETE* bit for each block, and the replicated metadata includes the version number and requestID for each block.

In Gnothi, caching in memory the *COMPLETE* bit of each block is feasible in both size and cost. For example, with a small 4KB block each 1TB of disk storage requires about 30MB of *COMPLETE* bits. In May 2012, a commodity 2TB internal hard drive costs about $120 and a common 4GB memory DIMM costs about $25. This means that keeping *COMPLETE* bits in memory adds about 0.3% to the dollar cost of the disk data it tracks. Gnothi regularly stores checkpoints of the *COMPLETE* bits by writing the current state to local files.

The block number, version number, and requestID are 8 bytes each, and it would be costly for Gnothi to keep them all in memory. Gnothi uses a metadata storage design similar to that of BigTable [27, 74]. Each Gnothi node maintains in a local key-value store the mapping from logical block ID to version number and requestID. Metadata updates are logged to disk first as described before. Afterwards,

to update a record, Gnothi first puts the record in a memory buffer; then when the buffer is full, Gnothi sorts the buffer according to the key and then writes the whole buffer to a new file. A background thread merges these files when there are too many of them. Metadata writes and merges are fast, since they are sequential writes to disk. Our micro benchmark shows that this approach can sustain a throughput of about 200K writes per second, which is enough for our needs. Reading from metadata storage only occurs when Gnothi recovers a crashed or slow replica by fetching metadata from another replica: this case requires a sequential scan of the metadata, which is again fast. Individual read operations do not access metadata storage, since a read operation only needs to access the *COMPLETE* bit.

2.3 Implementation

We implement Gnothi by modifying ZooKeeper's source code. In particular: 1) we reuse ZooKeeper's network and agreement modules to replicate metadata; 2) we add chain replication to forward data; 3) we modify the read protocol to provide linearizable and scalable reads; 4) we replace ZooKeeper's storage module with one that supports preferred, reserve, and metadata storage; 5) we modify ZooKeeper's logging system to record PrepareData messages; 6) we implement recovery as described in Section 2.2.4.

We apply several additional modifications to improve performance: first, Gnothi modifies ZooKeeper's agreement module to incorporate batching [26, 31], which improves performance by about 10% for the sequential write workload. Second, Gnothi reuses memory buffers to reduce memory allocation. ZooKeeper's server is implemented in Java, and our profiling shows that the overhead due to memory allocation and garbage collection is quite substantial, especially if the block size is large (ZooKeeper is explicitly not designed for large data blocks). To alleviate this problem, we reuse allocated memory buffers, which is not hard since they all

have the same size. This device improves read performance by about 10-15% in our experiments.

2.4 Evaluation

2.4.1 Workload and Configuration

First, we evaluate the performance of Gnothi using micro benchmarks that issue both sequential and random reads and writes. We compare Gnothi's performance to a Gaios-like system that we implement (denoted in the following as G'), and to an unreplicated local disk. Both G' and Gnothi use the same code base; the only significant difference is that G' forwards all updates and stores all blocks at all replicas, while Gnothi processes each block at $f + 1$ of the $2f + 1$ replicas.

Second, we evaluate Gnothi in failure and recovery. We compare Gnothi with G' and Cheap Paxos in terms of availability, performance in the face of failures, and recovery time.

We use five machines as servers and five machines as clients. We run our performance evaluation experiments for two configurations: $f = 1$ (three servers) and $f = 2$ (five servers); we run the recovery experiments with $f = 1$. Gnothi's design calls for using a disk array for data storage and an additional disk to store log and metadata, but since our machines have only two Western Digital WD2502ABYS 250GB 7200 RPM hard drives, we evaluate Gnothi in a configuration where one disk is used as preferred and reserve storage, while the other stores the log and metadata. Each machine is equipped with a 4-core Intel Xeon X3220 2.40GHz CPU and 3GB of memory. For all experiments, we allocate 96GB of logical storage space replicated across nodes by the system under test. All machines are connected with 1Gbps ethernet.

For each experiment, we make sure there are enough client processes and

outstanding requests to saturate the system; we make sure the experiment is long enough so that the write buffers are full; and we use the last 80% of requests to calculate the stable throughput. In all experiments, the read and write batches at each replica consist of, respectively, 100 and 10 requests. The values of other parameters (number of clients, number of outstanding requests per client, etc) depend on the block size (4K, 64K, 1M) and workloads (sequential/random write/read), and we do not list all of them. In general, sequential workloads and small blocks need more outstanding client requests to saturate the system; random workloads and big blocks need fewer; and random workloads with small blocks need a longer time to saturate the write buffer. For example, for the 4KB sequential write workloads, we use 30 clients, each with 200 outstanding requests, to saturate the system; for the 4KB random write workloads, three clients with 200 requests each are enough, but we need to run the experiments for three hours to measure the stable throughput; and for the 1MB sequential write workloads, it takes just five clients with 60 outstanding requests each to saturate the system.

2.4.2 I/O Throughput

Gnothi maximizes I/O throughput by executing reads and writes on subsets of disks.

Figure 2.4 shows the random I/O performance for $f = 1$ and $f = 2$. For random workloads, the bottleneck of the system is the seek time for each replica's data disk.

For write operations, Gnothi is 40-64% faster than writing to local disk or to G' for $f = 1$ and 53-75% for $f = 2$. Gnothi's advantage comes from only having to perform the writes at 2/3 (for $f = 1$) or 3/5 (for $f = 2$) of the nodes. As expected [19], the random write performance of G' is close to that of a single local disk because all replicas process all updates.

For read operations, Gnothi and G' perform identically since they use the

same read protocol. Gnothi/G' is 2.5-3.4 times faster than a single local disk for $f = 1$ and 3.3-6.1 times faster for $f = 2$, because it executes each read on one replica. For small requests, the improvement factor can exceed $2f + 1$ since each replica is responsible for $1/(2f + 1)$ of the data, and thus the average seek time is reduced.

Note that for small random I/O, the local per-disk write bandwidth significantly exceeds the corresponding read bandwidth. The reason is that, once writes are committed to the log, we can buffer large numbers of writes before writing them back to the data disk, allowing the disk scheduler more opportunities to minimize seek and rotational latency. Reads, on the other hand, must be processed immediately, so the scheduler has fewer opportunities for optimization. Taking for example the 4KB random workload, a local disk can process 383 random writes per second, while it can only process about 155 random reads per second if there are 100 concurrent read requests.

Figure 2.5 shows the effect of a burst of random writes when $f = 1$ and the system buffers are not full. During the first few seconds, since writes are logged to the logging disk, buffered in memory, but not bottlenecked by flushing to the data disk, Gnothi's throughput is much higher than that of the data disk write back. Then, when the operating system detects that more than 10% of the system memory is dirty, it begins to write back data to disk at the same rate it receives new requests, and Gnothi slows down. Figure 2.4 shows the stable write throughput, where, to eliminate the effects of the initial spike, we run our experiments for sufficiently long (more than 3 hours) and calculate the throughput of the last 80% of requests.

Figure 2.6 shows the sequential I/O performance with $f = 1$ and $f = 2$.

For the sequential write workload with $f = 1$, Gnothi can achieve about 60MB/s with a 4KB block size and about 90MB/s with a 1MB block size. The bottleneck for 4KB block size is probably ZooKeeper's agreement, which is processing

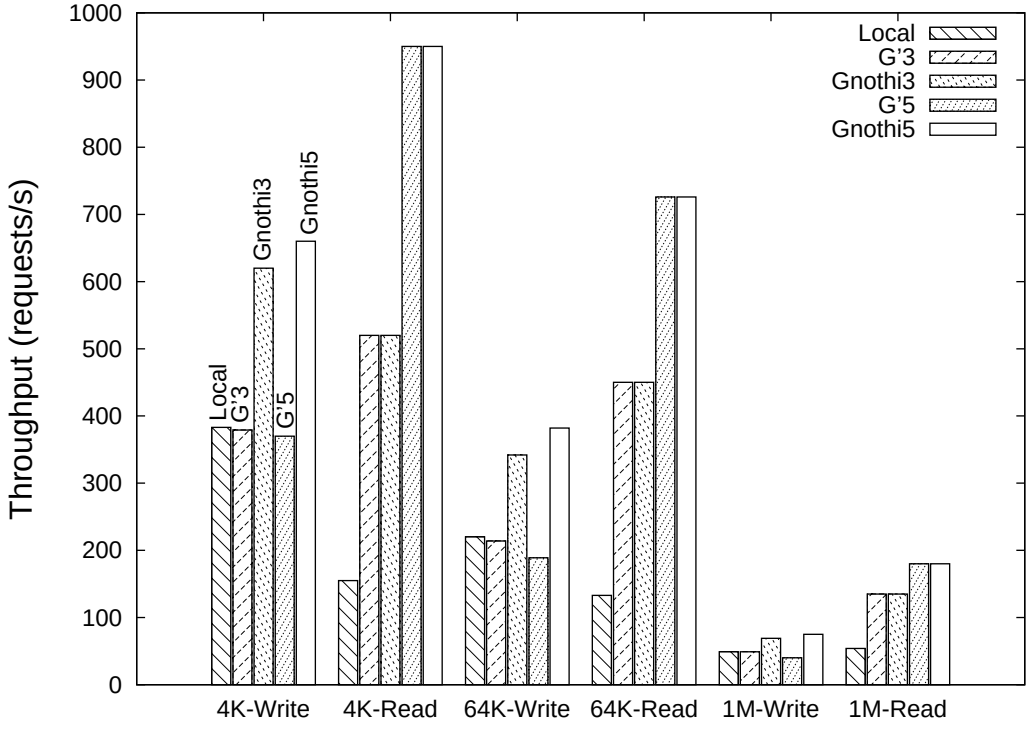

Figure 2.4: Random I/O with 3 (f=1) and 5 (f=2) servers.

about 15K updates per second. For 1MB requests, our profiler shows that the bottleneck is probably Java's memory allocation and garbage collection, so customized memory management or a C implementation may achieve better performance. Compared to G', Gnothi is 44% to 56% faster because Gnothi directs writes to subsets of nodes.

For the read workload, Gnothi/G' can achieve a total bandwidth of about 250MB/s with 1MB blocks. One problem with reads is that if we use only 1 client, the client network becomes a bottleneck, and if we use multiple clients, then the workload is not fully sequential. This problem is more severe for small requests.

Compared with the $f = 1$ case, Gnothi's throughput for 64K and 1MB

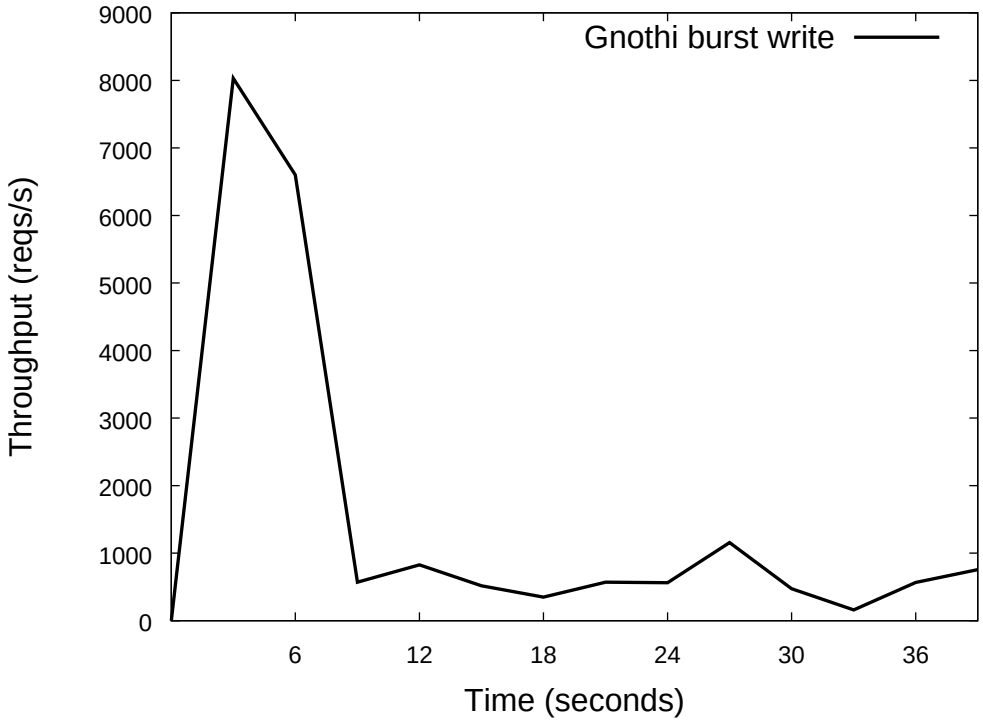

Figure 2.5: Burst writes. In the default configuration, Linux starts to flush dirty data to disk if 10% of total system memory pages are dirty.

writes increases by about 10% when $f = 2$. In the 4KB case the bottleneck is agreement, so there is almost no improvement. G's throughput slightly decreases since its replication cost is higher. For reads, Gnothi/G' scales throughput by nearly a factor of 4 compared to a single disk.

2.4.3 Failure Recovery

Gnothi does three things to maximize availability and recovery speed. First, it fully replicates metadata, allowing the system to remain continuosly available in the face of up to f failures despite partial replication of data. Second, partial replication

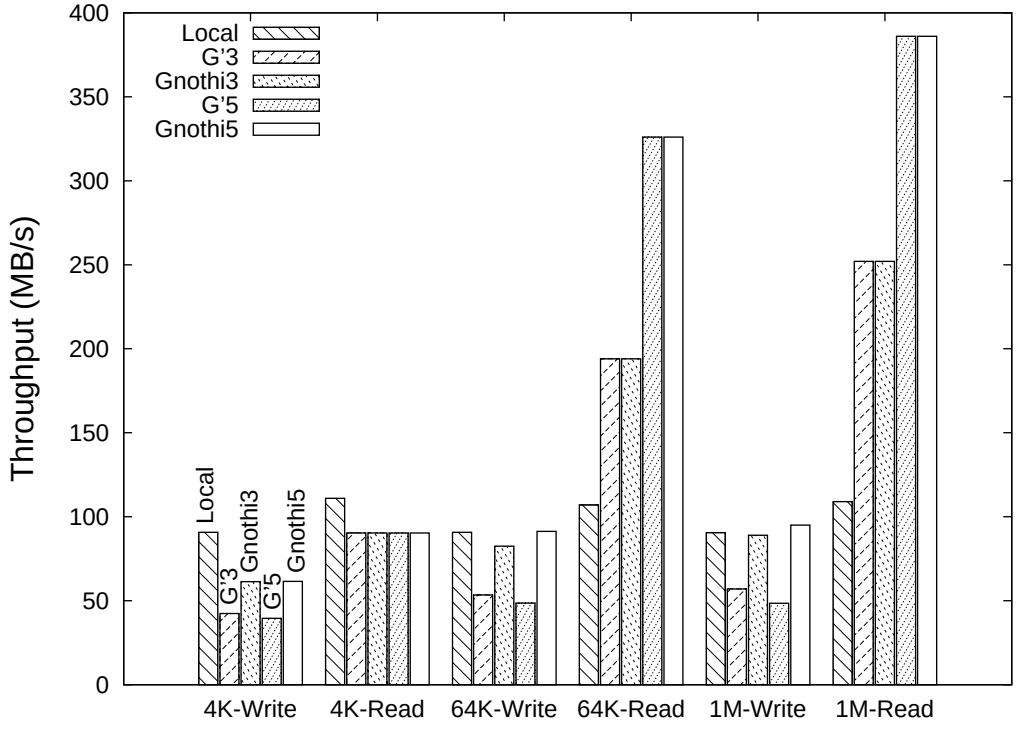

Figure 2.6: Sequential I/O with 3 (f=1) and 5 (f=2) servers.

of data reduces recovery time, because the recovering node only needs to fetch $(f+1)/(2f+1)$ (e.g. 2/3 for $f=1$) of the data. It also improves performance during recovery, because once metadata is restored, full block updates are only sent to and executed on the block's preferred quorum. Third, separation of data and metadata improves system throughput during recovery and reduces recovery time. The recovering node can catch up with other nodes even if they continue to process new updates at a high rate. In particular, since processing metadata is faster than processing full requests, Phase 1 of recovery can always catch up with missed and new requests. Once Phase 1 is complete, the recovering replica no longer falls behind as new requests are executed, since it can process and store all new block updates

directed to it, while it fetches old update bodies for all *INCOMPLETE* blocks in its preferred slices.

Figures 2.7 and 2.8 look at two recovery scenarios. Figure 2.7 shows the case when a node temporarily fails and then recovers by fetching just the updated blocks it missed. Figure 2.8 shows the case when a node permanently fails and is replaced by a new node that must fetch all data from others. We run both experiments with $f = 1$, 4KB blocks, and a sequential write workload. We choose the sequential write workload because it is the most challenging workload for recovery, since during recovery the clients are writing new contents at a high speed, which consumes a large portion of the network and disk bandwidth from the servers.

In Figure 2.7, we kill one server 60 seconds after the experiment starts and restart it 60 seconds later. Here both Gnothi and G' suffer a brief drop in throughput while they wait for timeout and then continue without the failed node. After the replica restarts at time 120, it takes about 110 seconds (to time 230) to recover from its local disk (mainly replaying logs), and about 22 seconds (to time 252) to join the agreement protocol. Then Gnothi spends 26 seconds (to time 278) in Phase 1, during which the recovering replica fetches write metadata (but not data) and marks all updated blocks as *INCOMPLETE*. Once Phase 1 completes, the recovering replica begins servicing new requests, writing new writes to its local state, and marking updated blocks as *COMPLETE*. After Phase 1 completes, the recovering replica also begins Phase 2 of recovery by fetching from other replicas *INCOMPLETE* blocks in its preferred slices. Phase 2 completes at time 530, at which point recovery is complete, and Gnothi returns to its original throughput.

G's throughput starts at 50MB/s and remains the same while the failure occurs. After the replica resumes operation, in order to complete the recovery at time 530, it must throttle the rate at which it services new requests to about 16 MB/s.

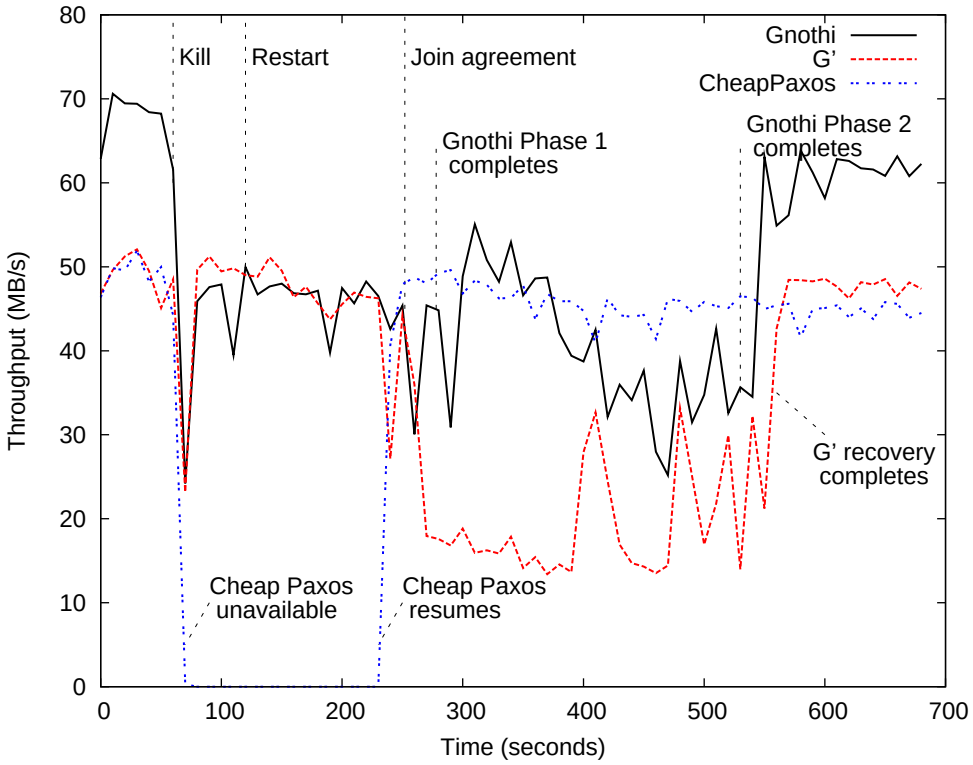

Figure 2.7: Failure recovery (catch up).

Cheap Paxos is unavailable from time 30 to 230, since there is only one available replica and since it does not have sufficient time to copy 96GB to a spare machine. When the replica resumes operation, Cheap Paxos can immediately go back to normal (time 230) since it does not process any new requests during the failure period.

In Figure 2.8, one server is killed 300 seconds after the experiment is started and is replaced 300 seconds later by a new server whose local disk is initialized

and needs to be fully rebuilt. Gnothi takes about 80 seconds[2] in Phase 1 to fetch metadata from the primary. After Phase 1 completes, the recovering replica begins servicing new requests, and at the same time, re-replicating its disk by fetching blocks from others. The recovering replica completes re-replication at time 3400, and during this period, it can service new requests at a rate of about 48 MB/s.

G' can also complete recovery at time 3400, but during this period, it can only service new requests at a rate of about 16 MB/s.

Cheap Paxos is unavailable before re-replication completes, but since it uses all its bandwidth to perform recovery, it ends re-replication at time 2400.

Comparing Figure 2.7 and Figure 2.8, Gnothi's catch-up recovery takes less time than full re-replication (410 seconds vs 2800 seconds), but catch-up inflicts a bigger hit on throughput because when re-replicating all blocks, the disk accesses are always sequential, and when re-replicating a subset of them, the disk accesses may be random. This means the recovery cost per block is smaller in full re-replication, though the total number of blocks to be fetched is larger, and this results in higher client throughput but longer recovery time.

Both Gnothi and G' can divide resources between servicing new requests and fetching state for recovery by tuning the time interval (in milliseconds) for a replica to issue a 16 MB state fetch request: a smaller number means more aggressive recovery. In Figures 2.7 and 2.8, we configure this parameter so that Gnothi and G' can recover in similar time, while still providing reasonable throughput for new requests. In Figures 2.9 and 2.10, we show the effect of different configurations.

Figure 2.9 shows that Gnothi can always catch up, so the administrator

[2]This may seem like a long time considering that the size of metadata is about 576MB. This figure is due to three reasons: first, the metadata storage uses a design based on Bigtable (see Section 2.2.6), which logs metadata first and garbage-collects outdated log entries periodically in the background: in such a design, the actual size of metadata to be scanned is usually larger than 576MB, because certain outdated metadata may not have been garbage-collected yet. Second, although ideally, disk scan and network transfer of metadata should be parallelized to maximize throughput, we have not implemented this optimization. Third, clients may still send requests to replicas during Phase 1, causing contentions on the network.

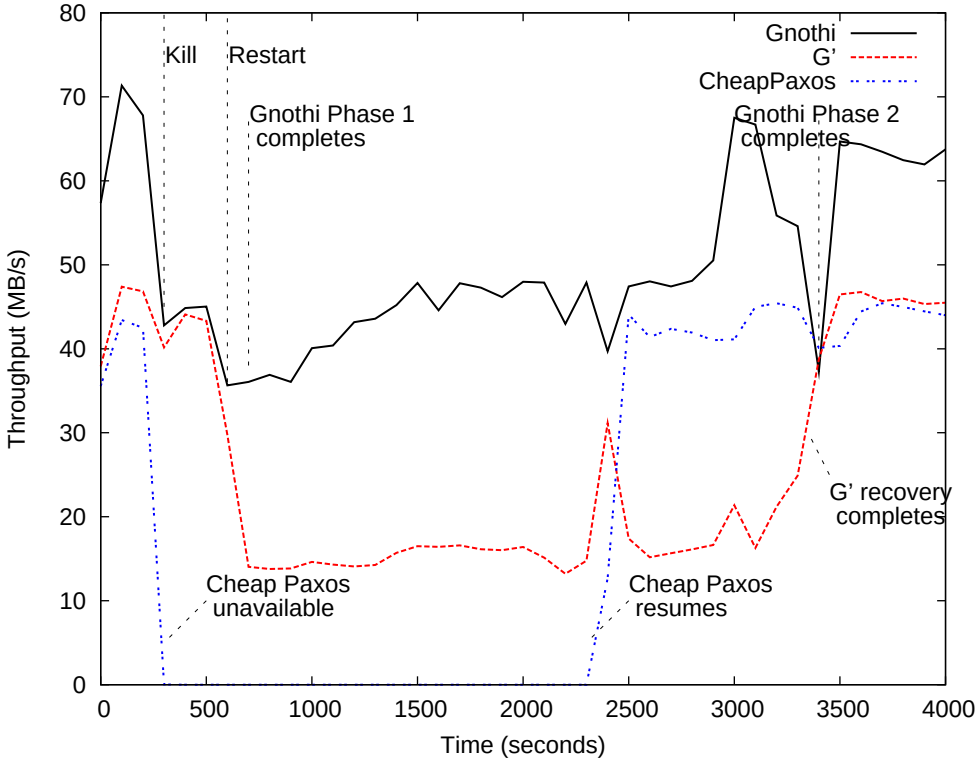

Figure 2.8: Failure recovery (re-replicate).

can tune this parameter to balance resources used for recovery and for processing new requests. Conversely, if G' sets this parameter too high (not aggressive), the recovering replica never catches up. For example, in Figure 2.10, the replica in the experiment with parameter 600 does not catch up, since the recovery speed is similar to the speed of processing new requests. Gnothi is almost always better than G' in our experiments: if recovery times are similar, Gnothi can provide better throughput during recovery; and if throughputs are similar, Gnothi can recover faster.

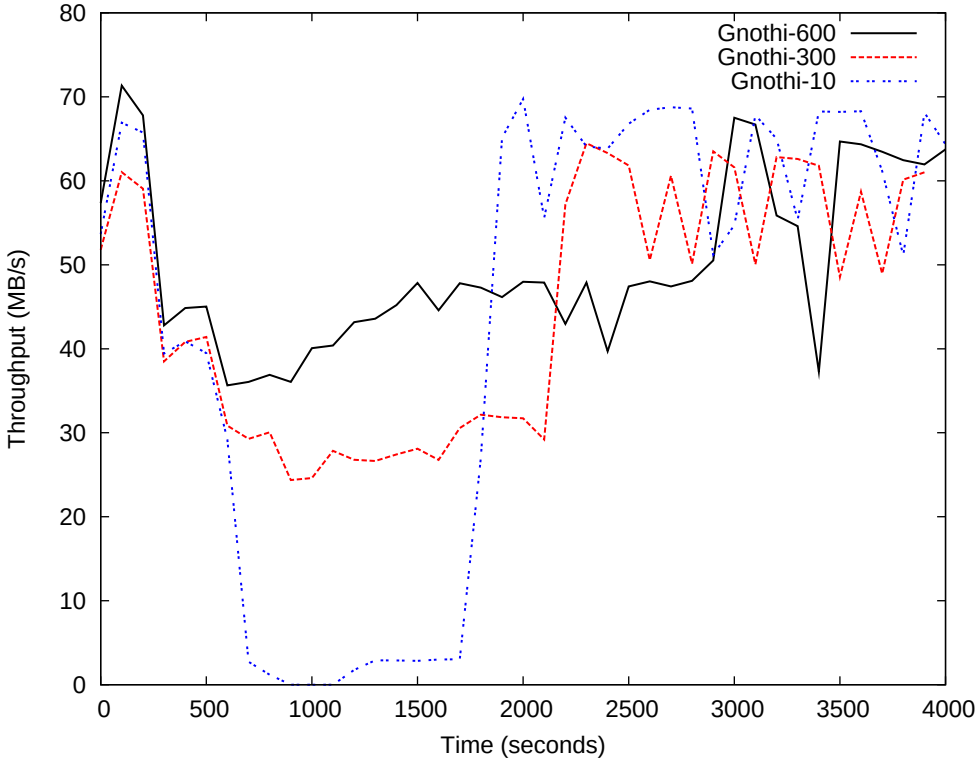

Figure 2.9: Gnothi with different recovery values.

2.5 Conclusion

Gnothi is an asynchronous replicated storage system with low replication cost and fast failure recovery. Gnothi accomplishes this by separating data and metadata and replicating metadata on all replicas, while replicating data on subsets of them.

Gnothi demonstrates that full replication of metadata can 1) ensure that the system works correctly despite partial replication of data and 2) speed up recovery when replicas fail. The evaluation shows that Gnothi achieves higher throughput and availability than previous work.

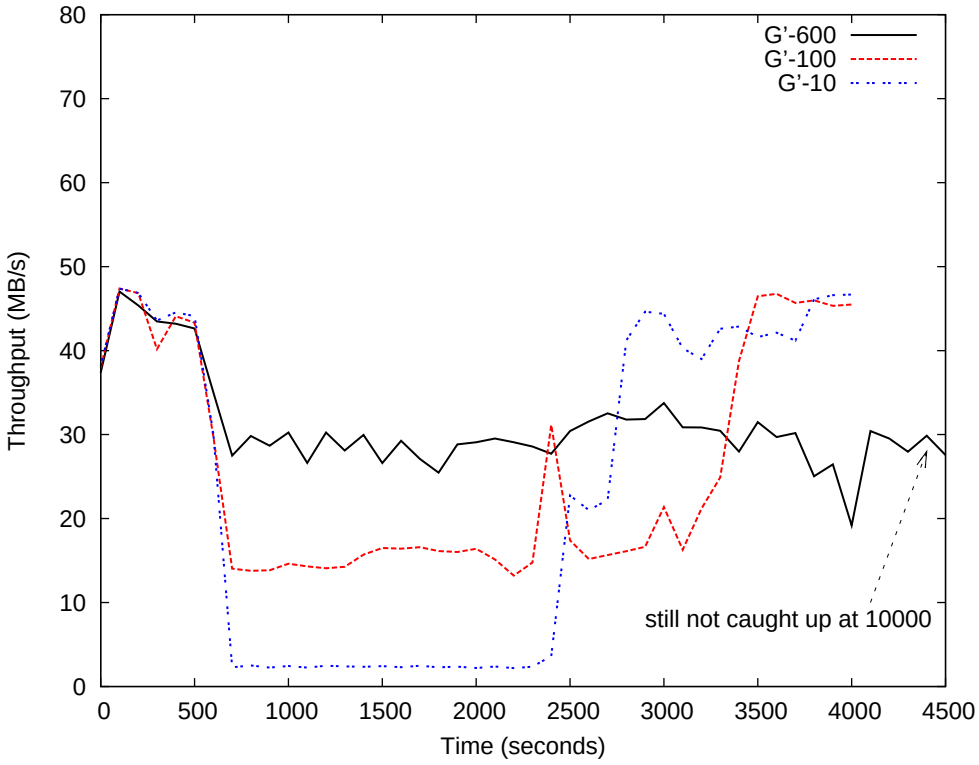

Figure 2.10: G' with different recovery values.

Chapter 3

Salus: Robust and Scalable Storage

Many distributed storage systems experience the tension between robustness and scalability, because stronger protection techniques are usually more expensive and thus are rarely deployed at large scale.

This chapter describes the design and implementation of Salus,[1] a scalable block store in the spirit of Amazon's Elastic Block Store (EBS) [7]: a user can request storage space from the service provider, mount it like a local disk, and run applications upon it, while the service provider replicates data for durability and availability.

What makes Salus unique is its dual focus on *scalability* and *robustness*. Some recent systems have provided end-to-end correctness guarantees on distributed storage despite arbitrary node failures [26, 31, 73], but these systems are not scalable—they require each correct node to process at least a majority of updates. Conversely, scalable distributed storage systems [9, 10, 12, 23, 27, 44, 66, 72, 100] typically protect some subsystems like disk storage with redundant data and checksums, but fail to

[1] Salus is the Roman goddess of safety and welfare.

protect the entire path from client PUT to client GET, leaving them vulnerable to single points of failure that can cause data corruption or loss.

Salus provides strong end-to-end correctness guarantees for read operations, strict ordering guarantees for write operations, and strong durability and availability guarantees despite a wide range of server failures (including memory corruptions, disk corruptions, firmware bugs, etc). To scale these guarantees to thousands of machines and tens of thousands of disks, Salus leverages an architecture similar to scalable key-value stores like Bigtable [27] and HBase [12], but achieving this unprecedented combination of robustness and scalability presents several challenges.

First, to build a high-performance block store from low-performance disks, Salus must be able to write different sets of updates to multiple disks in parallel. Parallelism, however, can threaten the basic consistency requirement of a block store, as "later" writes may survive a crash, while "earlier" ones are lost.

Second, aiming for efficiency and high availability at low cost can have unintended consequences on robustness by introducing single points of failure. For example, in order to maximize throughput and availability for reads while minimizing latency and cost, scalable storage systems execute read requests at just one replica. If that replica experiences a *commission failure* that causes it to generate erroneous state or output, the data returned to the client could be incorrect. Similarly, to reduce cost and for ease of design, many systems that replicate their storage layer for fault tolerance (such as HBase) leave unreplicated the computation nodes that can modify the state of that layer: hence, a memory error or an errant PUT at a single HBase region server can irrevocably and undetectably corrupt data (see Section 3.4.1).

Third, additional robustness should ideally not result in higher replication cost. For example, in a perfect world Salus' ability to tolerate commission failures would not require any more data replication than a scalable key-value store such as

HBase already employs to ensure durability despite omission failures.

To address these challenges Salus introduces three novel ideas—pipelined commit, active storage, and scalable end-to-end verification—based on the principle of separating data from metadata.

Pipelined commit. By tracking the necessary dependency metadata, Salus' new pipelined commit protocol allows writes to be processed in parallel at multiple disks but guarantees that, despite failures, the system will be left in a state consistent with the ordering of writes specified by the client.

Active storage. To prevent a single node from corrupting data or metadata, Salus replicates both the storage and the computation layer. To reduce the cost of storage replication, Salus incorporates the idea of Gnothi, requiring only $f + 1$ data replicas to tolerate f failures. To reduce the cost of replicating computation, Salus applies an update to the system's persistent state only if the update is agreed upon by *all* of the replicated computation nodes. This approach, again, requires only $f+1$ replicas to tolerate f failures. We make two observations about active storage. First, perhaps surprisingly, replicating the computation nodes can actually improve system performance by moving the computation near the data (rather than vice versa), a good choice when network bandwidth is a more limited resource than CPU cycles. Second, by requiring the *unanimous consent* of all replicas before an update is applied, Salus comes near to its perfect world with respect to overhead: Salus remains safe (i.e. keeps its blocks consistent and durable) despite two *commission* failures with just three-way replication—the same degree of data replication needed by HBase to tolerate two permanent *omission* failures. The flip side, of course, is that insisting on unanimous consent can reduce the times during which Salus is live (i.e. its blocks are available)—but liveness is easily restored by replacing the faulty set of computation nodes with a new set that can use the storage layer to recover the state required to resume processing requests.

Scalable end-to-end verification. Salus maintains sufficient metadata—a Merkle tree [18, 75]—for each volume so that a client can validate that each GET request returns consistent and correct data: if not, the client can reissue the request to another replica. Reads can then safely proceed at a single replica without leaving clients vulnerable to reading corrupted data; more generally, such end-to-end assurances protect Salus clients from the opportunities for error and corruption that can arise in complex, black-box cloud storage solutions. Further, Salus' Merkle tree, unlike those used in other systems that support end-to-end verification [3, 43, 69, 73], is scalable: each server only needs to keep the sub-tree corresponding to its own data, and the client can rebuild and check the integrity of the whole tree even after failing and restarting from an empty state.

We have prototyped Salus by modifying the HBase key-value store. The evaluation confirms that Salus can tolerate servers experiencing commission failures like memory corruption, disk corruption, etc. Although one might fear the performance price to be paid for Salus' robustness, Salus' overheads are low in all of our experiments. In fact, despite its strong guarantees, Salus often *outperforms* HBase, especially when disk bandwidth is plentiful compared to network bandwidth. For example, Salus' active replication allows it to halve network bandwidth while increasing aggregate write throughput by a factor of 1.74 in a well-provisioned system.

3.1 Requirements and model

Salus provides the abstraction of a large collection of virtual disks, each of which is an array of fixed-sized blocks. Each virtual disk is a *volume* that can be mounted by a client running in the datacenter that hosts the volume. The volume's size (e.g., several hundred GB to several hundred TB) and block size (e.g., 4 KB to 256 KB) are specified at creation time.

A volume's interface supports GET and PUT, which on a disk correspond to read and write. A client may have many such commands outstanding to maximize throughput. At any given time, only one client may mount a volume for writing, and during that time no other client can mount the volume for reading. Different clients may mount and write different volumes at the same time, and multiple clients may simultaneously mount a read-only snapshot of a volume.

We explicitly designed Salus to support only a single writer per volume for two reasons. First, as demonstrated by the success of Amazon EBS, this model is sufficient to support disk-like storage. Second, we are not aware of a design that would allow Salus to support multiple writers while achieving its other goals: strong consistency,[2] scalability, and end-to-end verification for read requests.

Even though each volume has only a single writer at a time, a distributed block store has several advantages over a local one. Spreading a volume across multiple machines not only allows disk throughput and storage capacity to exceed the capabilities of a single machine, but balances load and increases resource utilization. Further, storing data on multiple servers opens up the opportunities to improve availability and durability: if a server fails, the client can access another server holding a copy of the data; if the client fails, the user can quickly start a new client by recovering data from the servers.

To minimize cost, a typical server in existing storage deployments is relatively storage heavy, with a total capacity of up to 24TB [11, 99]. We expect a storage server in a Salus deployment to have ten or more SATA disks and two 1Gbit/s network connections. In this configuration disk bandwidth is several times more plentiful than network bandwidth, so the Salus design seeks to minimize network bandwidth consumption.

[2]More precisely, *ordered commit* (defined in Section 3.1.2) which implies FIFO-compliant linearizability.

3.1.1 Failure model

Salus is designed to operate on an unreliable network with unreliable nodes. The network can drop, reorder, modify, or arbitrarily delay messages.

For storage nodes, we assume that 1) servers can crash and recover, temporarily making their disks' data unavailable (transient omission failure); 2) servers and disks can fail, permanently losing all their data (permanent omission failure); 3) disks and the software that controls them can cause corruption, where some blocks are lost or modified, possibly silently [87] and servers can experience memory corruption, software bugs, etc, sending corrupted messages to other nodes (commission failure). When calculating failure thresholds, we only take into account commission failures and permanent omission failures. Transient omission failures are not treated as failures: in asynchronous systems a node that fails and recovers is indistinguishable from a slow node.

In line with Salus' aim to provide end-to-end robustness guarantees, we do not try to explicitly enumerate and patch all the different ways in which servers can fail. Instead, we design Salus to tolerate arbitrary failures, both of omission, where a faulty node fails to perform actions specified by the protocol, such as sending, receiving, or processing a message; and of commission [31], where a faulty node performs arbitrary actions not called for by the protocol. However, while we assume that faulty nodes will potentially generate arbitrarily erroneous state and output, given the data center environment we target we explicitly do not attempt to tolerate cases where a malicious adversary controls some of the servers. Hence, we replace the traditional BFT assumption that *faulty nodes cannot break cryptographic primitives* [89] with the stronger (but fundamentally similar) assumption that *a faulty node never produces a checksum that appears to be a correct checksum produced by a different node.* In practice, this means that where in a traditional Byzantine-tolerant system [25] we might have used signatures or arrays of message

authentication codes (MACs) with pairwise secret keys, we instead *weakly sign* communication using checksums salted with the checksum creator's well-known ID.

Salus relies on weak synchrony assumptions for both safety and liveness. For safety, Salus assumes that clocks are sufficiently synchronized that a ZooKeeper lease is never considered valid by a client when the server considers it invalid. Salus only guarantees liveness during *synchronous intervals* where messages sent between correct nodes are received and processed within some timeout [22].

3.1.2 Consistency model

To be usable as a virtual disk, Salus must ensure the *standard disk semantics* provided by physical disks. These semantics allow some requests to be marked as *barriers*. A disk must guarantee that all requests received before a barrier are committed before the barrier, and all requests received after the barrier are committed after the barrier. Additionally, a disk guarantees *freshness*: a read to a block returns the latest committed write to that block.

Salus implements a stronger guarantee: When there are no more than f (we use $f = 2$ in Salus) failures, Salus guarantees both freshness and a property we call *ordered-commit*: if a write request R commits, then all write requests that were sent by the client before R are guaranteed to eventually commit. Note that ordered-commit eliminates the need for explicit barriers since every write request functions as a barrier. Although we did not set out to achieve ordered-commit and its stronger guarantees, Salus provides them without any noticeable effect on performance.

Salus mainly focuses on tolerating arbitrary failures (in the sense specified in Section 3.1.1) of server-side storage systems, since they entail most of the complexity and are primarily responsible for preserving the durability and availability of data. Client commission failures can also be handled using replication, but this falls beyond the scope of this work.

3.2 Background

Salus' starting point is the scalable architecture of HBase/HDFS, which Salus carefully modifies to boost robustness without introducing new bottlenecks. We chose the HBase/HDFS architecture for three main reasons: first, it provides a key-value interface that can be easily modified to support a block store; second, it has a large user base that includes companies such as Yahoo!, Facebook, and Twitter; and third, unlike other successful large-scale storage systems with similar architectural features, such as Windows Azure Storage [23] and Google's Bigtable/GFS [27, 44], HBase/HDFS is open source.

HDFS HDFS [95] is an append-only distributed file system whose design is based on Google's File System [44]. It stores the system's namespace and membership information in a *NameNode* and replicates the data over a set of *DataNodes*. Each file consists of a set of blocks, and HDFS ensures that each block is replicated across a specified number of DataNodes (three by default) despite DataNode failures. HDFS is widely used, primarily because of its scalability.

HBase HBase [12] is a distributed key-value store whose design is based on Google's Bigtable [27]. It exports the abstraction of tables accessible through a PUT/GET interface. Each table is split into multiple *regions* of non-overlapping key-ranges (for load balancing). Each region is assigned to one *region server* that is responsible for all requests to that region. Region servers use HDFS as a storage layer to ensure that data is replicated persistently across enough nodes. Additionally, HBase uses a *Master* node to manage the assignment of key-ranges to various region servers.

Region servers receive clients' PUT and GET requests and transform them into equivalent requests that are appropriate for the append-only interface exposed by HDFS. On receiving a PUT, a region server logs the request to a write-ahead-log

stored on HDFS and updates its sorted, in-memory map (called *memstore*) with the new PUT. When the size of the memstore exceeds a predefined threshold, the region server *flushes* the memstore to a *checkpoint* file stored on HDFS.

On receiving a GET request for a key, the region server looks up the key in its memstore. If a match is found, the region server returns the corresponding value; otherwise, it looks up the key in various checkpoints, starting from the most recent one, and returns the first matching value. Periodically, to minimize the storage overheads and the GET latency, the region server performs *compaction* by reading a number of contiguous checkpoints and merging them into a single checkpoint.

ZooKeeper ZooKeeper [54] is a replicated coordination service. It is used by HBase to ensure that each key-range is assigned to at most one region server.

3.3 The design of Salus

The architecture of Salus, as Figure 3.1 shows, bears considerable resemblance to that of HBase. Like HBase, Salus uses HDFS as its reliable and scalable storage layer, partitions key ranges within a table in distinct regions for load balancing, and supports the abstraction of a region server responsible for handling requests for the keys within a region. As in HBase, blocks are mapped to their region server through a Master node, leases are managed using ZooKeeper, and Salus clients need to install a *block driver* to access the storage system, not unlike the client library used for the same purpose in HBase. These similarities are intentional: they aim to retain in Salus the ability to scale to thousands of nodes and tens of thousands of disks that has secured HBase's success. Indeed, one of the main challenges in designing Salus was to achieve its robustness goals (strict ordering guarantees for write operations across multiple disks, end-to-end correctness guarantees for read operations, strong availability and durability guarantees despite arbitrary failures) without perturbing

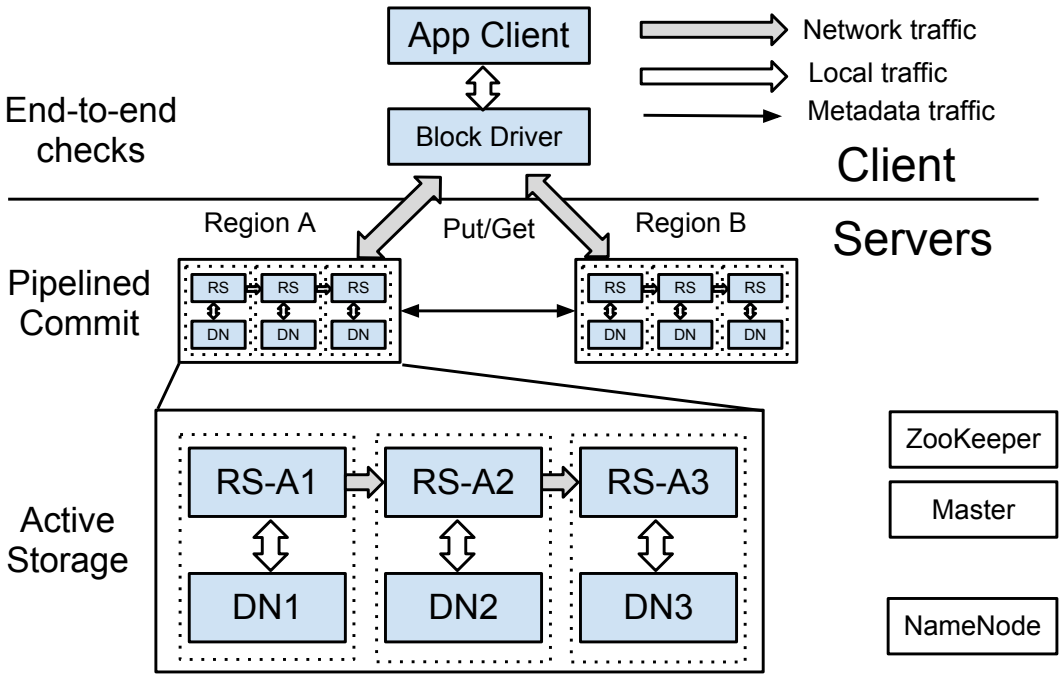

Figure 3.1: The architecture of Salus. Salus differs from HBase in three key ways. First, Salus' block driver performs end-to-end checks to validate each GET reply. Second, Salus performs pipelined commit across different key regions to ensure ordered commit. Third, Salus replicates region servers via active storage to eliminate spurious state updates. For efficiency, Salus tries to co-locate the replicated region servers with the replicated DataNodes (DNs).

the scalability of the original HBase design. With this in mind, we have designed Salus so that, whenever possible, it buttresses architectural features it inherits from HBase—and does so scalably. So, the core of Salus' active storage is a three-way replicated region server (RRS), which upgrades the original HBase region server abstraction to guarantee safety despite up to two arbitrary server failures. Similarly, Salus' end-to-end verification is performed within the familiar architectural feature of the block driver, though upgraded to support Salus' scalable verification mechanisms.

Figure 3.1 also helps describe the role played by our novel techniques (pipelined commit, scalable end-to-end verification, and active storage) in the operation of Salus.

Every client request in Salus is mediated by the block driver, which exports a virtual disk interface by converting the application's API calls into Salus GET and PUT requests. The block driver, as we saw, is the component in charge of performing Salus' scalable end-to-end verification (see Section 3.3.3): for PUT requests it generates the appropriate metadata, while for GET requests it uses the request's metadata to check whether the data returned to the client is consistent.

To issue a request, the client (or rather, its block driver) contacts the Master, which identifies the RRS responsible for servicing the block that the client wants to access. The client caches this information for future use and forwards the request to that RRS. The first responsibility of the RRS is to ensure that the request commits in the order specified by the client. This is where the pipelined commit protocol becomes important: as we will see in more detail in Section 3.3.1, the protocol requires only minimal coordination to enforce dependencies among requests assigned to distinct RRSs. If the request is a PUT, the RRS also needs to ensure that the data associated with the request is made persistent, despite the possibility of individual region servers suffering commission failures. This is the role of active storage (see

Section 3.3.2): the responsibility of processing PUT requests is no longer assigned to a single region server, but is instead conditioned on the set of region servers in the RRS achieving unanimous consent on the update to be performed. Thanks to Salus' end-to-end verification guarantees, GET requests can instead be safely carried out by a single region server (with obvious performance benefits), without running the risk that the client sees incorrect or stale data.

3.3.1 Pipelined commit

The goal of the pipelined commit protocol is to allow clients to issue requests to multiple regions concurrently, while preserving the ordering specified by the client (ordered-commit). In the presence of even simple crash failures, however, enforcing the ordered-commit property can be challenging.

Consider, for example, a client that, after mounting a volume V that spans regions 1 and 2, issues a PUT u_1 for a block mapped to region 1 and then, without waiting for the PUT to complete, issues a barrier PUT u_2 for a block mapped to region 2. Untimely crashes of the client and of the region server for region 1 may lead to u_1 being lost even as u_2 commits.[3] Volume V would now violate the standard disk semantics; further, V would be left in an invalid state that can potentially cause severe data loss [28, 87].

A simple way to avoid such inconsistencies would be to allow clients to issue one request (or one batch of requests) at a time, but, as we show in Section 3.4.2, performance would suffer significantly. Instead, we would like to achieve the good performance that comes with issuing multiple outstanding requests, without compromising the ordered-commit property.

The basic insight of Salus' solution is that processing data out of order, as a result of parallel processing and failures, is fine as long as the clients can only observe

[3]For simplicity, in this example and throughout this section we consider a single logical region server to be at work in each region. In practice, in Salus this abstraction is implemented by a RRS.

the data in the correct order, even if such out-of-order data is made persistent on disks: such prematurely-persisted data can be garbage-collected without being noticed by clients. To ensure clients can only observe data in the correct order, we once again apply the idea of separating data from metadata: Salus parallelizes the bulk of the processing (such as cryptographic checks and disk-writes) required to handle each request, while ensuring that requests commit in order by transferring a small quantity of metadata sequentially.

Salus ensures ordered-commit by exploiting the sequence number that clients assign to each request. Region servers use these sequence numbers to guarantee that a request does not commit (become visible to clients) unless all the previous requests are persistent on disks, and thus are guaranteed to eventually commit. Similarly, during recovery, these sequence numbers are used to ensure that a consistent prefix of issued requests are recovered (Section 3.3.4).

Salus' approach to ensure ordered-commit for GETs is simple. Like other systems before it [19], Salus neither assigns new sequence numbers to GETs, nor logs GETs to stable storage. Instead, to prevent returning stale values, a GET request to a region server simply carries a `prevNum` field indicating the sequence number of the last PUT executed on that region: region servers do not execute a GET until they have committed a PUT with the `prevNum` sequence number. Conversely, to prevent the value of a block from being overwritten by a later PUT, clients block PUT requests to a block that has outstanding GET requests.[4]

Salus' pipelined commit protocol for PUTs is illustrated in Figure 3.2. The client, as in HBase, issues requests in batches. Unlike HBase, Salus allows each client to issue multiple outstanding batches. Each batch is committed using a protocol

[4]This requirement has minimal impact on performance, as such PUT requests are rare in practice.

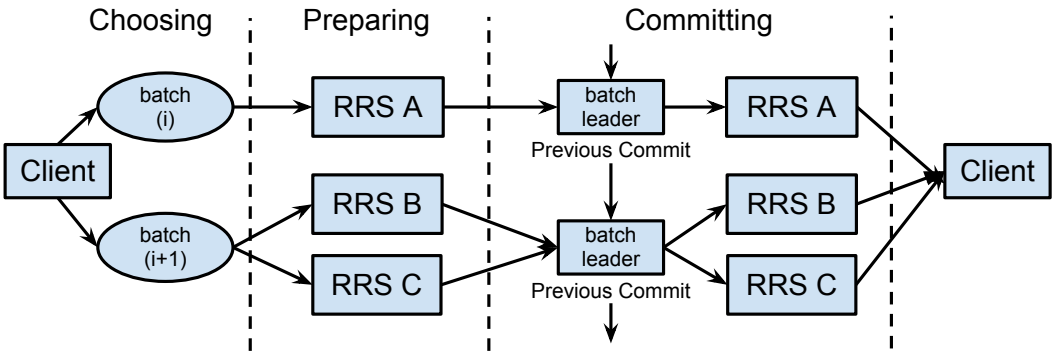

Figure 3.2: Pipelined commit (each batch leader is actually replicated to tolerate arbitrary faults.)

consisting of the phases described below.[5]

PC1. *Choosing the batch leader and participants.* To process a batch, a client divides its PUTs into various sub-batches, one per region server. Just like a GET request, a PUT request to a region also includes a `prevNum` field to identify the last PUT request issued to that region. The client identifies one region server as *batch leader* for the batch and sends each sub-batch to the appropriate region server along with the batch leader's identity. The client sends the sequence numbers of all requests in the batch to the batch leader, along with the identity of the leader of the *previous* batch.

PC2. *Preparing.* A region server preprocesses the PUTs in its sub-batch by *validating* each request, i.e. by checking whether the request is signed and by using the `prevNum` field to verify it is the next request that the region server should process. If validation succeeds for all requests in the sub-batch, the region server logs the requests (which are now *prepared*) and sends its YES vote to

[5]This protocol looks similar to a two-phase commit (2PC) protocol [45, 65], but it is designed for a different purpose: the objective of 2PC is to ensure that a batch or transaction is either fully executed or not, while that of pipelined commit is to ensure that if a batch is committed, all batches that preceded it will eventually be committed.

the batch's leader; otherwise, the region server votes NO.

PC3. *Deciding.* The batch leader can decide COMMIT only if it receives a YES vote for all the PUTs in its batch and a COMMIT-CONFIRMATION from the leader of the previous batch; otherwise, it decides ABORT. Either way, the leader notifies the participants of its decision. Upon receiving COMMIT for a request, a region server updates its memory state (memstore), sends a PUT_SUCCESS notification to the client, and asynchronously marks the request as committed on persistent storage. On receiving ABORT, a region server discards the state associated with that PUT and sends the client a PUT_FAILURE message. Abort can happen when the client is faulty (e.g. corrupted memory), the network is faulty (e.g. corrupted messages), or a logical region server is faulty. As shown in the next section, each logical region server is actually a replicated group and Salus can mask up to two failures inside a group. If more than two failures occur as a result of a severe problem (e.g. configuration error, bugs in the code, etc), then the logical region server may behave unexpectedly and the batch leader may decide to abort. When a client receives the PUT_FAILURE message as the result of an aborted request, it retries the corresponding request for a certain number of times to see whether the problem is transient (e.g. an occasional bit blip in the network), and if not, the client must notify the administrator for further assistance.

Notice that all disk writes—both within a batch and across batches—can proceed in parallel and that the voting and commit phases for a given batch can be similarly parallelized. Different region servers receive and log the PUT and COMMIT asynchronously. The only serialization point is the passing of COMMIT-CONFIRMATION from the leader of a batch to the leader of the next batch.

Despite its parallelism, the protocol ensures that requests commit in the order specified by the client. The presence of COMMIT in any correct region server's

log implies that all preceding PUTs in this batch must have prepared. Furthermore, all requests in preceding batches must have also prepared. Our recovery protocol (Section 3.3.4) ensures that all prepared PUTs eventually commit without violating ordered-commit.

The pipelined commit protocol enforces ordered-commit assuming the abstraction of (logical) region servers that are correct. It is the *active storage* protocol (Section 3.3.2) that, from physical region servers that *can* lose committed data and suffer arbitrary failures, provides this abstraction to the pipelined commit protocol.

3.3.2 Active storage

Active storage provides the abstraction of a region server that does not experience arbitrary failures or lose data. Salus uses active storage to ensure that the data remains available and durable despite arbitrary failures in the storage system by addressing two key limitations of existing scalable storage systems: first, they replicate data at the storage layer (e.g. HDFS) but leave the computation layer (e.g. HBase) unreplicated. As a result, the computation layer that processes clients' requests represents a single point of failure in an otherwise robust system. For example, a bug in computing the checksum of data or a corruption of the memory of a region server can lead to data loss and data unavailability in systems like HBase. Second, the storage layer usually replicates data with three-way synchronous primary-backup protocols, requiring minimal cost but leaving the system vulnerable to uncommon errors.

To prevent a region server from being a single point of failure, the design of Salus uses a new *active storage* architecture that embodies a simple principle: all changes to persistent state should happen with the consent of a quorum of nodes. Salus uses these *compute quorums* to protect its data from faults in its region servers: when a compute quorum sends data to the storage layer, all nodes in the quorum

must unanimously agree on a certificate, which carries information to validate data, and send the certificate to the storage layer so that the storage nodes can verify data integrity.

To tolerate arbitrary failures in the storage layer, the active storage architecture incorporates the central idea of Gnothi: to tolerate two failures, it replicates data on just three nodes, but replicates metadata (certificate) on five nodes. By using separate policies when replicating data and metadata, Salus can identify the correct version of data despite arbitrary failures with minimal replication costs.

Using active storage, Salus can provide strong availability and durability guarantees: a data block will remain available and durable as long as there are no more than two failures occurring on the physical machines that are processing the block.[6] These guarantees hold irrespective of whether the nodes fail by crashing (omission) or by corrupting their disk, memory, or logical state (commission).

Replication typically incurs network and storage overheads. Salus uses three key ideas—(1) moving computation to data, (2) using unanimous consent quorums, and (3) separating data from metadata—to ensure that active storage does not incur significant network or storage overhead compared to existing approaches. Perhaps surprisingly, in addition to improving fault-resilience, active storage improves performance, by trading relatively cheap CPU cycles for expensive network bandwidth.

Moving computation to data to minimize network usage

Salus implements active storage by blurring the boundaries between the storage layer and the computation layer. Existing storage systems [12, 23, 27] require a designated primary DataNode to mediate updates. In contrast, Salus modifies the storage system API to permit region servers to directly update any replica of a

[6] A data block is processed by three region servers, three DataNodes, and two witness nodes. Salus allows a region server to be colocated with a DataNode or a witness node on the same physical machine. Salus can tolerate two physical machine failures, which means that two pairs of colocated processes can be faulty, but at least one region server/DataNode pair is correct.

block. Using this modified interface, Salus can efficiently implement active storage by colocating a computation node (region server) with the storage node (DataNode) that it needs to access.

Active storage thus reduces bandwidth utilization in exchange for additional CPU usage (Section 3.4.2)—an attractive trade-off for bandwidth constrained datacenters. In particular, because a region server can now update the colocated DataNode without requiring the network, the bandwidth overheads of flushing (Section 3.2) and compaction (Section 3.2) in HBase are avoided.

We have implemented active storage in HBase by changing the NameNode API for allocating blocks. As in HBase, to create a block a region server sends a request to the NameNode, which responds with the new block's location; but where the HBase NameNode makes its placement decisions in splendid solitude, in Salus the request to the NameNode includes a list of preferred DataNodes as a *location-hint*. The hint biases the NameNode toward assigning the new block to DataNodes hosted on the same machines that also host the region servers that will access the block. The NameNode follows the hint unless doing so violates its load-balancing policies.

Loosely coupling in this way the region servers and DataNodes of a block yields Salus significant network bandwidth savings (Section 3.4.2). Why then not go all the way—eliminate the HDFS layer and have each region server store its state on its local file system? The reason is that maintaining flexibility in block placement is crucial to the robustness of Salus: our design allows the NameNode to continue to load balance and re-replicate blocks as needed, and makes it easy for a recovering region server to read state from any DataNode that stores it, not just its own disk.

Using unanimous consent to reduce replication overheads

To prevent a failed region server from propagating errors to DataNodes, Salus asks the RRS to agree on the PUTs to be stored. BFT agreement, the traditional solution to this problem, requires $3f + 1$ replicas and thus is usually perceived as too expensive in practice. Salus reduces this threshold to its minimal value of $f + 1$ by leveraging two observations. First, one key challenge in BFT agreement—a faulty leader proposes different execution orders to different participating servers—does not exist in Salus, because (1) Salus relies on the single client allowed to perform writes on a given volume to specify the order in which PUTs should execute[7] and (2) each region server can validate a client's requests independently. This allows Salus to reduce by f the number of replicas it needs to guarantee safety [30]. Second, region servers do not have any persistent state: their state is stored on the DataNodes. Therefore, a region server failed on one machine can be recovered on another machine by reading data from the relevant DataNodes. This allows Salus to further reduce its replication cost by f in the failure free case, down to $f + 1$ by sending requests to only a minimal preferred quorum [33] and failing over to a different quorum if the preferred quorum fails to make progress.

Taking advantage of these two observations, Salus uses unanimous-consent quorums for PUTs: the replicated region servers check clients' requests independently, reach unanimous consent for any operation that updates the state, and generate a certificate proving the legitimacy of the update.

This approach provides safety guarantee despite two commission failures because an update must be agreed by three region servers, each of whom can validate requests independently. However, the failure of any of the replicated region servers can prevent unanimous consent. To ensure liveness, Salus replaces any RRS that is not making adequate progress with a new set of region servers, which read all state

[7] Of course the client can also be faulty, but no guarantee can be achieved for the user anyway if an unreplicated client is faulty.

committed by the previous region server quorum from the DataNodes and resume processing requests. This fail-over protocol is a slight variation of the one already present in HBase to handle failures of unreplicated region servers. If a client detects a problem with a RRS, it sends a *RRS-replacement request* to the Master, which first attempts to get all the nodes of the existing RRS to relinquish their leases; if that fails, the Master coordinates with ZooKeeper to prevent lease renewal. Once the previous RRS is known to be disabled, the Master appoints a new RRS. Then Salus performs the recovery protocol as described in Section 3.3.4.

Separating data from metadata to reduce storage overheads

Three DataNodes are not enough to tolerate two commission failures in an asynchronous environment—during failure recovery, if two nodes are not responding in time and the remaining available node provides a valid version of the data, it's impossible for the system to decide the correct way to proceed. On the one hand, the valid data may be a stale version provided by a faulty node, so accepting it may cause the system to lose a suffix of data. On the other hand, waiting for other nodes to respond is unacceptable, since those nodes may have failed permanently. Indeed, it has been proved that we need at least five execution replicas to tolerate two commission failures in an asynchronous environment [109].

The lesson we learned from Gnothi, however, is that it is not necessary for all these five replicas to store all data: in Salus, only three DataNodes store data and certificates while the remaining two serve as witnesses, storing only the certificates. As in Gnothi, during recovery (see Section 3.3.4), the fully replicated certificates help the system identify the correct data.

Active storage protocol

To provide the other components of Salus with the abstraction of a correct region server, region servers within a RRS are organized in a chain. In response to a client's PUT request or to attend a periodic task (such as flushing and compaction), the *primary* region server (the first replica in the chain) forwards the request (the PUT request or a special flush or compact request) to all region servers in the chain. After executing the request, the region servers in the RRS coordinate to create a *certificate* attesting that all replicas executed the request in the same order and obtained identical responses. The components of Salus (such as client, NameNode, and Master) that use active storage to make data persistent require all messages from a RRS to carry such a certificate: this guarantees no spurious changes to persistent data as long as at least one region server and its corresponding DataNode do not experience a commission failure.

Figure 3.3 shows how active storage refines the pipelined commit protocol for PUT requests.

- Step ①. The PUT issued by a client is received by the primary region server as part of a sub-batch. The primary region server needs to ensure the safety property that the system processes correct requests in the correct order. To make this possible, the PUT request carries additional metadata including the client ID, the checksum of the data, the sequence number and the `prevNum` assigned by the client (see Section 3.3.1), and finally the client's signature of all information. Upon receiving a PUT, the primary region server validates it independently by checking 1) the signature matches the client ID; 2) the checksum matches the data; and 3) the `prevNum` is the same as the sequence number of the previous PUT received by this region. If the validation succeeds, the region server forwards it down the chain of replicas and each replica will perform the same validation independently. With such design, it is not possible

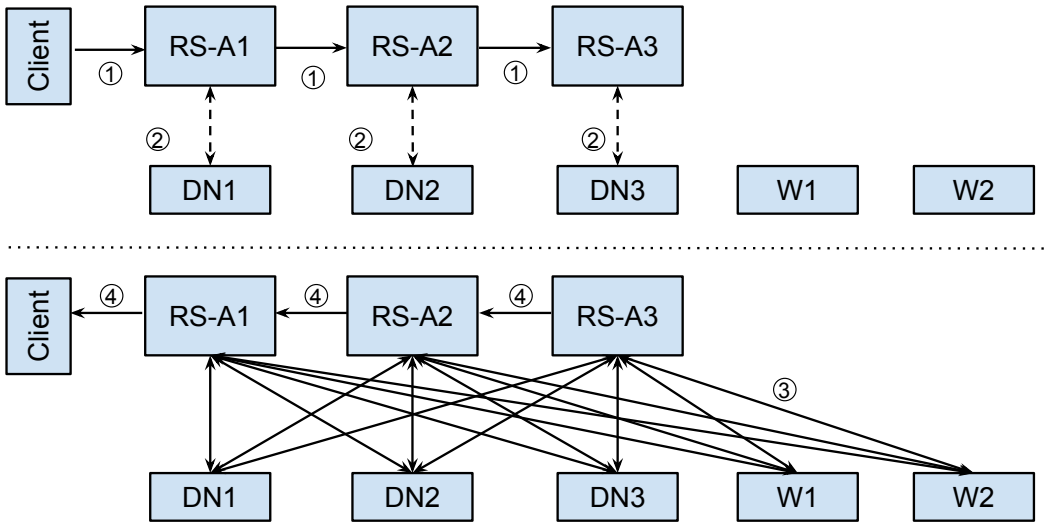

Figure 3.3: Steps to process a PUT request in Salus using active storage. W1 and W2 are witness nodes. Note that steps 1 to 4 are executed in the same group of nodes, but for clarity, we separate these steps in two figures.

for a region server to accept a corrupted PUT or miss a PUT from the client, as long as the client does not experience commission failures.

- Step ②. Each region server writes the PUT requests to the corresponding DataNode of its log file, assigned by the NameNode, and waits for the DataNode to confirm that data is persistent. Note that at this point, no unanimous consent is reached yet and the DataNode cannot validate the correctness of data, except that data is from the expected region server; the DataNode just logs the data optimistically, hoping that unanimous consent will be reached later. And if not, the data will be garbage-collected in the recovery protocol (see Section 3.3.4). One may wonder why the DataNode cannot verify the data directly by using the mechanism described in step ①. This is because a DataNode, as a component HDFS, is designed to process append-only logs, and it does not understand the semantics of the PUT/GET interface of HBase;

it is the task of the region server to convert a PUT request into an append operation on a DataNode: during this procedure, a region server may split its log into multiple files and HDFS may also split a file into multiple blocks. In addition, both the HBase layer and the HDFS layer are adding additional information to the log, such as timestamp. As a result, what is received by a DataNode is a mixture of clients' PUTs, possibly broken into pieces, and the additional information from HBase and HDFS. In this case, it is impossible for the DataNode to verify PUTs directly.

- Step ③. For the data it just writes, each region server generates a TENTATIVE_LOG_ENTRY certificate, which contains the file ID (assigned by NameNode) of the log, the starting position of the data in the file, the length of the data, the checksum of the data, and a signature. The certificate not only validates data content, but also validates data location, which is critical because the order of PUTs in log files can affect recovery. Each region server then broadcasts its TENTATIVE_LOG_ENTRY certificate to three DataNodes and two witness nodes. If a DataNode or witness node receives three correctly signed and matching TENTATIVE_LOG_ENTRY certificates, it combines them into a LOG_ENTRY certificate and stores it. Note that at this time a DataNode does not even need to verify data with the LOG_ENTRY certificate: such verification can be delayed until recovery. A LOG_ENTRY certificate serves two purposes: first, it proves the correctness of the data, since it has been agreed by three region servers, of which one is correct. Such proof allows Salus to validate data during recovery. Second, it also implicitly serves as a proof that the data is already stored on at least one correct DataNode, because at least a region server and DataNode pair is correct, and the correct region server only proposes its TENTATIVE_LOG_ENTRY certificate after it writes data to the correct DataNode. As we will see, this property is important in recovery. After

storing the LOG_ENTRY certificate, a DataNode or witness node replies to all region servers, and a region server considers the procedure as complete if it receives five replies. Any failure during the procedure, of course, will block the execution and finally trigger recovery.

- Step ④. Each region server independently sends its vote (YES if Step 3 completes successfully and NO otherwise) to the leader of the batch to which the PUT belongs and, if it voted YES, waits for the decision. On receiving COMMIT, the region servers mark the request as committed, update their in-memory state and generate a TENTATIVE_PUT_SUCCESS certificate; on receiving ABORT the region servers generate instead a TENTATIVE_PUT_FAILED certificate. A client considers a request as complete if it receives a PUT_SUCCESS certificate, which consists of three matching TENTATIVE_PUT_SUCCESS certificates from three region servers. Otherwise, the block driver triggers recovery if there is a proof of misbehavior by any of the servers (e.g. no unanimous consent) or retries the request if there is no such proof (e.g. timeout): the block driver also triggers recovery if it retries a request several times but the servers still cannot make progress. The PUT_SUCCESS certificate proves that at least one correct region server believes that a correct LOG_ENTRY certificate has been successfully generated and stored, which indicates that all the correct DataNodes and witness nodes must have already stored the LOG_ENTRY certificate.

Similar changes are also required to leverage active storage in flushing and compaction. Unlike PUTs, these operations are initiated by the primary region server: the other region servers use predefined deterministic criteria, such as the current size of the memstore, to verify whether the proposed operation should be performed. The writing protocol of flush and compaction, however, can be greatly simplified: the RRSs only need to agree on a hash of the generated file at the end of a flush or compaction, and store the hash on the NameNode if unanimous consent is

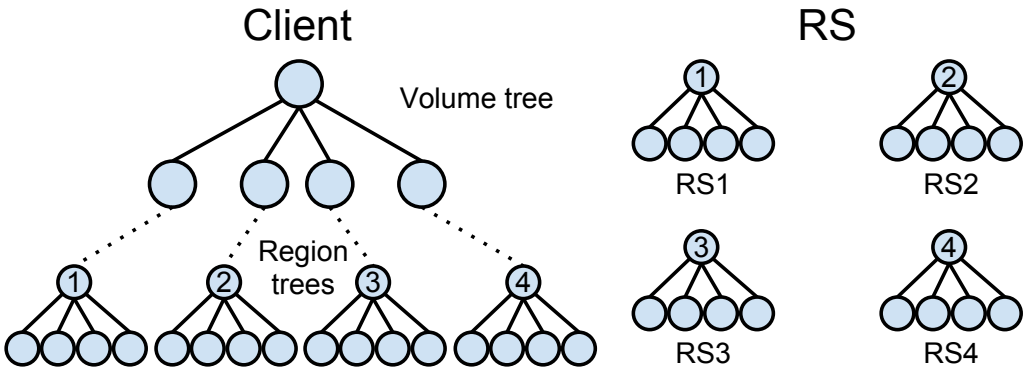

Figure 3.4: Merkle tree structure on client and region servers

reached. If a failure occurs, the Master node can simply delete the incomplete files and instructs a new set of RRSs to retry the flush or compaction: we can do this because data is already stored in the logs and can be recovered, a luxury obviously not offered to log files.

3.3.3 End-to-end verification

Local file systems fail in unpredictable ways [87]. Distributed systems like HBase are even more complex and are therefore more prone to failures. To provide strong correctness guarantees, Salus implements end-to-end checks that (a) ensure that clients access correct and current data and (b) do so without affecting performance: GETs can be processed at a single replica and yet retain the ability to identify whether the returned data is correct and current.

Like many existing systems [3, 43, 69, 73], Salus' mechanism for end-to-end checks leverages Merkle trees [18, 75], another type of metadata, to efficiently verify the integrity of the state whose hash is at the tree's root. Specifically, a client accessing a volume maintains a Merkle tree on the volume's blocks, called *volume tree*, that is updated on every PUT and verified on every GET.

For fast recovery, Salus distributes a copy of the volume tree across the region servers that host the volume so that, after a crash, a client can rebuild its volume tree by contacting the region servers responsible for the regions in that volume. Replicating the volume tree at the region servers also allows a client, if it so chooses, to only store a subset of its volume tree during normal operation, fetching on demand what it needs from the region servers serving its volume.

Since a volume can span multiple region servers, for scalability and load-balancing each region server only stores and validates a *region tree* for the regions that it hosts. The region tree is a sub-tree of the volume tree corresponding to the blocks in a given region. In addition, to enable the client to recover the volume tree, each region server also stores the hash for the root of the full volume tree generated by the most recent update to a block in the region for which the server is responsible, together with the sequence number of the PUT request that produced it.

Figure 3.4 shows a volume tree and its region trees. The client stores the top levels of the volume tree that are not included in any region tree so that it can easily fetch the desired region tree on demand. A client can also cache recently used region trees for faster access.

To process a GET request for a block, the client sends the request to any of the region servers hosting that block. As shown in Figure 3.5, the response includes a subset of the region tree (all non-cached nodes on the path from the target to the root of the corresponding region tree, together with their siblings) sufficient for the client to validate the response using the locally stored volume tree. If the check fails (because of a commission failure) or if the client times out (because of an omission failure), the client retries the GET using another region server. If the GET fails at all region servers, the client contacts the Master triggering the recovery protocol (Section 3.3.4). To process a PUT, the client updates its volume tree and sends the weakly-signed root hash of its updated volume tree along with the PUT request to

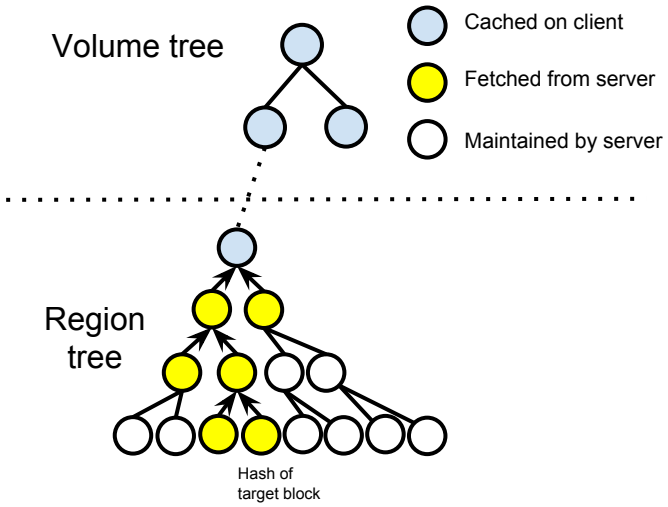

Figure 3.5: Verifying a response with the Merkle tree: the client caches the whole volume tree and part of the region tree and fetches a subset of the region tree to verify the response.

the RRS. Attaching the root hash of the volume tree to each PUT request enables clients to ensure that, despite commission failures, they will be able to mount and access a consistent volume.

A client's protocol to mount a volume after losing the volume tree is simple. The client begins by fetching the region trees, the root hashes, and the corresponding sequence numbers from the various RRSs. Before responding to a client's fetch request, a RRS commits any prepared PUTs pending to be committed using the *commit-recovery* phase of the recovery protocol (Section 3.3.4). Using the sequence numbers received from all the RRSs, the client identifies the most recent root hash and compares it with the root hash of the volume tree constructed by combining the various region trees. The check should always succeed as long as there are no more than two failures.

3.3.4 Recovery

The recovery protocol ensures that, despite commission or permanent omission failures in up to two physical servers, Salus continues to provide the abstraction of a virtual disk with *standard disk semantics*.

To achieve this goal, Salus' recovery protocol collects the longest available prefix P_C of prepared PUT requests that satisfy the ordered-commit property. Recall from Section 3.3.2 that every PUT for which the client received a PUT_SUCCESS certificate must appear in the log of at least one correct replica in the region that processed that PUT. Hence, P_C will contain all PUT requests for which the client received a PUT_SUCCESS certificate, thus guaranteeing standard disk semantics.

Specifically, recovery must address two key issues.

Resolving log discrepancies Because of omission or commission failures, different DataNodes within the same RRS may store different logs. A prepared PUT, for example, may have been made persistent at one DataNode, but not at another.

Identifying committable requests Because COMMIT decisions are logged asynchronously, some PUTs for which a client received PUT_SUCCESS may not be marked as committed in the logs. It is possible, for example, that a later PUT be logged as committed when an earlier one is not; or that a suffix of PUTs for which the client has received a PUT_SUCCESS be not logged as committed.

One major challenge in addressing these issues is that, while P_C is defined on a global *volume log*, Salus does not actually store any such log: instead, for efficiency, each region keeps its own separate *region log*. Hence, after retrieving its region log, a recovering region server needs to cooperate with other region servers to determine whether the recovered region log is correct and whether the PUTs it stores can be committed.

Figure 3.6 describes the protocol that Salus uses to recover faulty DataNodes and region servers. The first two phases describe the recovery of individual region

```
1  foreach failed-region i
2   remapRegion(i)
3  end
4  foreach failed-region i
5    region_logs[i] ← recoverRegionLog(region i)
6  end
7  LCP ← identifyLCP(region_logs)
8  rebuildVolume(LCP)
```

Figure 3.6: Pseudocode for the recovery protocol.

logs, while the last two phases describe how the RRSs coordinate to identify commitable requests. Note that just as a local disk is unavailable during a post-failure consistency check, so is a volume unavailable to its client during recovery.

1. Remap (`remapRegion`). As in HBase, when a RRS crashes or is reported by the client as non-responsive, the Master swaps out the servers in that RRS and assigns its regions to one or more replacement RRSs.

2. Recover region log (`recoverRegionLog`). To recover all prepared PUTs of a failed region, the new region servers need to replay the region's log. As shown in the active storage protocol, Salus' log is organized as a chain of entries, each containing a LOG_ENTRY certificate and its corresponding data, which may contain a number of PUTs and some additional information for HBase and HDFS. The recovery protocol iterates these entries in three steps: ① the primary (first) region server in the new RRS asks the corresponding three DataNodes and two witness nodes to provide the LOG_ENTRY certificate at a specific position (which starts from zero and is updated if an entry is successfully read) and waits for three valid responses. A response is valid if it is a valid LOG_ENTRY certificate with proper signatures, file ID, and position, or a message of "No next entry" properly signed. The primary region server is guaranteed to get three valid responses, since there are at least three correct nodes. ②If at least one of the responses is a valid LOG_ENTRY certificate, the primary region server tries to read the corresponding data from DataNodes. This read is guaranteed to succeed because the LOG_ENTRY certificate proves that the data has already been

stored on at least one correct DataNode; if all three responses are "No next entry", then the region server considers the log replaying as completed. ③ In either case, the primary region server forwards its decision together with the proof—the data with LOG_ENTRY certificate or three properly signed "No next entry"s—to the other region servers so that they can verify the decision. This procedure works exactly as does unanimous consent in the active storage protocol. If the new region servers reach unanimous consent, they write the data to a new file and garbage-collect the old ones, as in HBase. Finally, if a LOG_ENTRY certificate is obtained in ②, meaning that the log may have not been fully replayed yet, the protocol loops back to ①. Otherwise, the protocol completes.

The protocol's liveness is already briefly discussed above. We now briefly prove the following safety property: if a PUT request completes successfully for a client, then this PUT request is replayed in the correct order during the recovery protocol described above. First, we prove that if a PUT completed successfully, it must be replayed during recovery: recall that a client considers a PUT as complete only after it receives a PUT_SUCCESS certificate, which proves that the corresponding LOG_ENTRY certificate must have been stored on all correct DataNodes and witness nodes. Therefore, in the recovery protocol, the primary region server must receive at least one LOG_ENTRY certificate in ①, because it waits for three replies and at least one of them must be from a correct DataNode or witness node. And the correct LOG_ENTRY certificate proves that the data must be persistent on at least one correct DataNode, and the region server can iterate all DataNodes to find the correct data. Second, the order of replaying is guaranteed by the design of LOG_ENTRY certificate itself because the certificate contains the position information and such information is checked in ①. Note that a log may be split into multiple actual files on HDFS, and the order of those files is determined by the timestamps in the names of these files, which are stored on the trusted NameNode.

3. Identify the longest commitable prefix (LCP) of the volume log (`identifyLCP`). If the client is available, Salus can determine the LCP and the root of the corresponding volume tree simply by asking the client. Otherwise, all RRSs must coordinate to identify the longest prefix of the volume log that contains either committed or prepared PUTs (i.e. PUTs whose data has been made persistent in at least one correct DataNode). Since Salus keeps no physical volume log, the RRSs use ZooKeeper as a means of coordination, as follows. The Master asks each RRS to report its maximum committed sequence number as well as its list of prepared sequence numbers by writing the requested information to a known file in ZooKeeper. Upon learning from Zookeeper that the file is complete (i.e. all RRSs have responded),[8] each RRS uses the file to identify the longest prefix of committed and prepared PUTs in the volume log. Finally, the sequence number of the last PUT in the LCP and the attached Merkle tree root are written to ZooKeeper.

4. Rebuild volume state (`rebuildVolume`). This phase ensures that all PUTs in the LCP are committed and available. The first half of the task is simple: if a PUT in the LCP is prepared, then the corresponding region server marks it as committed. With respect to availability, Salus makes sure that all PUTs in the LCP are available, in order to reconstruct the volume consistently. To that end, the Master asks the RRSs to replay their log and rebuild their region trees; it then uses the same checks used by the client in the mount protocol (Section 3.3.3) to determine whether the current root of the volume tree matches the one stored in ZooKeeper during Phase 3. The check should always succeed as long as there are no more than two failures and it mainly serves as a sanity check.

[8] If some RRS are unavailable during this phase, recovery starts again from Phase 1, replacing the unavailable servers.

3.4 Evaluation

We have implemented Salus by modifying HBase [12] and HDFS [95] to add pipelined commit, active storage, and end-to-end checks. Our current implementation lags behind our design in two ways. First, our prototype supports unanimous consent between HBase and HDFS but not between HBase and ZooKeeper. Second, while our design calls for a BFT-replicated Master, NameNode, and ZooKeeper, our prototype does not yet incorporate these features. We intend to use UpRight [31] to replicate NameNode, ZooKeeper, and Master. Implementing these features would require a replicated state machine (RSM) to communicate with another RSM, which looks simple but actually is not, and we are currently working on this problem in another project [56].

Our evaluation tries to answer two basic questions. First, does Salus provide the expected guarantees despite a wide range of failures? Second, given its stronger guarantees, is Salus' performance competitive with HBase? Figure 3.7 summarizes the main results.

In the originally published version [106], Salus did not incorporate witness nodes in the active storage protocol and a combination of concurrent errors could cause it to lose a suffix of data [106]. The active storage protocol presented in this dissertation eliminates this problem and guarantees that no data will be lost. However, since we don't have the resources to rerun all our original experiments, most of the results presented in this section are still obtained with the old protocol. We add an experiment at the end of the section to measure the additional overhead of the new protocol.

3.4.1 Robustness

In this section, we evaluate Salus' robustness, which includes guaranteeing freshness for read operations and liveness and ordered-commit for all operations.

Salus ensures *freshness, ordered-commit, and liveness* when there are no more than 2 failures within any RRS and the corresponding DataNodes.	§3.4.1
Salus achieves comparable or better single-client throughput compared to HBase with slightly increased latency.	§3.4.2
Salus' active replication can reduce network usage by 55% and increase aggregate throughput by 74% for sequential write workload compared to HBase. Salus can achieve similar aggregate read throughput compared to HBase.	§3.4.2
Salus' overhead over HBase does not grow with the scale of the system.	§3.4.2

Figure 3.7: Summary of main results.

Salus is designed to ensure these properties as long as there are no more than two failures in the region servers within an RRS and their corresponding DataNodes, and fewer than a third of the nodes in the implementation of each of UpRight NameNode, UpRight ZooKeeper, and UpRight Master nodes are incorrect; however, since we have not yet integrated in Salus UpRight versions of NameNode, ZooKeeper, and Master, we only evaluate Salus' robustness when DataNode or region server fails.

We test our implementation via fault injection. We introduce failures and then observe what happens when we attempt to access the storage. For reference, we compare Salus with HBase (which replicates stored data across DataNodes but does not support pipelined commit, active storage, or end-to-end checks).

In particular, we inject faults into clients to force them to crash and restart. We inject faults into DataNodes to force them either to temporarily or permanently crash or to corrupt block data. We cause data corruption in both log files and checkpoint files. We inject faults into region servers to force them to either 1) crash; 2) corrupt data in memory; 3) write corrupted data to HDFS; 4) refuse to process requests or forward requests out of order; or 5) ask the NameNode to delete files. Once again, we cause corruption in both log files and checkpoint files. Note that data on region servers is not protected by checksums. Figure 3.1 summarizes our results.

Affected nodes	Faults	HBase		Salus	
		GET	PUT	GET	PUT
Client	Crash and restart	Fresh	Not ordered	Fresh	Ordered
DataNode	One or two permanent crashes	Fresh	Ordered	Fresh	Ordered
	Corruption of one or two replicas of log or checkpoint	Fresh	Ordered	Fresh	Ordered
	Three arbitrary failures	Fresh*	Lost	Fresh*	Lost
Region server+DataNode	One (for HBase) or three (for Salus) region server permanent crashes	Fresh	Ordered	Fresh	Ordered
	One (for HBase) or two (for Salus) region server arbitrary failures that potentially affect DataNodes	Corrupted	Lost	Fresh	Ordered
	Three (for Salus) region server arbitrary failures that potentially affect DataNodes	-	-	Fresh*	Lost
Client+Region server+DataNode	Client crashes and restarts, one (for HBase) or two (for Salus) region server arbitrary failures causing the corresponding DataNodes to not receive a suffix of data	Corrupted	Lost	Fresh	Ordered

Table 3.1: Robustness towards failures affecting the region servers within an RRS, and their corresponding DataNodes. (- = not applicable, * = corresponding operations may not be live). Note that a region server failure has the potential to cause the failure of the corresponding DataNode.

First, as expected, when a client crashes and restarts in HBase, a volume's on-disk state can be left in an inconsistent state, because HBase does not guarantee ordered commit. HBase can avoid these inconsistencies by blocking all requests that follow a barrier request until the barrier completes, but this can hurt performance when barriers are frequent (see Section 3.4.2). Second, HBase's replicated DataNodes tolerate crash and *benign* file corruptions that alter the data but don't affect the checksum, which is stored separately. Thus, when considering only DataNode failures, HBase provides the same guarantees as Salus. Third, HBase's unreplicated region server is a single point of failure, vulnerable to commission failures that can violate freshness as well as ordered-commit.

In Salus, end-to-end checks ensure freshness for GET operations in all the scenarios covered in Figure 3.1: a correct client does not accept GET reply unless it can pass the Merkle tree check. Second, pipelined commit ensures the ordered-commit property in all scenarios involving one or two failures, whether of omission or of commission: if a client fails or region servers reorder requests, the out-of-order requests will not be accepted and eventually recovery will be triggered, causing these requests to be discarded. Third, active storage protects liveness failure scenarios involving one or two region server/DataNode pairs: if a client receives an unexpected GET reply, it retries until it obtains the correct data. Furthermore, during recovery, the recovering region servers find the correct log by using the certificates generated by active storage protocol. As expected, ordered-commit and liveness cannot be guaranteed if *all* replicas either permanently fail or experience commission failures.

3.4.2 Performance

Salus' architecture can in principle result in both benefits and overhead when it comes to throughput and latency: on the one hand, pipelined commit allows multiple batches to be processed in parallel and active storage reduces network band-

width consumption. On the other hand, end-to-end checks introduce checksum computations on both clients and servers; pipelined commit requires additional network messages for preparing and committing; and active storage requires additional computation and messages for certificate generation. Compared to the cost of disk-accesses for data, however, we expect these overheads to be modest.

This section quantifies these tradeoffs using benchmarks for both sequential and random access patterns and for both reads and writes. We compare Salus' single-client throughput and latency, aggregate throughput, and network usage to those of HBase. We also include measured numbers from Amazon EBS to put Salus' performance in perspective.

Salus targets clusters of storage nodes with 10 or more disks each. In such an environment, we expect a node's aggregate disk bandwidth to be much larger than its network bandwidth. Unfortunately, we have only three *storage nodes* matching this description, the rest of our *small nodes* have a single disk and a single active 1Gbit/s network connection.

Most of our experiments run on a 15-node cluster of *small nodes* equipped with a 4-core Intel Xeon X3220 2.40GHz CPU, 3GB of memory, and one Western Digital WD2502ABYS 250GB hard drive. In these experiments, we use nine small nodes as region servers and DataNodes, another small node as the Master, ZooKeeper, and NameNode, and up to four small nodes acting as clients. In these experiments, we set the Java heap size to 2GB for the region server and 1GB for the DataNode.

To understand system behavior when disk bandwidth is more plentiful than network bandwidth, we run several experiments using the three storage nodes, each equipped with a 16-core AMD Opteron 4282 3.0GHz CPU, 64GB of memory, and 10 Western Digital WD1003FBYX 1TB hard drives. These storage nodes have 1Gbit/s networks, but the network topology constrains them to share an aggregate

bandwidth of about 1.2Gbit/s.

To measure the scalability of Salus with a large number of machines, we run several experiments on Amazon EC2 [8]. The detailed configuration is shown in the "Scalability" section.

For all experiments, we use a small 4KB block size, which we expect to magnify Salus' overheads compared to larger block sizes. For read workloads, each client formats the volume by writing all blocks and then forcing a flush and compaction before the start of the experiments. For write workloads, since compaction introduces significant overhead in both HBase and Salus and the compaction interval is tunable, we first run these experiments with compaction disabled to measure the maximum throughput; then we run HBase with its default compaction strategy and measure how many bytes it reads for each compaction; finally, we tune Salus' compaction interval so that Salus performs compaction on the same amount of data as HBase.

Single client throughput and latency

We first evaluate the single-client throughput and latency of Salus. Since a single client usually cannot saturate the system, we find that executing requests in a pipeline is beneficial to Salus' throughput. However, the additional overheads of checksum computation and message transfer of Salus increase its latency.

We use the nine small nodes as servers and start a single client to issue sequential and random reads and writes to the system. For the throughput experiment, the client issues requests as fast as it can and performs batching to maximize throughput. In all experiments, we use a batch size of 250 requests, so each batch accesses about 1MB. For the latency experiment, the client issues a single request, waits for it to return, and then waits for 10ms before issuing the next request.

Figure 3.8 shows the single client throughput. For sequential reads, Salus

outperforms the HBase system with a speedup of 2.5. The reasons are that Salus' three region servers increase parallelism for reads and that reads are pipelined to have multiple batches outstanding; the HBase client instead issues only one batch of requests at a time. For random reads, disk seeks are the bottleneck and HBase and Salus have comparable performance.

For sequential writes and random writes, Salus is slower than HBase by 3.5% to 22.8% for its stronger guarantees. For Salus, pipelined execution does not help write throughput as much as it helps sequential reads, since write operations need to be forwared to all three nodes and unlike reads cannot be executed in parallel.

As a sanity check, Figure 3.8 also shows the performance we measured from a small compute instance accessing Amazon's EBS. Because the EBS hardware differs from our testbed hardware, we can only draw limited conclusions, but we note that the Salus prototype achieves a respectable fraction of EBS's sequential read and write bandwidth, and that it modestly outperforms EBS's random read throughput (probably because it is utilizing more disk arms), and that it substantially outperforms EBS's random write throughput (probably because it transforms random writes into sequential ones.)

Figure 3.9 shows the 90th-percentile latency for random reads and writes. In both cases, Salus' latency is within two or three milliseconds of that of HBase, which is reasonable considering Salus' additional work to perform Merkle tree calculations, certificate generation, and network transfer. Note that, in the random write latency experiment, the HBase DataNode does not call *sync* when performing disk writes: that's why its write latency is small. This may be a reasonable design decision when the probability of three simultaneous crashes is small [71]. In this experiment, we also show what happens when adding this call to both HBase and Salus: calling sync adds more than 10ms of latency to both. To be consistent, we do not call *sync* in other throughput experiments.

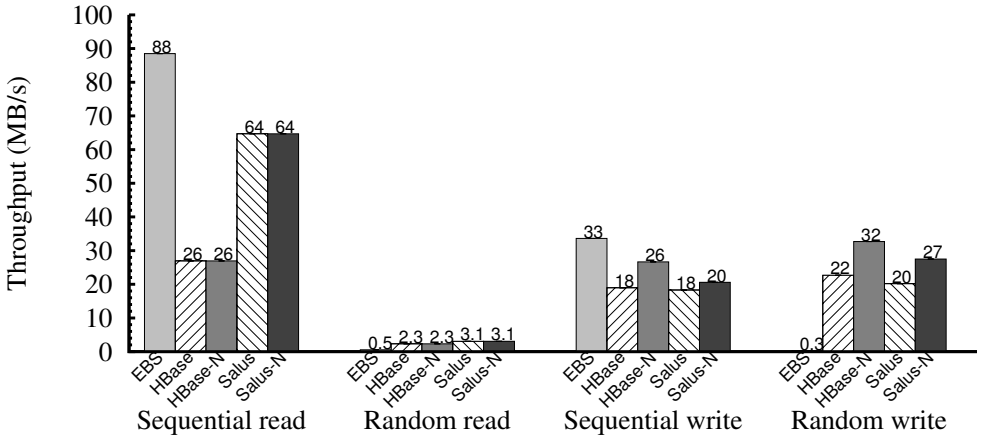

Figure 3.8: Single client throughput on small nodes. HBase-N and Salus-N disable compactions. EBS's numbers are measured on different hardwares and are included for reference.

Again, as a sanity check we note that Salus (and HBase) are reasonably competitive with EBS (though we emphasize again that EBS's underlying hardware is not known to us, so not too much should be read into this experiment.)

Overall, these results show that despite all the extra computation and message transfers to achieve stronger guarantees, Salus' single-client throughput and latency are comparable to those of HBase, because the additional processing Salus requires adds relatively little to the time required to complete disk operations. In an environment in which computational cycles are plentiful, trading off, as Salus does, processing for improved reliability appears to be a reasonable trade-off.

Aggregate throughput/network bandwidth

We then evaluate the aggregate throughput and network usage of Salus. The servers are saturated in these experiments, so pipelined execution does not improve Salus' throughput at all. On the other hand, we find that active replication of region servers, introduced to improve robustness, can reduce network bandwidth and sig-

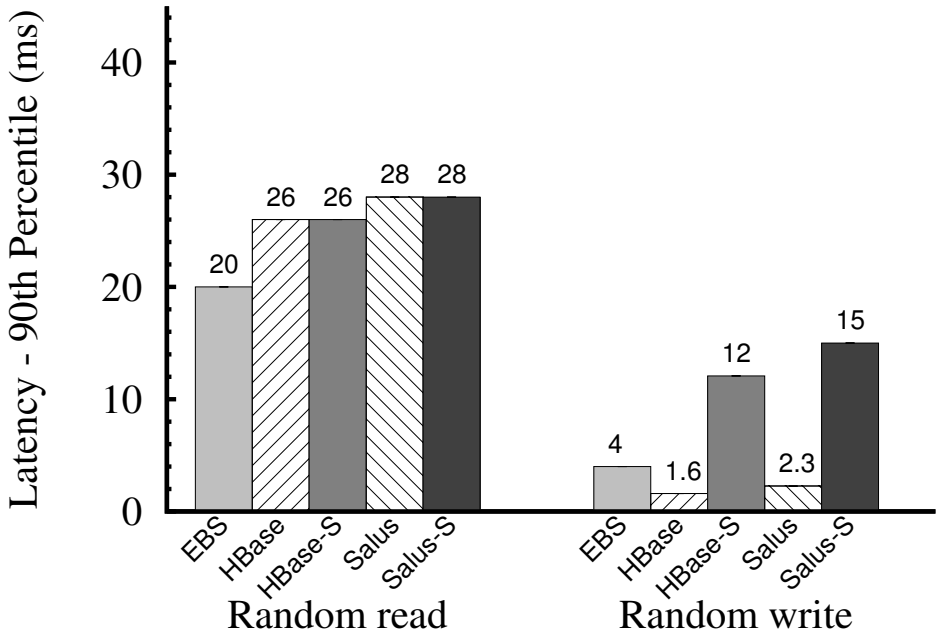

Figure 3.9: Single client latency on small nodes. HBase-S and Salus-S enable sync. EBS's numbers are measured on different hardwares and are included for reference.

nificantly improve performance when the total disk bandwidth exceeds the aggregate network bandwidth.

Figure 3.10 reports experiments on our small-server testbed with nine nodes acting as combined region server and DataNodes; we increase the number of clients until the throughput does not increase.

For sequential read, both systems achieve about 110MB/s. Pipelining reads in Salus does not improve aggregate throughput since also HBase has multiple clients to parallelize network and disk operations. For random reads, disk seek and rotation are the bottleneck, and both systems achieve only about 3MB/s.

The relative slowdowns of Salus versus HBase for sequential and random writes are, respectively, 19.4% and 16.4%, and significantly lower (12.8% and 11.1%)

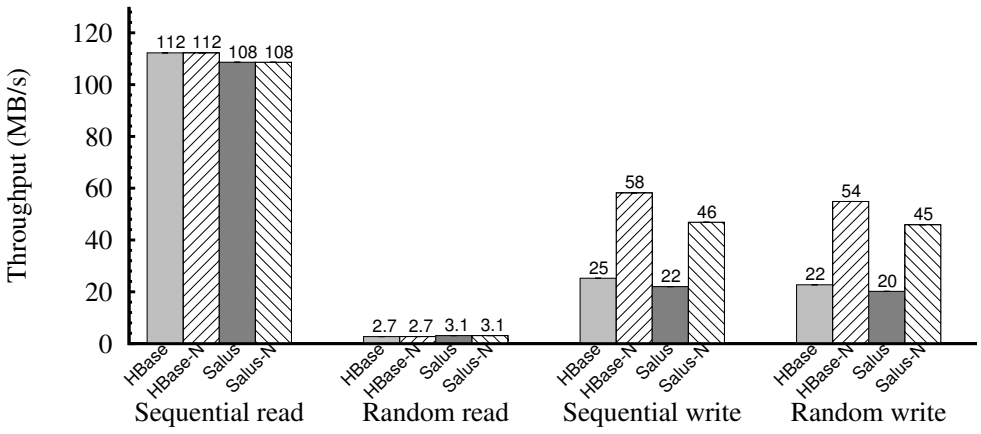

Figure 3.10: Aggregate throughput on small nodes. HBase-N and Salus-N disable compactions.

	HBase	Salus
Throughput (MB/s)	27	47
Network consumption (network bytes per byte written by the client)	5.3	2.4

Table 3.2: Aggregate sequential write throughput and network bandwidth usage with fewer server machines but more disks per machine.

when compaction is enabled, since compaction adds more disk operations to both HBase and Salus. Salus reduces network bandwidth at the expense of higher disk and CPU usage, but this trade-off does not help because disk and network bandwidth are comparable. Even so, we find this to be an acceptable price for the stronger guarantees provided by Salus.

Table 3.2 shows what happens when we run the sequential write experiment using the three 10-disk storage nodes as servers. Here, the tables are turned and Salus outperforms HBase (47MB/s versus 27MB/s). Our profiling shows that in both experiments, the bottleneck is the network topology that constrains the aggregate bandwidth to 1.2Gbit/s.

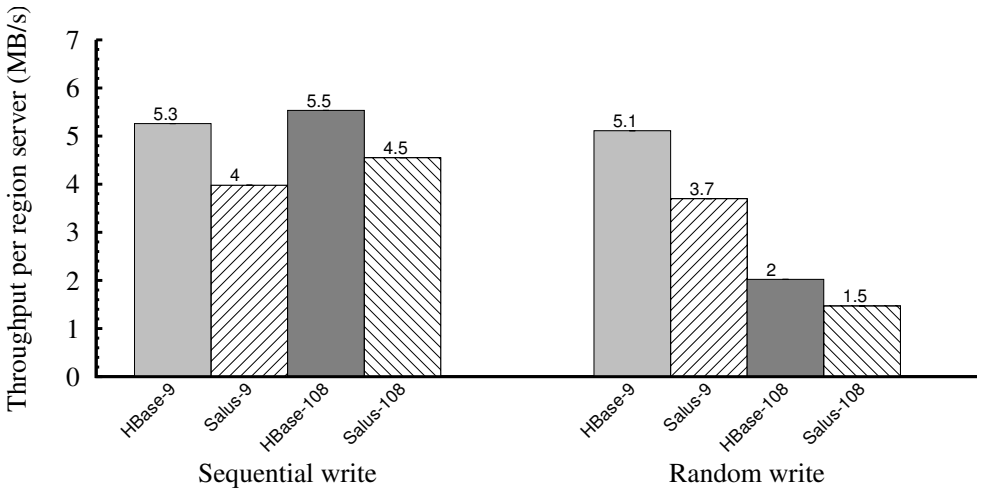

Figure 3.11: Write throughput per server with nine servers and 108 servers (compaction disabled).

Table 3.2 also compares the network bandwidth usage of HBase and Salus under the sequential write workload. HBase sends more than five bytes for each byte written by the client (two network transfers each for logging and flushing, but fewer than two for compaction, since some blocks are overwritten.) Salus only uses two bytes per byte written to forward the request to replicas; logging, flushing, and compaction are performed locally. The actual number is slightly higher than two, because of Salus additional metadata. Salus halves network bandwidth usage compared to HBase, which explains why its throughput is 74% higher than that HBase when network bandwidth is limited.

Note that we do not measure the aggregate throughput of EBS because we do not know its internal architecture and thus we do not know how to saturate it.

Scalability

In this section we evaluate the degree to which the mechanisms that Salus uses to achieve its stronger robustness guarantees impact its scalability. Growing by an

order of magnitude the size of the testbed used in our previous experiments, we run Salus and HBase on Amazon EC2 [8] with up to 108 servers. While short of our goal of showing conclusively that Salus can scale to thousands of servers, we believe these experiments can offer insights on the relevant trends.

For our testbed we use EC2's extra large instances, with DataNodes and region servers configured to use 3GB of memory each. Some preliminary tests run to measure the characteristics of our testbed show that each EC2 instance can reach a maximum network and disk bandwidth of about 100MB/s, meaning that network bandwidth is not a bottleneck; thus, we do not expect Salus to outperform HBase in this setting.

Given our limited resources, we focus our attention on measuring the throughput of sequential and random writes: we believe this is reasonable since the only additional overhead for reads are the end-to-end checks performed by the clients, which are easy to make scalable. We run each experiment with an equal number of clients and servers and for each 11-minute-long experiment we report the throughput of the last 10 minutes.

Because we do not have full control over EC2's internal architecture, and because one user's virtual machines in EC2 may share resources such as disks and networks with other users, these experiments have limitations: the performance of EC2's instances fluctuates noticeably and it becomes hard to even determine what the stable throughput for a given experimental configuration is. Further, while in most cases, as expected, we find that HBase performs better than Salus, some experiments show Salus with a higher throughput than HBase, possibly because the network is being heavily used and pipelined commit helps Salus handle high network latencies more efficiently: to be conservative, we report only results for which HBase performs better than Salus.

Figure 3.11 shows the per-server throughput of the sequential and random

write workloads in configuration with nine and 108 servers. For the sequential write workload, the throughput per server remains almost unchanged in both HBase and Salus as we move from nine to 108 servers, meaning that for this workload both systems are perfectly scalable up to 108 servers. For the random write workload, however, both HBase and Salus experience a significant drop in throughput-per-server when the number of servers grows. The culprit is the high number of small I/O operations that this workload requires. As the number of server increases, the number of requests randomly assigned to each server in a sub-batch decreases, even as increasing the number of clients causes each server to process more sub-batches. The net result is that as the number of server increases, each server performs an ever larger number of ever smaller-sized I/O operations—which of course hurts performance. Note however that the extent of Salus' slowdown with respect to HBase is virtually the same (about 28%) in both the 9-server and the 108-server experiments, meaning that Salus' overhead does not grow with the scale of the system.

Pipeline commit

Salus achieves increased parallelism by pipelining PUTs across barrier operations—Salus' PUTs always commit in the order they are issued, so the barriers' constraints are satisfied without stalling the pipeline. Figure 3.12 compares HBase and Salus by varying the number of operations between barriers. Salus' throughput remains constant at 18 MB/s as it is not affected by barriers, whereas HBase's throughput suffers with increasing barrier frequency: HBase achieves 3MB/s with a batch size of one and 14 MB/s with a batch size of 32.

Overhead of writing to witness nodes

To measure the overhead of Salus' new active storage protocol, which stores certificates on additional witness nodes, we compare the throughput of the new protocol

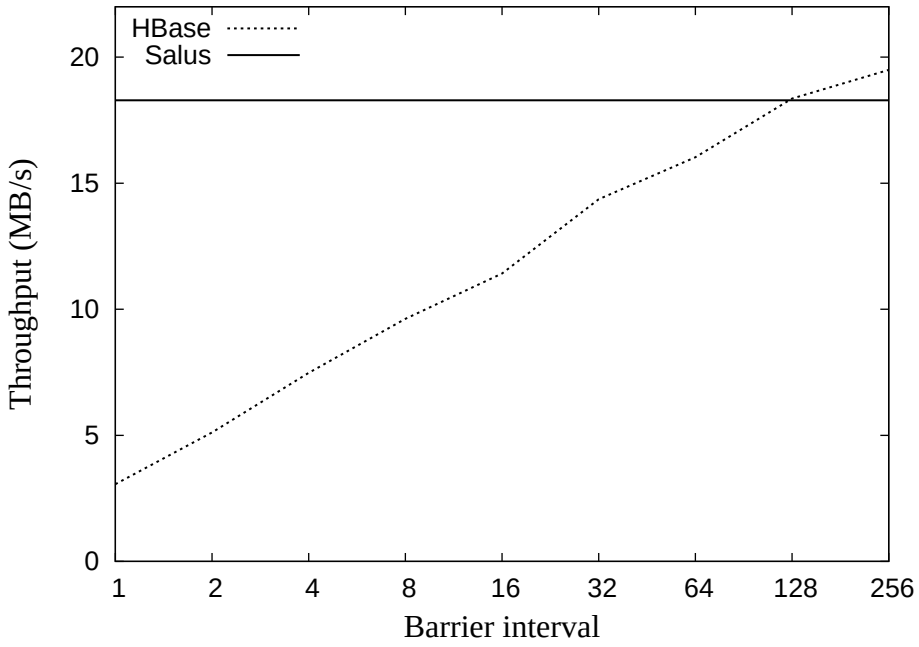

Figure 3.12: Single client sequential write throughput as the frequency of barriers varies.

to that of our earlier active storage protocol [106], which stores certificates on only three DataNodes. We perform this set of experiments on 12 servers, each equipped with 16 cores, 64GB of memory, and a single disk, and increase the number of clients until the system is saturated. Since witness nodes are only accessed for writes, we only measure write throughputs in these experiments.

Figure 3.13 shows that the aggregate throughput of the new protocol is comparable to that of the old protocol under both the sequential and the random write workloads. This result once again confirms one of the basic ideas of this dissertation: for storage systems processing large bulk of data, adding a small metadata can enhance the robustness of the system with little overhead.

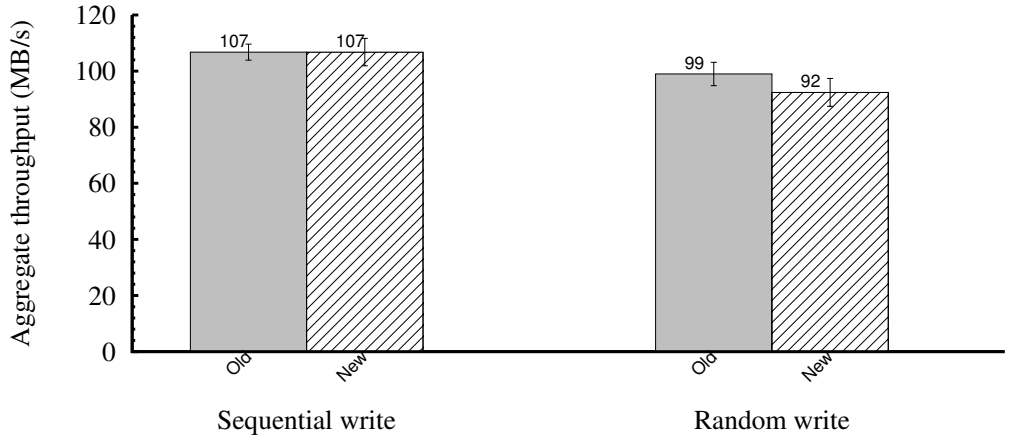

Figure 3.13: Overhead of storing metadata on additional witness nodes. The "New" protocol stores certificates on witness nodes while the "Old" protocol does not.

3.5 Conclusions

Salus is a distributed block store that offers an unprecedented combination of scalability and robustness. Surprisingly, Salus' robustness does not come at the cost of performance: pipelined commit allows updates to proceed at high speed while ensuring that the system's committed state is consistent; end-to-end checks allow reading from one replica safely; and active replication not only eliminates reliability bottlenecks but also eases performance bottlenecks.

Chapter 4

Exalt: A Tool to Evaluate Large-Scale Storage Systems

Exalt is a library that gives back to researchers the ability to verify the scalability claims of today's large storage systems, which, ironically, have become hard to corroborate precisely because of the scale of these systems.

The advent of Big Data has strained the scalability of traditional storage systems, and several new architectures have been proposed to respond to this challenge [12,23,27,32,44,60,82,95] by supporting up to hundreds of petabytes of storage and tens of thousands of storage nodes. Testing systems at such scale, however, requires access to tens of thousands of machines and at least as many disks, and few researchers have access to resources that plentiful: the rest of us have to design systems that are supposed to operate at a scale much larger than the infrastructure available to test them. Nor are such resource limitations affecting only academia: even industrial researchers at organizations with clusters of the necessary size may not be able to reserve them for large scale experiments, since these clusters are a primary source of revenue.

These limitations are typically sidestepped in one of two ways. The first is to

run experiments on a medium-sized cluster (100-200 machines) and extrapolate the results to larger scales. Although this approach may work reasonably well in some cases, the fundamental assumption on which it rests—that resource consumption increases linearly with the load and the number of machines in the system—does not always hold, as we show in Section 4.1. To make matters worse, sources of non-linear growth are sometimes hard or impossible to observe in small deployments. For example, the time needed to add a new block to an HDFS file [95] increases with the file's size, but it is only after that size has grown beyond what is likely to be observable in small deployments that the slowdown becomes a limiting factor for the system's performance.

The second common approach for predicting the behavior of large-scale systems is simulation [83, 98, 103, 110]. Unfortunately, the results of a simulation are only as accurate as the model on which the simulation relies; as systems grow in size and complexity, modeling them faithfully becomes prohibitive.

This dissertation proposes a third way: the Exalt library offers researchers the ability to test the scalability of a large-scale storage system by running its real code, but without requiring access to thousands of machines. The basic insight at the core of Exalt is that, in many large-scale experiments, how data is processed is not affected by the content of the data being written, but only by its metadata, such as its size. Exalt leverages this freedom by virtualizing the data, while keeping the metadata intact to ensure that the system continues to function correctly. Specifically, the format that Exalt clients use to write data, which we call *Tardis*, has two key advantages. First, it allows Exalt to compress the behavior of the system in both space and time. Space compression is a powerful tool for performing large-scale experiments: for example, running 10,000 storage nodes on just 100 machines can bring to light previously unknown scalability bottlenecks in the metadata service. Since compressed data takes much less time to write, compression in space

can in turn result in compression in time: with the system running faster, bugs and performance issues can be discovered more rapidly.

The second key advantage of Tardis is that it addresses a fundamental challenge in virtualizing data: being able to distinguish data from metadata. While the content of the former is not important for the system to function correctly and can therefore be virtualized, the integrity of the latter is essential. This problem is particularly prominent in modern storage systems, which employ a two-layer architecture where the upper layer uses the lower layer as black-box storage: files written to the lower layer contain both data and metadata, which look indistinguishable to the lower layer. The need to ensure the integrity of the metadata is why approaches that virtualize data by altogether disposing of file contents (e.g. [6]) cannot be used in our context.

In summary we make the following contributions:

- We introduce Tardis, a data representation scheme that allows data to be identified and efficiently compressed even at lower-level storage layers that are not aware of the semantics and formatting used by higher levels of the system. Tardis provides transparent, lossless, computationally efficient compression of data and achieves high compression ratios.

- We present a methodology that utilizes Tardis to test the scalability and robustness of large-scale storage systems: our goal is not to predict every aspect of the performance of such systems (e.g. their power consumption) but, more modestly, to identify scalability problems. Our approach has a "Truman-show" [101] feel: the part of the system whose scalability is being tested processes real data and interacts with the rest of the system as it would in a true large-scale deployment, while the rest of the system uses Tardis to compress data and achieve high degrees of colocation, thereby emulating the behavior of a large number of nodes.

- We present our experience using Exalt, a library that implements Tardis and uses our methodology to identify scalability issues in large-scale storage systems. Using Exalt we found several such issues in three mature storage systems: HDFS [95], HBase [12], and Cassandra [60], and fixed a subset of them. All the problems we identified manifest when the scale of the system becomes larger than a typical research cluster. In the case of HDFS, resolving these problems resulted in an order of magnitude improvement of the aggregate system throughput. Our ability to identify these issues was not, for better or worse, due to a prior deep understanding, but rather to the opportunity offered by Exalt to test them at an unprecedented scale.

The rest of this chapter is organized as follows. Section 4.1 discusses the common practices for testing the scalability of large systems. Section 4.2 introduces the Tardis data representation scheme and Section 4.3 describes how it can be used to identify scalability problems in large-scale systems. Section 4.4 reviews the assumptions of Exalt and discusses its applicability in various contexts. Section 4.5 presents our experience using Exalt to identify performance problems in three mature systems: HDFS, HBase, and Cassandra. Section 4.6 concludes our discoveries of Exalt.

4.1 Testing for scalability: common practices

When faced with the challenge of running experiments on a system whose scale vastly exceeds their infrastructure, researchers typically resort to one of two options: they either run the system at the largest scale they can afford and try to extrapolate their results, or they explicitly forgo running certain components of the system, substituting them with stubs that, ideally, maintain the interactions of the original components with the rest of the system, but are simpler and less resource-demanding

to run. We discuss both options, and why they are not well-suited for performing scalability tests on large-scale systems.

4.1.1 Extrapolation

A common approach to estimate the behavior of systems that are too big to test is to run them at a small or medium scale and then to extrapolate, based on those results, how they will behave at a large scale. For example, if the CPU utilization of a bottleneck node is 10% in a 100-node experiment, extrapolation would lead one to estimate that the system will scale to about 1,000 nodes. While attractive for its simplicity, this approach has several drawbacks that make it inaccurate in practice.

First, extrapolation is based on the assumption that resource usage grows linearly with the scale of the system. However, because of design choices and implementation issues, this assumption is frequently violated in practice. For example, HDFS uses an array to maintain a sorted list of files within a directory. Using an array causes insertion to be an $O(N)$ operation, where N is the number of files in the directory. As more files are added to the directory, insertion becomes increasingly expensive: indeed, the cost of adding N files to a directory is $O(N^2)$. Note that a more efficient directory implementation (e.g. a sorted tree map) does not restore linear growth in resource usage, but simply reduces the growth rate to $O(N \cdot \log N)$. In general, once the load on the system is not linear, accurate extrapolation becomes much harder, especially because, as we have seen, the system's performance may depend on the details of the implementation.

A second, more subtle drawback of extrapolation is that at small scales some important behaviors can easily escape notice. Consider again the above example of a workload of $O(N^2)$ complexity: as long as the value of N is low, the potential scalability bottleneck remains largely inconspicuous. To exacerbate the problem, measuring resource utilization is an inherently noisy process. For example, observ-

ing that a Java process uses 100MB of memory does not, by itself, indicate how much memory is being used by the data structures of that process. Answering that question requires accurate information about the amount of memory used internally by the JVM, the amount of non-garbage-collected memory, etc. The uncertainty added by measurement noise is significantly more prominent at lower scales, where resource utilization is low.

The final drawback, which is closely related to the previous one, is that extrapolation cannot be used to predict behaviors that are only triggered when some resource utilization reaches a certain threshold. For example, HDFS has a blocking disk-scanning procedure that becomes increasingly expensive as the system grows in size. Beyond a certain size, running the procedure causes the corresponding DataNode to start missing heartbeats, which in turn can cause it to be evicted and force all its data to be re-replicated, with serious performance repercussions.

4.1.2 Using stubs

Another technique for predicting the performance of a system too big to test is to emulate, rather than actually run, some of its components. The emulated components are implemented as stubs, running either locally or remotely. For this approach to be successful, the stubs should be simple to implement and require much more modest resources than the original components they stand in for; at the same time, they should be able to correctly exercise the rest of the system, allowing it to be stress-tested at scale using relatively modest resources.

While attractive in theory, the promises of emulation are often elusive in practice: reproducing accurately the behavior of a non-trivial real system component is hard, and in the process the stub component can end up being almost as complex as the real one, defeating its purpose.

We faced this challenge first-hand when trying to test the scalability of the

HDFS NameNode using stub DataNodes. In particular, our goal was to create a large number of stub DataNodes and use them to stress-test the NameNode. Our first attempt did not involve the DataNodes in the protocol at all; to create files and add blocks to them, clients simply invoked `createFile` and `addBlock` at the NameNode. However, the system did not work, since the NameNode expects the DataNodes to confirm the receipt of each block. We therefore modified our clients to notify the stub DataNodes, so they could in turn appropriately notify the NameNode. This did not work, either: the NameNode, we discovered, also expects each DataNode to periodically report the list of blocks it stores on disk. After several frustrating iterations, we eventually came to realize that emulating the correct behavior of DataNodes would have required us to reimplement the full HDFS protocol, including all inter-DataNode communication, local bookkeeping, etc.

4.2 Compressing data with Tardis

Our approach is based on a simple intuition: for the purposes of testing the scalability of large-scale storage systems, it is typically the size of the data being written that matters, not its actual content. We are then free to *choose* what data clients write during our tests: our work explores the opportunities that this freedom affords.

Specifically, our approach is to design a data format that achieves fast and efficient compression and decompression. As we discuss in Section 4.2.3, using compressed data lets us colocate multiple nodes on the same machine, which in turn enables running large-scale experiments on a small infrastructure.

Before presenting Tardis, our compression scheme, we set forth the requirements that it must fulfill.

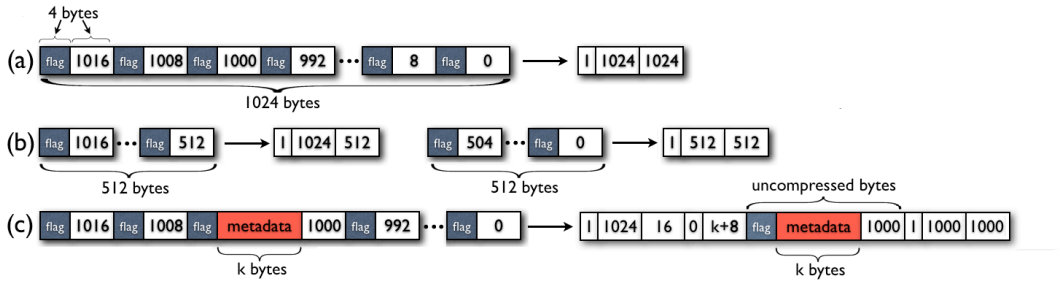

Figure 4.1: Examples of the Tardis format in compressed and uncompressed form.

4.2.1 Compression scheme requirements

The scheme must be lossless. While compression can reduce resource usage and allow node colocation, the ability to recreate the original data is essential. Modern large-scale storage systems typically use a two-layer architecture, where the upper layer uses the lower as a black-box storage [12, 23, 27]. What appears like generic data to the lower storage layer may actually be metadata necessary for the correct functioning of the upper layer; it is critical that none of this metadata be lost.

The scheme must achieve a high compression ratio. The motivation for this requirement is straightforward, since the compression ratio directly affects the amount of colocation we can achieve.

The scheme must be computationally efficient. As a counterexample, consider a straw-man scheme in which clients simply write sequences of 0's. This scheme offers obvious opportunities for significant compression; however, if it is possible for the system to interleave client data with metadata, the compression algorithm would need to scan all the input bytes to determine where the sequence of 0's begins and where it ends. The disk and network bottlenecks would have been removed, but at the expense of introducing a CPU bottleneck, severely limiting the scalability of this scheme.

Data chunks should be independently compressible. Modern storage systems do not necessarily store data as a single unit, but instead split it into multiple, separately stored chunks, which must be independently compressible. Meeting this requirement is challenging, however, since a client in general has no control over how data is divided into chunks. For example, in HBase the procedure of splitting data into chunks depends on a non-deterministic race between multiple threads.

4.2.2 Tardis compression

We introduce a novel compression scheme, called *Tardis*, that satisfies the above requirements. Tardis consists of a data format and an algorithm for compressing and decompressing the data. Intuitively, Tardis aims to achieve the following two complementary goals. When no metadata is inserted in the middle of the data, the compression algorithm should be able to compress the entire data after scanning only a small fraction of it. Otherwise, the compression algorithm should be able to quickly identify the location of the inserted metadata.

Data format Clients write data as a series of `<flag> <marker>` entries, where `<flag>` is a predefined byte sequence that does not appear in the system metadata, and `<marker>` denotes the number of remaining bytes in the data. For example, using a 4-byte flag and 4-byte markers, 1KB of data would be formatted as:

```
<flag>1016<flag>1008...<flag>8<flag>0
```

In this example, the first marker denotes that there are 1016 bytes remaining in the sequence, since the (first) flag and the marker itself are 4 bytes each. Of course, the size of flags and markers need not be the same: our prototype uses 8-byte flags and 4-byte markers.

Compressed data format Given a byte sequence in the above format, the compression algorithm would simply need to return its length. However, to enable

data chunks to be independently compressible, the algorithm actually returns two numbers: the starting byte of the sequence as well as its length. In the above example (also illustrated in Figure 4.1a), if the entire 1KB of data were being compressed, the result would be the pair (1024,1024). If, however, the data were split into two chunks of 512 bytes each (Figure 4.1b), the first chunk would be compressed as (1024,512) and the second as (512,512).

As we discussed above, in modern storage systems data and metadata are frequently stored together. Figure 4.1c shows an example where metadata is inserted in the middle of a Tardis sequence. In this case, the metadata splits the original sequence into two subsequences, of length 20 and 1004, respectively. Ideally, we would like to compress each of these sequences separately, leaving the metadata uncompressed. However, since in this case the metadata is inserted in the middle of a flag-marker pair, we simply leave these 8 bytes—the flag and the corresponding marker—uncompressed.[1] This shortens the first subsequence to a length of 16 and the second subsequence to 1000. Note that even if the metadata were not aligned with the flags and markers, the result would be the same: only the flag-marker pair that is split by the metadata is left uncompressed and the rest of the data is compressed as two separate subsequences.

To distinguish between compressed and uncompressed data during decompression, an uncompressed sequence is preceded by a 0 and a 4-byte integer denoting its length, while a compressed sequence is preceded by a 1.

Compression Figure 4.2 shows the pseudocode for the Tardis compression algorithm. The main function, `TardisCompress`, calls the `FindSubsequence` function iteratively until all input data has been consumed. When `FindSubsequence` returns a new subsequence (line 7), the main function appends the appropriate bytes to the compressed data buffer. We detect the presence of metadata between

[1]It is actually possible to include the flag in the compressed sequence, but we omit this optimization for simplicity of presentation.

```
1  #define unit_size = flag_size + marker_size

3  (compressed_data) TardisCompress(data)
4      result = empty buffer
5      index = 0
6      while index < data.len
7          (pos,marker,len)=FindSubsequence(data,index)
8          if pos == -1
9             result.AppendMeta(data,index,data.len-index)
10            return result
11         else
12            if pos > index
13                result.AppendMeta(data,index,pos-index)
14            result.AppendTardis(marker,len)
15            index = pos+len
16     return result

18 (pos,marker,length) FindSubsequence(data,startIndex)
19     (pos,marker) = ScanForFlag(data,startIndex)
20     if pos == -1
21         return (-1,-1,-1)
22     lastMarker = data.len-(unit_size+(data.len-pos)%unit_size)
23     target = min(pos+marker,lastMarker)
24     marker2 = BinarySearch in data from pos to target
25         for the rightmost flag-marker pair such that:
26         (pos2,marker2) = ScanForFlag(data,target) and
27         pos2 != -1 and pos2-pos == marker-marker2
28         return (pos, marker, marker-marker2+unit_size)
29     if no such marker2 is found
30         return (-1,-1,-1)

32 (position, marker) ScanForFlag(data, startIndex)
33     index = linearly search data for (flag,marker) starting at startIndex
34     if index >= 0
35         return (index, marker)
36     else
37         return (-1, -1)
```

Figure 4.2: Pseudocode for Tardis compression.

two subsequences by checking whether the starting position of the new subsequence (*pos*) is after the end of the previous subsequence (*index*). If so, we append a 0 to denote the beginning of an uncompressed sequence, followed by the length of the metadata, and finally by the metadata itself, uncompressed (*AppendMeta*, line 13).

It is then time to add the new subsequence. To denote that what follows is compressed, we append a 1 before the compressed form of the Tardis subsequence (which, recall, consists of the starting point and length of the subsequence) (*AppendTardis*, line 14).

Function `FindSubsequence` is the core of the algorithm: its task is to identify a Tardis subsequence. Two factors complicate this task: the sequence may have been split into multiple chunks and metadata may have been inserted somewhere in the sequence. Given a starting index in the data, `FindSubsequence` first scans the data to find the first flag, indicating the start of a Tardis sequence, and reads the corresponding marker (line 19). Then, it checks whether some metadata has been added in the middle of this sequence. The check is simple: if no metadata is inserted between two markers with values A and B, then these markers should be placed $B - A$ bytes apart. The purpose of lines 22-23 is to determine which marker should serve as marker B. If the original sequence is not split across chunks, then B is marker 0, which should be m bytes after the first marker, where m is the value of the first marker. Otherwise, B is set to the last marker of the current chunk. If the difference between the values of markers B and A is indeed equal to the byte distance between the markers, the algorithm has found an uninterrupted Tardis subsequence. If that is not the case, the algorithm performs a binary search to find the rightmost flag-marker pair that satisfies the above condition, leveraging the fact that the values of the markers form a sorted sequence (lines 24-30).

In practice, the common case is very simple: as long as there is no metadata inserted in the byte sequence, the compression algorithm needs only to check the first and last number of the sequence. This allows Tardis to compress data much faster than off-the-shelf compression algorithms. For example, when compressing data chunks of 1MB, Tardis is about 33,000 times faster than Gzip [50] and 2,300 times faster than the straw man compression scheme where client data consists only of 0's and the compression algorithm simply scans the data and compresses sequences of 0's into an integer denoting their length. Of course, the comparison to Gzip is not apples-to-apples, since Gzip is a generic compression algorithm; what it does show, however, is that being able to choose the data format drastically reduces

the CPU overhead of our approach.

Decompression The decompression algorithm is straightforward. Given a compressed sequence, it iterates through each sequence, whether compressed (preceded by a 1) or uncompressed (preceded by a 0 and the length of the sequence). Uncompressed sequences are copied without modification, while compressed sequences are expanded to their uncompressed form.

Choosing the flag To prevent portions of metadata from being accidentally compressed, the flag sequence should never appear in the metadata. If it did and, by unlucky coincidence, the length value following the fake flag pointed to another flag followed by a 0, all that sequence of bytes would be compressed. Although we could altogether eliminate this danger,[2] it seems unnecessary: Exalt is not intended for production use, and an accidental compression would simply require us to rerun the affected experiment. With a sufficiently large flag, the odds of a false positive can be driven arbitrarily low: our pragmatic approach was to choose as flag an 8-byte random sequence and take our chances. Our experiments have yet to produce a false positive.

4.2.3 Using compression to enable large-scale tests

Since we are attempting to run a large number of nodes on a much smaller number of machines, we will necessarily have to colocate multiple nodes on the same machine. However, such colocation will cause significant contention on the physical resources of the machine. Specifically, the disk- and memory capacity, and the disk- and network bandwidth available to each machine are typically enough to support only a single node, making straightforward colocation infeasible.

Data compression can help here: storing compressed data on disk decreases

[2]It would suffice to escape the flag sequence in the metadata. However, this would require intrusive modifications to the server code, as all metadata insertions would need to be aware of the escaping logic.

the disk capacity and bandwidth requirements of each node, as well as memory capacity and network bandwidth. Of course, data compression is not without cost; in this case, the cost is CPU utilization.

This tradeoff, however, is very attractive for storage systems, where CPU cycles are plentiful and bandwidth and storage capacity are typically the system's bottleneck. It also opens the door to emulating the behavior of storage systems too big to test using HPC computation clusters: indeed, as we will see in Section 4.5, our analysis of the scalability of HDFS/HBase/Cassandra has been performed by running Exalt on the Stampede high performance cluster at the Texas Advanced Computing Center (TACC) [97].

If data compression is used without colocation, it results in a system that is "compressed" in time, rather than space, since each write will take less time to complete. Running the system at an accelerated pace offers the potential of identifying bugs or performance problems much faster: Section 4.5.1 discusses a case where time compression allowed us to identify a problematic behavior about 100 times faster than in a real deployment.

4.2.4 Implementation

Our implementation of Exalt performs data compression for three key resources: disk, network, and memory. Our goal is to be minimally intrusive. While in-memory compression does require minor modifications to the source code of the storage system being tested, we achieve fully transparent disk and network compression by using byte code instrumentation (BCI) to modify the relevant Java library classes (Socket, SocketInputStream, SocketOutputStream, SocketChannel for network compression; File, FileInputStream, FileOutputStream, RandomAccessFile, and FileChannel for on-disk compression).

File compression is more challenging than network compression because the

file interface allows a user to partially update existing data. When that data is already compressed, updating it in place is not straightforward. A naive solution would be to decompress the existing data, update it, and compress it again. However, if the old and the newly compressed data have different sizes, all following data chunks would have to be moved. To address this problem, similarly to the Log-Structured File System (LFS) [90], we transform in-place update operations into append operations. This allows us to efficiently process in-place updates, with only a small bookkeeping overhead to keep track of the latest version of each range of bytes.

Memory compression In-memory data structures do not use a well-defined interface, such as the File or Socket abstractions used by the disk and network. As a result, transparently modifying these data structures to compress and decompress data at the application layer is very hard.[3] Instead, when the in-memory data needs to be compressed, we manually modify the source code of the system. Fortunately, this process is quite simple. One needs only identify the data structures that hold the client data. When data is stored in the data structure, it is compressed; when data is retrieved from the data structure, it is decompressed. For example, compressing the in-memory key-value store of HBase required adding 71 lines of code across four files.

4.3 Finding scalability bottlenecks

Data compression gives us the ability to colocate multiple nodes on a single physical machine: in this section, we discuss how we can selectively use this ability to draw meaningful conclusions about the scalability of a large-scale storage system. We will view the system as a collection of *real* and *emulated* nodes. A real node runs the

[3]Transparent compression of in-memory data could be potentially implemented at the kernel level, but it would sacrifice portability.

system's actual code and handles unmodified data. An emulated node still runs the system's actual code, but, as needed to support colocation, may (i) store compressed data on its disk, (ii) send compressed data over the network, when it communicates with other emulated nodes, and (iii) store compressed data in memory.

4.3.1 Exalt methodology

We use this combination of real and emulated nodes as a microscope of sorts that we can focus on a part of the system to identify performance bottlenecks. To ensure that the part of the system "under inspection" behaves as it would in a real large-scale deployment, we leave the corresponding nodes real, while using emulated nodes for the rest of the system. This approach works particularly well at identifying performance issues at centralized components that can become a bottleneck as the scale of the system increases (e.g. HDFS NameNode, HBase Master). Section 4.5 discusses our experience using this technique to find scalability problems in real systems.

A downside of this methodology is that it may not discover scalability problems that arise at the nodes that are being emulated. To address this issue, after having stress-tested the part of the system under inspection by using the maximum amount of colocation for emulated nodes, we perform a new set of experiments where a small subset of formerly emulated nodes are also run as real, while the rest is kept emulated. This hybrid configuration makes it possible to identify scalability problems also at nodes that are not under inspection, while maintaining a high degree of colocation, but it is not a panacea: for example, it is still unable to detect performance issues that only manifest when a large number of nodes that are not under inspection perform some collective action (e.g. system-wide recovery).

4.4 Limitations and applicability

Exalt relies on a number of assumptions to provide high-degrees of node colocation. This section reviews these assumptions and discusses which of them can be weakened to widen the applicability of our approach.

Exalt is primarily designed to evaluate I/O-intensive applications like distributed file systems storing large files [44, 95] or key-value stores with relatively large values [23, 82]. Applications that are not I/O-intensive or store small values cannot benefit significantly from Tardis, as they gain little by compressing data. In Section 4.5.2 we explore in more detail how the size of the value in a key-value store affects the colocation ratio of Exalt.

Our current implementation of Exalt makes two additional assumptions: first, that the target application does not modify the data written by the clients, although it can split the data and insert metadata; and second, that experiments are not sensitive to the contents of the data, so that benchmarks can operate with synthetic data.

While these assumptions hold for the systems we have so far applied Exalt to, they are not fundamentally required for Exalt to be applicable. We consider below some popular techniques that violate these assumptions and discuss how our implementation of Exalt can be modified to work in conjunction with them.

Encryption and erasure coding Both techniques involve encoding data into a different format, violating the assumption that client data is immutable. To handle these cases, Exalt would compress the data using Tardis *before* encoding it, and then add *filler bytes* as necessary to match the length of the (encoded) original data. Filler bytes would use the same format as Tardis (making them highly compressible), but with a different flag sequence (so that they can be distinguished from real data). When reading the data, Exalt would remove the filler bytes before performing decryption and then decompress the Tardis sequence to obtain the client

data.

Deduplication Deduplication compares the contents of different data units (files, chunks, etc.) to eliminate duplicates and, by making execution dependent on the actual data, violates our second assumption. Indeed, deduplication schemes that directly compare the units' data are incompatible with Exalt. However, Exalt can still be applied to deduplication approaches that only compare hashes of data units. Exalt would first compute the hash of the client data and then replace the client data with data formatted using an extended version of Tardis, which inserts the hash of the data unit between the flag and the marker. The deduplication module could then use this hash directly to identify duplicate data units.

Compression If the system being tested already uses compression, it is in general not possible to use synthetic data at the clients, since the compression ratio depends on the actual data. If, however, compression is performed only at the client side, Exalt could apply a technique similar to the one used to handle encryption and erasure coding: the client would first compress the real data to determine its compressed size, then create synthetic (Tardis) data, compress it using Tardis' compression and finally append the right amount of filler bytes to match the length of the (compressed) real data.

Data sensitive applications Many applications use SQL-like languages for their queries. The execution of these queries depends on the data, since SQL predicates can be expressed as a function of the data. The rest of the data, which does not affect the processing of the queries, can be synthetic. The efficiency of Exalt in these cases depends on the ratio of sensitive to non-sensitive data.

4.5 Case studies

Exalt allows us to evaluate storage systems at an unprecedented scale. This section presents our experience applying Exalt to evaluate three real-world storage systems:

the Hadoop Distributed File System (HDFS) [95], the HBase key-value store [12], and the Cassandra key-value store [60]. We chose these systems for several reasons. First, not only they are popular in their own right, especially among researchers, but their architectures are representative of a broad range of existing large-scale storage systems. Second, all systems are open-source, which allows us to perform code modifications where necessary (i.e. for in-memory compression). Finally, all systems have a large development community that has produced a mature and stable codebase. Despite the maturity of the code, we identified several performance issues that arise as the scale of the system increases. Our ability to diagnose these issues was not due to a prior deeper understanding of these systems, but simply to the ability to evaluate them at an unprecedented scale.

In our evaluation, we run HDFS and HBase at a scale about 100 times larger than the size of the infrastructure available to us. For example, one of our experiments uses 96 machines to run an HDFS cluster with 9600 DataNodes. Our experiments identify a number of performance problems that arise at such large scales. Some of these problems pertain to low-level implementation details, while others are due to high-level design choices. For example, we find that storing many files on an HDFS directory causes file creations to that directory to become increasingly slow; and that keeping less than $\frac{3}{4}$ of the region data in the memory of an HBase region server causes its performance to degrade precipitously. Using Exalt, we were able to identify and fix many of these problems, improving as a result the aggregate HDFS throughput by an order of magnitude. Our experience with Cassandra is different: the scalability of Cassandra is so limited that its scalability problems can be identified even without the help of Exalt.

Unfortunately, we have not yet been able to validate our results by running the actual systems at a large scale: after all, it is the very reason that we do not have access to such plentiful resources that has motivated our work in the first place.

The largest validation we have performed involved running HDFS on 1,500 nodes of the Stampede cluster at the Texas Advanced Computing Center [97]: while our results confirm the prediction of Exalt for that configuration, the scale of the system is still too small to exhibit even the first of the scalability issues identified by Exalt. Our current confidence in Exalt's effectiveness stems from two sources. First, for each problem that Exalt has identified, we have traced the cause of the problem in the source code and, if possible, we fixed it and run the modified system to confirm that its performance has been improved. Second, some of our findings have been confirmed by engineers at Facebook, among the few who have access to a large-scale deployment of HDFS [39].

Most of our experiments were performed on the Stampede cluster at TACC, whose machines have 16 cores, 32GB of memory, but only 80GB of local disk storage. Since our access to TACC was limited, we ran some of our experiments on three local machines with 16 cores, 64GB memory and ten 1TB disks each. These machines were used to test the capacity limitations of individual storage nodes.

4.5.1 Case study: HDFS

HDFS [95] is an open source implementation of the Google File System (GFS) [44]. Each HDFS cluster contains a single NameNode that stores the file system namespace information and several DataNodes that store the file contents. Each file is split into multiple blocks and each block is stored on three DataNodes. When a client creates a file or adds a block to an existing file, it first contacts the NameNode, which responds with a list of the DataNodes that will store the new block. The client can then directly write the block contents to these DataNodes.

We mainly focus on write workloads since they are more likely to cause scalability problems. Unless otherwise specified, in our experiments each client creates a file in its own directory, writes 192MB of data to it (as suggested by the

HDFS developers in their white paper on how to test HDFS' scalability [94]), closes the file, and then starts a new file. This workload achieves the highest scalability among all workloads that we tried; Section 4.5.1 describes the performance problems caused by other workloads. We use a block size of 128MB and the default three-way replication (again, as suggested in [94]). Unless otherwise specified, we run DataNodes and clients in emulated mode while the NameNode runs in real mode.

For the above workload, Tardis achieves a compression ratio of over 500, but in practice the degree of colocation is limited by CPU utilization: we colocate 100 DataNodes on one machine and achieve an effective write bandwidth of 10GB/sec on a disk with 100MB/sec physical bandwidth. For experiments with modest storage capacity requirements, we can increase the write bandwidth to 20GB/sec by writing to tmpfs, an in-memory file system. Our largest configuration experiment uses 192 server machines, to emulate an HDFS cluster with 19,200 DataNodes.

HDFS throughput scalability

In some sense, the result of our experiments to test the scalability of HDFS is not surprising: the bottleneck of the system is the centralized NameNode. What is perhaps surprising is that, thanks to Exalt, we were able to increase the system throughput by an order of magnitude without changing the architecture of the system.

Figure 4.3 reports the results of our experiments. On the x-axis we increase the number of DataNodes and on the y-axis we plot the aggregate throughput of the system, as observed by the clients. The vertical arrows represent the process of fixing an issue that was limiting the system throughput. When an issue is fixed, we rerun the experiment for the same number of DataNodes, to verify that the system indeed achieves a higher throughput. For reference, we also plot a straight line that shows the ideal throughput achievable by a perfectly scalable system that leverages the full bandwidth of all disks (100MB/s).

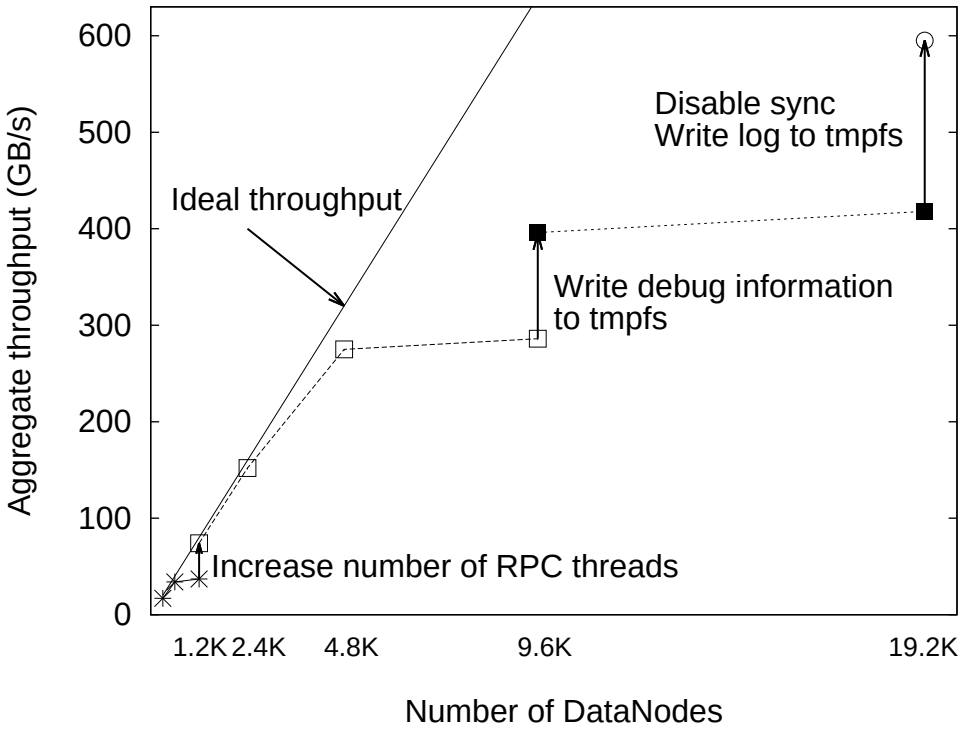

Figure 4.3: HDFS throughput scalability.

Our first experiment shows that the original HDFS system quickly saturates at around 37GB/s. We discovered through profiling that the default number of RPC threads at the NameNode was limiting the achievable throughput; increasing the number of RPC threads from 10 to 256 allows the NameNode to achieve much higher throughput.

After fixing the first issue, the system saturates at around 286GB/s. Further profiling showed that the I/O accesses at the NameNode were becoming the system bottleneck. More specifically, the NameNode debug information was being stored on the same disk as its log file, which, of course, hurts the speed of logging. Our solution was to write the debug information to tmpfs instead, thereby making sure

Memory size	1GB	2GB	4GB	8GB
HDFS Capacity	1.15PB	2.35PB	4.76PB	9.49PB

Table 4.1: HDFS space scalability as a function of NameNode memory size.

that the NameNode log file was accessing the disk with full speed. Alternatively, one could store the debug information on another disk, if one were available.

Applying the second fix increases the system throughput to 418GB/s, at which point the system again becomes saturated. This finding is consistent with the scalability assessment of the HDFS developers that with each client having a write throughput of 40MB/s, "10,000 writers can produce enough workload to saturate the name-node" [94], which corresponds to an aggregate throughput of 400GB/s. While this assessment was obtained using extrapolation, we consider it reasonably accurate since it is based on a large deployment of 4,000 nodes.

Since we suspected disk I/O to be the system bottleneck at this point, we performed a final experiment in which disk `sync` is disabled and the NameNode writes all logs to tmpfs. The purpose of this experiment is to project the scalability of the system in the presence of a fast storage medium (e.g. NVRAM, SSD). In this configuration, the system throughput increases by a further 42%, to a maximum throughput of 595GB/s.

Of course, we do not claim that Exalt's throughput predictions are perfectly accurate; on the contrary, we acknowledge the limitations of running a system whose resources are partially emulated. Nonetheless, the benefits of Exalt are clear: it allowed us to test the system's real code and identify and resolve performance issues at a scale that would have otherwise remained the sole province of a few large companies.

HDFS capacity

The capacity of an HDFS cluster is limited by the amount of memory available to the NameNode. In this experiment, we try to measure how much memory the NameNode needs per 1PB of HDFS storage space. Table 4.1 shows that the capacity of HDFS grows linearly with the amount of memory at the NameNode. In particular, 1GB of NameNode memory can support about 1.2PB of raw HDFS space (400TB of data, since blocks are 3-way replicated). This result is close to the estimation of HDFS developers: "1GB of metadata $\approx$ 1PB of physical storage" [94].

Using Exalt allows us to perform this experiment using only 16TB of disk storage, while a real deployment would require a total of 10PB of disk storage.

Performance degradation in HDFS

The above experiments use a workload that provides high scalability. Other workloads are not as accommodating. We evaluate two such workloads that can drastically degrade the performance of HDFS.

In the first workload, all clients create files in the same directory. As shown in Figure 4.4, the aggregate system throughput steadily decreases as more files are created. Further profiling allowed us to identify the cause of this behavior in the source code: the NameNode uses an ArrayList data structure to maintain an alphabetically sorted list of the files inside a directory. Adding an element to a sorted array is an O(N) operation, since it requires a suffix of the sorted array to be shifted by one position. Therefore, the bigger the directory, the longer it takes to add a file to it. As a double check, we verified that, if we limit the number of files written to each directory, creating more files does not cause a performance degradation.

In the second workload, one client creates a file and keeps appending data to it. As shown in Figure 4.5, once the file grows sufficiently large, the aggregate

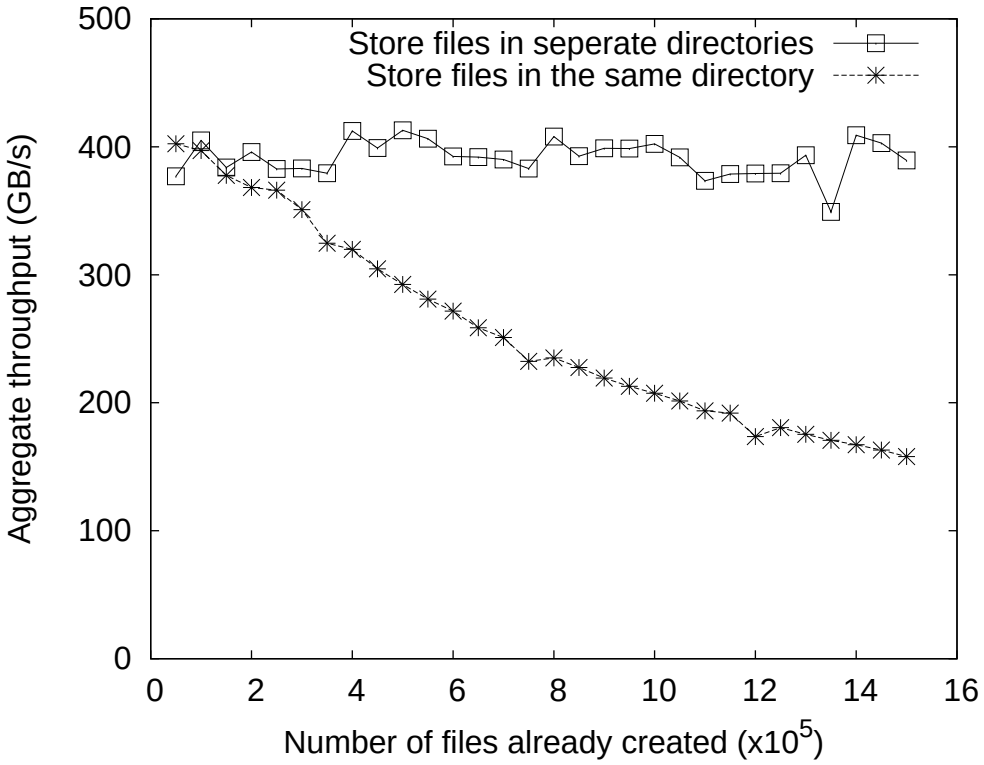

Figure 4.4: HDFS throughput degradation as the size of directories increases.

system throughput decreases steadily. Note that in this experiment there are only a few clients and the system is not fully saturated, which accounts for the fact that the aggregate system throughput is lower than in the previous experiment. Profiling led us to the cause of the problem: before the NameNode creates a new block for a file, it needs to calculate the file's length. It does this by scanning all existing blocks and computing the sum of the lengths of all blocks. This, too, is an O(N) operation. We fixed this problem by adding a length field to each file and updating the field when a block is added or updated. As Figure 4.5 shows, after applying our fix the system throughput no longer decreases as the files grow in size.

As before, Exalt allows us to identify these performance issues without re-

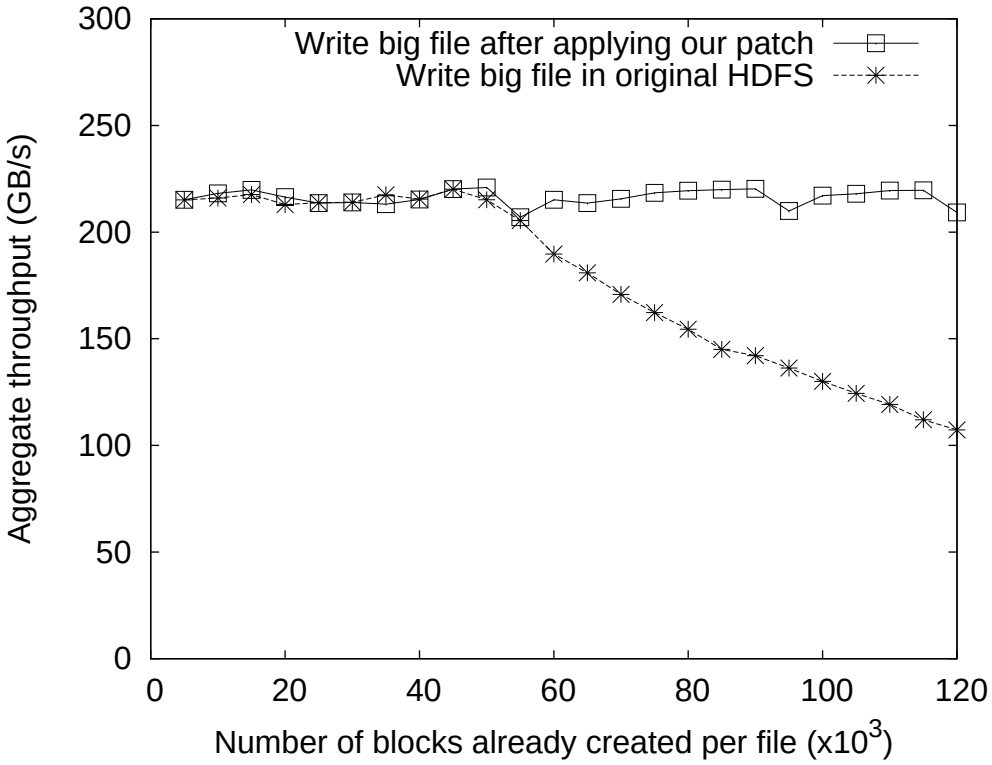

Figure 4.5: HDFS throughput degradation as the size of files increases.

quiring access to a large amount of disk storage. Running this experiment in a real deployment would require 900TB of disk storage; with Exalt, we only need 1.5TB.

DataNode scalability

As disk capacities increase every year, and most HDFS deployments use multiple disks per DataNode, it is important for the DataNode's performance to not decrease as more storage capacity is added to it. While running HDFS in hybrid mode—keeping some DataNodes real—we observed uncommonly high latencies for some requests. Our profiling indicated that the source of the problem was a disk scan that the DataNode periodically performs on all its blocks. Figure 4.6 shows that the time

a real node takes to perform this scan increases linearly with the number of blocks stored on the disk. Unfortunately, this scan is a blocking operation, preventing write requests and heartbeats from being sent or received. As the duration of this scan becomes longer, it can have serious performance consequences, including timeouts at the clients or even missed heartbeats, which would cause unnecessary re-replication of the DataNode's data. This issue is confirmed by Facebook engineers; to address it, they modified HDFS to allow the block scan to be performed in parallel with heartbeats and write requests [39].

While reproducing this problem is easy, triggering it in a real deployment would require 8TB of disk storage on a single DataNode; using Exalt, we triggered this problem using an 80GB disk. After identifying the problem, we reproduced it on a real DataNode with 8TB of disk storage (Figure 4.6).

Note that although it could be triggered with only a few machines, this problem would be hard to identify and tedious to reproduce during debugging, since it would take at least a few hours for the latency increase to be observable. Exalt's time compression helps in this case. If emulated nodes have exclusive access to a machine's resources, the system works at an accelerated speed: in this example, the problem would manifest itself in a matter of minutes.

4.5.2 HBase

HBase [12] is a distributed key-value store built upon HDFS. The basic data unit of HBase is a region, which corresponds to a continuous key range in a table. An HBase cluster includes a Master, responsible for assigning regions to different region servers. Client requests to a specific region are directed to the corresponding region server. The region server processes write requests by logging them to HDFS while also keeping them in a memory buffer called *memcache*. When the size of the memcache exceeds a threshold, the region server writes the whole memcache into a checkpoint

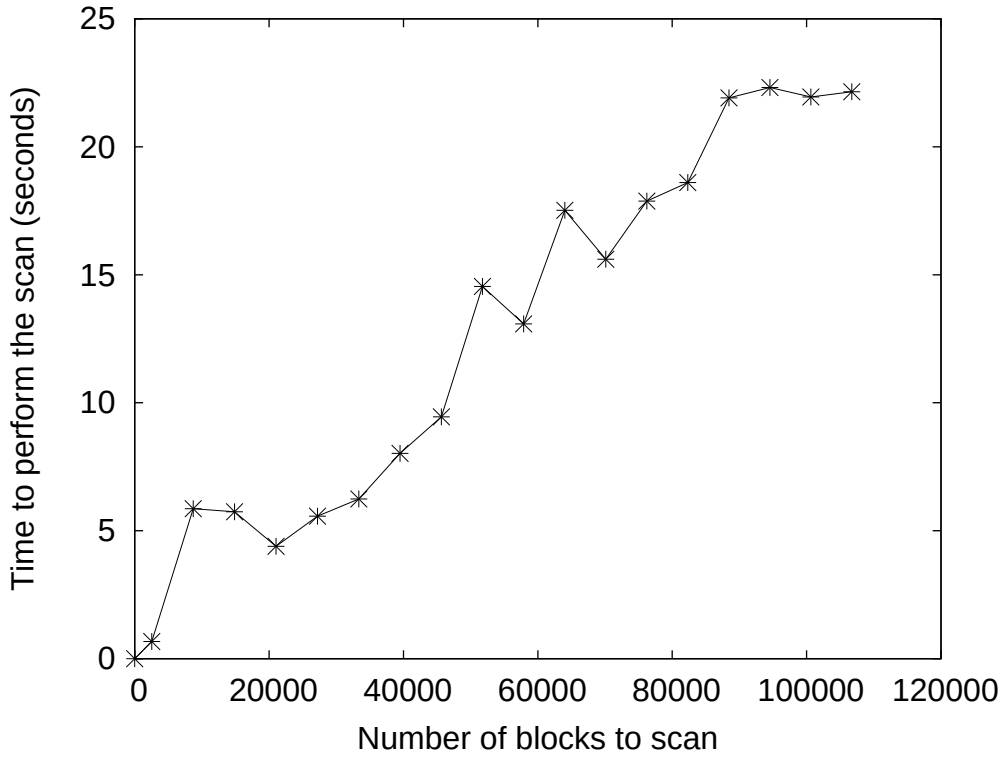

Figure 4.6: Time of the block-scan procedure on a DataNode, as the number of blocks increases.

file on HDFS, so that it can garbage-collect the previous log files. A checkpoint is also taken if the total memory usage across all regions exceeds some limit; in this case, a region server checkpoints the region with the largest memcache. When necessary to free up space, the region server performs *compaction* to merge several checkpoints. In essence, a region server transforms the random access patterns of a key-value store into the append-only interface of HDFS. When a region grows large, HBase splits that region into two for load balancing; conversely, if two adjacent regions are too small, they are merged into one. Apart from the Master and the region servers, an HBase cluster incorporates a ZooKeeper ensemble that performs lease management.

We evaluate HBase using a simple workload that can achieve a high throughput: we create enough regions so that each region server stores about 10 regions. We start multiple clients that randomly write key-value pairs to those regions. The key size is 4 bytes and the value size is 1MB. To measure the maximum achievable throughput, we disable split, merge, and compaction operations—to ensure that split and merge operations do not occur, we limit the number of key-value pairs written to a region. We plan to study the effects of split, merge, and compaction in the future.

Our experiments keep the HBase Master, HDFS NameNode and ZooKeeper cluster real, while all DataNodes and region servers are emulated. In each experiment we assign 500MB of physical memory to region servers. However, we perform in-memory compression, which effectively increases each region server's memory to 16GB.

Figure 4.7 demonstrates the throughput scalability of HBase as the number of available region servers increases. Note that the raw throughput of HBase is much lower than that of HDFS (see Figure 4.3). This is due to two reasons: first, HBase needs to write data twice to HDFS—once for logging and once for checkpointing. Second, region servers are relatively more CPU-intensive than DataNodes and therefore cannot benefit as much from colocating multiple nodes on the same machine.

HBase can achieve a maximum write throughput of about 80GB/s. Considering that HBase writes data twice, this translates to a 160GB/s throughput at the HDFS layer, which is about 40% of the maximum throughput achievable by HDFS. Our profiling shows that the `sync` calls to disk at the HDFS NameNode are still the bottleneck of the system. The reason for this 60% performance loss is that the region servers perform many additional directory operations, other than simply creating and closing files. For example, when a log file is garbage-collected, the region

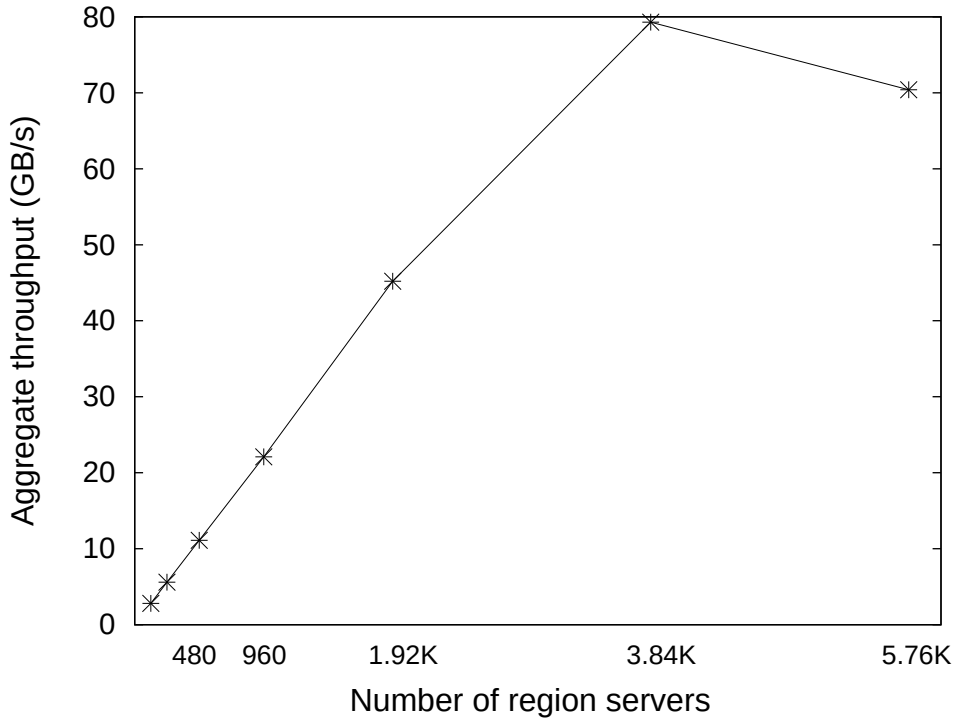

Figure 4.7: HBase throughput scalability.

server first moves it to an "old log" directory as a backup and only deletes it after some time has elapsed.

In Figure 4.7 each region server has 16GB of memory and holds 10 regions: since the default maximum size of a region is 200MB, all data can be cached in memory. Our next experiment evaluates how the performance of HBase is affected when we decrease the memory size per region. As shown in Figure 4.8, HBase throughput drops significantly when the number of regions per GB of memory exceeds 7, which translates to about 150MB of memory per region. In other words, in order for HBase to work efficiently in a large-scale deployment, each region server must be equipped with a considerably large amount of memory: enough to hold at least $\frac{3}{4}$ of its on-disk

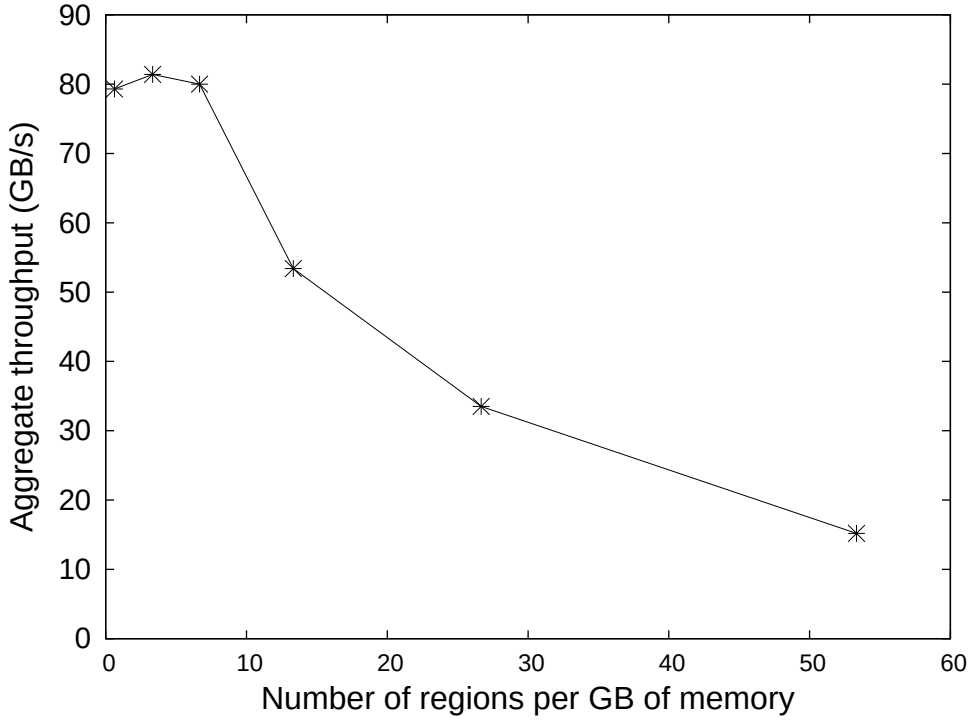

Figure 4.8: HBase aggregate throughput as the number of regions per GB of memory changes.

data. The reason for this performance drop is that region servers flush their regions to HDFS files when their memory usage exceeds a certain threshold. If the number of regions per GB of memory is high, this will create a large number of small files on HDFS, which stresses the HDFS NameNode. Resolving this problem requires a significant redesign of HBase, which is beyond the scope of this dissertation. Note that this performance drop is only observed at large scales, since small deployments cannot generate enough load to saturate the HDFS NameNode.

Our last experiment explores the effect of writing small values on the colocation ratio achievable in Exalt (Figure 4.9). Not surprisingly, Exalt achieves high

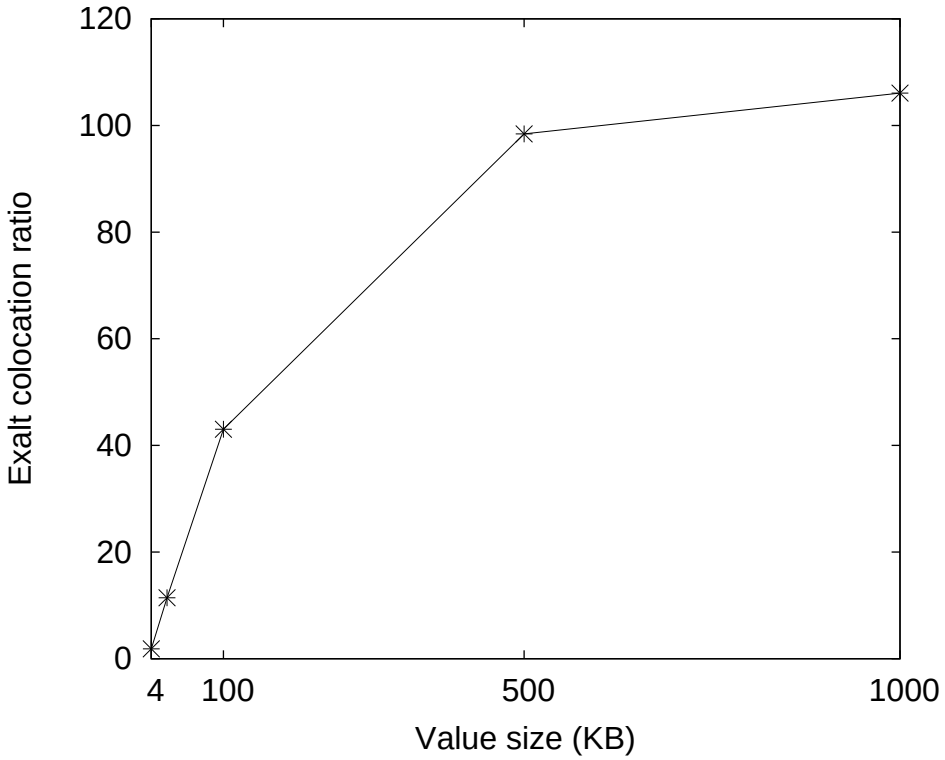

Figure 4.9: Colocation ratio of Exalt.

colocation ratios when the value sizes are large (around 500KB), but does not fare equally well for small values. It is worth noting that the achievable colocation ratio for a given workload is not infinite; eventually CPU utilization becomes the bottleneck. For HBase, this happens at a colocation ratio of about 110.

4.5.3 Case Study: Cassandra

Cassandra [60] borrows many elements from Amazon's Dynamo storage system. Unlike HDFS and HBase, Cassandra does not rely on a single metadata node to manage namespace and membership. Instead, Cassandra incorporates a distributed hashtable (DHT) protocol for these tasks, eliminating the single scalability bottle-

neck. Although one might expect this design choice to result in better scalability, our experience applying Exalt to Cassandra shows that this expectation is unfounded: the scalability problems of Cassandra appear at the scale of hundreds of nodes, even before we turn on Tardis compression.

We find two major problems preventing Cassandra from scaling up to tens of thousands of nodes. First, if multiple nodes concurrently join an existing Cassandra cluster, there is a non-negligible possibility that some of them may fail. This problem was confirmed by the developer of Cassandra behind the pseudonym *geekatcmu* who, when asked about the issue, wrote to us that: "you have to wait for each node to bootstrap before starting the next one" [24]. As a result, starting a cluster with tens of thousands of nodes may take prohibitively long, both in practice and in experiments, because a node usually takes several minutes to stabilize. This problem is actually rooted in the design of DHTs, where it is usually hard to provide a consistent namespace when a large number of nodes are joining or leaving.

The second problem is that the number of threads per node grows quickly with the scale of the cluster. This is because, in the current implementation, each Cassandra node creates four sockets—incoming and outcoming sockets for both data and metadata streams—to every other Cassandra node, and assigns a separate thread to each socket. In our experiments, the number of threads created quickly hit the system limit, which we didn't have sufficient privileges to change. Even if we had, the large number of threads would have created a memory problem: given that the stack of a thread takes at least 128KB[4] of memory in the JVM, in a cluster of 10,000 nodes, each node would need at least 5GB of memory just for creating threads. This problem may be addressed with an implementation using Non-blocking I/O (NIO), which creates a single thread for all sockets.

In sum, for the current implementation of Cassandra, trying to deploy a

[4]This number may vary depending on hardware architecture and operating system.

cluster with more than 1000 nodes is not suggested, as the developer of Cassandra behind the pseudonym *geekatcmu* remarked when we asked him to comment on our findings [24]: indeed, the scalability of Cassandra is so limited that these problems can be observed even without the help of Exalt.

4.6 Conclusion

Exalt is a library that gives back to researchers the ability to evaluate the scalability of large storage systems. Exalt is based on the Tardis compression scheme, which leverages a specific data format to achieve efficient compression and high degrees of colocation, which in turn allows researchers to perform large-scale experiments on as few as one hundred machines. We have used Exalt to identify several performance problems in HDFS and HBase. Fixing these problems allowed the system to significantly increase its maximum achievable throughput.

Chapter 5

Related Work

5.1 Separating data and metadata

Separating data and metadata is an old but effective idea adopted by multiple systems for different goals. This section only describes how it is used in storage systems to provide better robustness.

Many storage systems apply stronger protection to metadata and weaker protection to data because metadata, if damaged, could potentially cause any data, even the whole storage system, to become unavailable or corrupted; the effect of damaged data, however, is usually contained in the data item (e.g. a file) itself. Therefore, local filesystems such as the EXT series [2] and ZFS [88] keep multiple copies of the superblock and inodes on disks while keeping fewer copies of data; distributed storage systems such as Farsite [5] and Windows Azure Storage [23] apply strong replication (BFT and Paxos respectively) to its namespace metadata while using primary backup to minimize the cost of data. These systems usually achieve stronger guarantees on metadata but weaker guarantees on data. My dissertation, however, shows that with properly designed metadata, data as well can also be protected with strong guarantees and little additional cost.

Paris et al. [85] reduce the storage overhead of voting using volatile witnesses. Yin et al. [109] separate agreement from execution to reduce the number of execution nodes required for Byzantine replication, showing that while $3f + 1$ nodes are still required for agreement, $2f + 1$ nodes are enough for execution, and Clement et al. [31] refine these techniques. Gnothi and Salus both adopt similar ideas but with further improvements: Gnothi shows that the replication cost of execution in the failure free case can be further reduced to approximately $f+1$ by performing partial replication of data and full replication of metadata; Salus' active storage protocol shows that the replication cost of agreement can also be reduced to $f + 1$ under certain conditions (e.g. single writer per volume).

5.2 Robustness techniques

Three techniques are commonly used to protect a storage system against failures: replication, end-to-end checks, and erasure coding.

5.2.1 Replication

Tolerating omission failures Replication techniques used to tolerate omission failures can be classified as either synchronous or asynchronous.

In synchronous replication [20, 21, 34], a primary replica provides the service to the clients, and if the primary replica fails, a backup replica takes over and continues to provide service. It takes $f+1$ replicas to tolerate f crash failures. There are three main disadvantages to synchronous primary backup [19]: 1) its correctness is not guaranteed when there are timing errors caused by network partitions or server overloading, since these faults can cause replicas to diverge; 2) to minimize correctness issues, the system must be configured with conservative timeouts that can hurt availability; 3) read throughput is limited by the capability of a single machine, since only the primary replica processes requests.

Asynchronous replication does not assume an upper bound on network latency or node response time, and hence can ensure correctness even in the face of relatively rare events like server overload, network overload, or network partitions. The traditional approach to asynchronous replication involves a Replicated State Machine (RSM), in which a consensus protocol guarantees that each correct replica receives the same sequence of requests and in which each replica is a deterministic state machine.

Paxos [61, 62] is representative of the asynchronous RSM approach, which requires $2f + 1$ replicas to tolerate f crash failures. Paxos guarantees safety (all correct replicas receive the same sequence of requests) at all times and guarantees liveness (the system can make progress) when the network is available and node actions and message delivery are timely. Paxos uses timeouts internally, but it does not depend on their accuracy for safety and can adjust timeouts dynamically for liveness.

The standard Paxos protocol executes every request on each of the $2f + 1$ replicas, with costs (in bandwidth, storage space, etc.) higher than synchronous replication. Much work has been done to reduce the cost of Paxos: Gaios does not log reads, executes them on only one replica, and nonetheless guarantees linearizabilty by adding new messages to the original Paxos protocol [19]. ZooKeeper [54] includes a fast read protocol that executes on a single replica, but it does not provide Paxos's linearizability guarantee.

On-demand instantiation (ODI) [63] reduces write costs by executing requests on a preferred quorum of $f + 1$ replicas. If one of the active replica fails, a backup replica is activated, but before it can start processing any request it must be initialized by fetching the current value of all replicated state. In storage systems with large amounts of data, this approach does not scale, as the system can be unavailable for hours while it transfers terabytes of data.

Falcon [67] uses an accurate failure detector to eliminate the necessity of asynchronous replication, but it relies on the availability of all the network switches, an assumption that may not always hold in today's datacenters: a rack of machines together with its switch may be turned off for maintenance or fail unexpectedly, and large-scale storage systems should be designed to remain available in this case [42, 44, 77, 95]. Indeed, the authors of Falcon explicitly acknowledged the significant technical challenge involved in network failure localization [67].

Tolerating arbitrary failures Byzantine Fault Tolerance (BFT) replication is the standard technique to tolerate arbitrary failures. It can also be classified as either synchronous or asynchronous, and their relationship is similar to the pair described in the previous paragraph.

Synchronous BFT systems [16,64] take $3f+1$ replicas to tolerate f arbitrary failures, while asynchronous BFT systems [26,31,59] also take $3f+1$ replicas ($2f+1$ for execution) but, similar to Paxos, can only guarantee liveness during synchronous intervals. Several BFT systems [4,33] incorporate more replicas to optimize latency or throughput.

On-demand instantiation (ODI) is also applied in BFT techniques [108], but it suffers from the same problem that if a replica fails, the system is unavailable for a long time to wait for the data copy to complete. Distler et al. [38] propose to alleviate this problem by replaying a per-object log on demand, but again this approach is not appropriate for replicating applications with large amounts of state, because its logs and snapshots are on a per-object basis; to reduce overhead, per-object garbage collection is performed infrequently, once every 100 updates, which means that the system stores 100 copies of each object at each replica.

5.2.2 End-to-end checks

ZFS [3] incorporates an on-disk Merkle tree to protect the file system from disk corruptions. SFSRO [43], SUNDR [69], Depot [73], and Iris [96] also use end-to-end checks to guard against faulty servers. However, none of these systems is designed to scale to thousands of machines, because, to support multiple clients sharing a volume, they depend on a single server to update the Merkle tree. Instead, Salus is designed for a single client per volume, so it can rely on the client to update the Merkle tree and make the server side scalable. We do not claim this to be a major novelty of Salus; we see this as an example of how different goals lead to different designs.

We are not aware of any end-to-end verification techniques that can support multiple writers while achieving strong consistency, scalability, and end-to-end verification for read requests. One can tune Salus to support multiple writers by either using a single server to serialize requests to a volume as shown in SUNDR [69], which of course hurts scalability, or by using weaker consistency models like Fork-Join-Casual [73] or fork* [40].

End-to-end checks alone only provide safety guarantees but do not provide any durability or availability guarantees: an error can be detected by end-to-end checks, but how to recover from such an error remains unknown. That is why in distributed systems, end-to-end checks are often used in combination with replication to provide all the desired properties, and Salus adopts the same principle.

5.2.3 Erasure coding

Erasure coding [37,51,53] is another popular technique to protect data in distributed systems. It splits the raw data into multiple blocks and then codes them into a new set of blocks so that as long as a certain number of the new blocks survive the failures, the raw data can be reconstructed. Compared to replication, erasure coding

makes different tradeoffs: first, erasure coding uses more CPU resource, but usually requires less storage space to provide the same level of durability guarantee; second, erasure coding usually requires more network bandwidth to recover a lost block since many blocks need to be read to reconstruct the lost block; third, replication can usually provide better read throughput because reads can be directed to any of the replicas, while erasure coding doesn't have this advantage. All such tradeoffs make erasure coding an attractive option for cold data [57,68]—data that is written once and rarely accessed in the future—for which space is more of a concern than performance. This dissertation mainly focuses on hot replicated data, but several of the techniques we discuss, such as Salus' pipelined commit protocol and end-to-end verification, do not depend on how data is protected and thus should work with storage systems that use erasure coding.

5.3 Scalability techniques

5.3.1 Scalable and consistent storage

Sharding—breaking a storage space into multiple units (e.g. blocks, files, and key-value pairs) and assigning these units to different servers—is used in almost any large storage systems to scale them to more than thousands of servers. One of the key challenges is how to maintain membership, tracking which servers a data unit has been assigned to.

Many large-scale storage systems [5, 12, 23, 27, 44, 70] rely on a single (maybe replicated) metadata server to keep membership information. This approach is easy to design and implement, but suffers from the scalability problem that the single metadata server can become the bottleneck. On the other hand, systems like Cassandra [60] rely on a distributed hashtable (DHT) to maintain membership, eliminating the single bottleneck but suffering from another problem that the hashtable might

become inconsistent when a large number of nodes are joining or leaving. In the middle ground, several systems [10,41,107] use a group of metadata servers, instead of only one, to maintain membership, making a tradeoff between complexity and scalability.

Few such systems are designed to tolerate arbitrary node failures: while these systems usually use checksums to safeguard data written on disk, a memory corruption or a software glitch can lead to the loss of data in these systems (Section 3.4.1). In contrast, Salus is designed to be robust (safe and live) even if nodes fail arbitrarily.

5.3.2 Evaluating scalability

As we mentioned earlier, two common approaches to evaluating the scalability of large storage systems are using extrapolation and stub components. For example, extrapolation is used, among others, in RAMCloud [84], Spanner [32], and Salus [106], while the stub approach is used in HDFS [94, 95]. Section 4.1 discusses these approaches in detail, so we do not discuss them further here.

Several tools have been proposed to address the gap between the size of the experiments that researchers would like to run and the resources available to them.

In DieCast [48] this experimental gap is addressed using time dilation [49]. DieCast runs multiple processes inside virtual machines on a single host and slows down each process by a constant factor. It compensates for this slow-down by multiplying the measured throughput by the same factor. DieCast can achieve some degree of colocation when CPU utilization is the bottleneck, but does nothing to reduce the large amount of disk space necessary to evaluate large-scale storage systems.

The system that comes closer to addressing the experimental gap for storage systems is David [6]. David leverages the observation that to evaluate a local file system it is not necessary to store the actual data. Thus, David only stores the file

system's metadata: the data is simply discarded. This technique allows David to evaluate local file systems of much larger size than that of the local disk on which they are run. Unfortunately, this approach cannot be easily applied to distributed storage services. For example, when users write a key-value pair to HBase, the region server adds a timestamp and a region identifier to the write request and stores this metadata, together with the users' data, on the local file system of an HDFS DataNode. Since data and metadata look indistinguishable to the HDFS layer, David would discard metadata critical for the correct operation of the system.

Memulator [46] emulates nonexistent storage components by storing data in memory and accurately predicting how long each operation takes. Its purpose is to test the behavior of the system on devices that the researchers do not have access to. Unlike Exalt, it does not save any resource usage, which makes it not applicable to our goal.

Finally, simulation is a technique used by several systems to evaluate the performance of large-scale deployments. The approaches vary from disk simulation [98], network simulation [79, 83], to simulation of scheduling and checkpointing in large platforms [103, 110]. A well-known drawback of simulation is that its results are only as good as its model of how the system works. Unfortunately, as systems grow in complexity, coming up with a model that accurately captures all their features becomes prohibitively hard.

There exist several compression algorithms [29, 76, 80, 111, 112] one may consider using in our context. However, all these algorithms are designed to be general-purpose and as such they need to scan all the input bytes. Tardis, on the other hand, owes its efficiency largely to the fact that it does not have to scan most of the input bytes.

Chapter 6

Conclusion

This dissertation shows that, by utilizing the idea of separating data from metadata, a storage system can achieve strong robustness guarantees with little impact on efficiency and scalability. This dissertation demonstrates the power of this idea by applying it to three very different systems.

I have learned several lessons during my work. First, the general belief that stronger protection is more expensive, which is often proved to be true in a theorem, may not be an insuperable obstacle when coming to real systems: real systems are often complex and may deviate from the general model to which the theorem applies in subtle ways, opening up new opportunities for optimizations. Finding such opportunities usually requires a deep understanding of the theorem itself, and in particular its assumptions.

Second, when calculating the cost of a protection technique, what really matters is its cost in the failure-free case, because this is the most common scenario: optimizing cost in the failure-free case is the key in both Gnothi and Salus, and their costs in the presence of failures are actually not different from those of previous works. Although this is not a new insight, two things are worth noting: first, the replication thresholds proved in different theorems (e.g. $2f + 1$ for asynchronous

replication) often apply to worst-case scenarios, which might be misleading to some extent. Second, optimizing the failure-free case should not significantly hurt the availability of the system when failures occur: ensuring this property is exactly the key factor that distinguishes Gnothi from previous works.

Finally, although the size of metadata is small, how to process metadata still requires careful thoughts. For example, caching all metadata in memory can cause memory overhead while storing it to disks may cause random disk accesses: neither option is desirable. My experience in working on this dissertation suggests that identifying an effective way to reduce the overhead of processing metadata usually requires a significant engineering effort.

In conclusion, strong protection of data does not have to be expensive. Many systems that are making a tradeoff between robustness and scalability can actually enjoy the benefits of both, as long as we can design and protect metadata properly.

Bibliography

[1] Amazon S3 Availability Event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`.

[2] Ext4 (and Ext2/Ext3) Wiki. `https://ext4.wiki.kernel.org/index.php/Main_Page`.

[3] ZFS On-Disk Specification. Technical report, Sun Microsystems, 2006.

[4] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-Scalable Byzantine Fault-Tolerant Services. In *SOSP*, 2005.

[5] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, 2002.

[6] Nitin Agrawal, Leo Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Emulating Goliath Storage Systems with David. In *FAST*, 2011.

[7] Amazon Elastic Block Store (EBS). `http://aws.amazon.com/ebs/`.

[8] Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`.

[9] Darrell Anderson, Jeff Chase, and Amin Vahdat. Interposed Request Routing for Scalable Network Storage. In *OSDI*, 2000.

[10] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, 14(1):41–79, Feb 1996.

[11] Apache Hadoop FileSystem and its Usage in Facebook. `http://cloud.berkeley.edu/data/hdfs.pdf`.

[12] Apache HBASE. `http://hbase.apache.org/`.

[13] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage*, 4(3):8:1–8:28, November 2008.

[14] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *SIGMETRICS*, 2007.

[15] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, 2011.

[16] Rida A. Bazzi. Synchronous Byzantine quorum systems. *Distributed Computing*, 13(1):45–52, 2000.

[17] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *OSDI*, 2010.

[18] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the Correctness of Memories. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, 1991.

[19] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos Replicated State Machines as the Basis of a High-Performance Data Store. In *NSDI*, 2011.

[20] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault Tolerance. *ACM Transanctions on Computer Systems*, 14(1):80–107, February 1996.

[21] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary-Backup Protocols: Lower Bounds and Optimal Implementations. In *CDCCA*, 1992.

[22] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[23] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[24] Private communication with Cassandra developers *geekatcmu*.

[25] Miguel Castro and Barbara Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *OSDI*, 2000.

[26] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transanctions on Computer Systems*, 20(4):398–461, November 2002.

[27] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[28] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *FAST*, 2012.

[29] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32:396–402, 1984.

[30] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (Limited) Power of Non-equivocation. In *PODC*, 2012.

[31] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riché. UpRight Cluster Services. In *SOSP*, 2009.

[32] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.

[33] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba

Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *OSDI*, 2006.

[34] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*, 2008.

[35] Jeffrey Dean. Evolution and Future Directions of Large-scale Storage and Computation Systems at Google. Kenote talk at 2010 Symposium on Cloud Computing (SOCC), 2010.

[36] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[37] Alexandros G. Dimakis, P. Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, September 2010.

[38] Tobias Distler and Rüdiger Kapitza. Increasing Performance in Byzantine Fault-Tolerant Systems with On-Demand Replica Consistency. In *Eurosys*, 2011.

[39] Private communication with Facebook engineers Siying Dong and Liyin Tang.

[40] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *OSDI*, 2010.

[41] Andrew Fikes. Storage Architecture and Challenges. Faculty summit, 2010.

[42] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *OSDI*, 2010.

[43] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and Secure Distributed Read-only File System. *ACM Transanctions on Computer Systems*, 20(1):1–24, February 2002.

[44] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, 2003.

[45] Jim Gray. Notes on Data Base Operating Systems. In *Advanced Course: Operating Systems*, pages 393–481, 1978.

[46] John Linwood Griffin, Jiri Schindler, Steven W. Schlosser, John S. Bucy, and Gregory R. Ganger. Timing-accurate Storage Emulation. In *FAST*, 2002.

[47] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput Optimal Total Order Broadcast for Cluster Environments. *ACM Transactions on Computer Systems*, 28(2):5:1–5:32, July 2010.

[48] Diwaker Gupta, Kashi Venkatesh Vishwanath, and Amin Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *NSDI*, 2008.

[49] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. In *NSDI*, 2006.

[50] Gzip. http://www.gzip.org/.

[51] HDFS RAID . `http://wiki.apache.org/hadoop/HDFS-RAID`.

[52] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[53] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure Coding in Windows Azure Storage. In *USENIX ATC*, 2012.

[54] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*, 2010.

[55] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are Disks the Dominant Contributor for Storage Failures?: A Comprehensive Study of Storage Subsystem Failure Characteristics. *ACM Transactions on Storage*, 4(3):7:1–7:25, November 2008.

[56] Private communication with Manos Kapritsos.

[57] Rini T. Kaushik and Milind Bhandarkar. GreenHDFS: Towards an Energy-conserving, Storage-efficient, Hybrid Hadoop Compute Cluster. In *HotPower*, 2010.

[58] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *IISWC*, 2008.

[59] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *SOSP*, 2007.

[60] Avinash Lakshman and Prashant Malik. Cassandra – A Decentralized Structured Storage System. In *LADIS*, 2009.

[61] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[62] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, December 2001.

[63] Leslie Lamport and Mike Masa. Cheap Paxos. In *DSN*, 2004.

[64] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[65] Butler W. Lampson and Howard E. Sturgis. Crash Recovery in a Distributed Data Storage System. Xerox PARC Research Report, 1979.

[66] Edward Lee and Ramohan A. Thekkath. Petal: Distributed Virtual Disks. In *ASPLOS*, 1996.

[67] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. SOSP, 2011.

[68] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and Analysis of Large-scale Network File System Workloads. In *USENIX ATC*, 2008.

[69] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure Untrusted Data Repository (SUNDR). In *OSDI*, 2004.

[70] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *SOSP*, 2011.

[71] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Liuba Shrira. Replication in the Harp File System. In *SOSP*, 1991.

[72] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *OSDI*, 2004.

[73] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud Storage with Minimal Trust. In *OSDI*, 2010.

[74] Mike Mammarella, Shant Hovsepian, and Eddie Kohler. Modular Data Storage with Anvil. In *SOSP*, 2009.

[75] Ralph Merkle. Protocols for Public Key Cryptosystems. In *Symposium on Security and Privacy*, 1980.

[76] Alistair Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, November 1990.

[77] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's Warm BLOB Storage System. In *OSDI*, 2014.

[78] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *FAST*, 2008.

[79] Network Emulation with the NS Simulator. `http://www.isi.edu/nsnam/ns/ns-emulation.html`.

[80] Craig G. Nevill-Manning and Ian H. Witten. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7(1):67–82, July 1997.

[81] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, Cells and Platters: An Empirical Analysisof Hardware Failures on a Million Consumer PCs. In *Eurosys*, 2011.

[82] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat Datacenter Storage. In *OSDI*, 2012.

[83] NS-2. `http://nsnam.isi.edu/nsnam/`.

[84] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[85] Jehan-François Pâris and Darrell D. E. Long. Voting with regenerable volatile witnesses. In *ICDE*, 1991.

[86] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andr Barroso. Failure Trends in a Large Disk Drive Population. In *FAST*, 2007.

[87] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *SOSP*, 2005.

[88] A. Rich. ZFS, Sun's Cutting-Edge File System. Technical report, Sun Microsystems, 2006.

[89] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems (Reprint). *Communications of the ACM*, 26(1):96–99, 1983.

[90] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[91] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[92] Bianca Schroeder and Garth A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *FAST*, 2007.

[93] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-scale Field Study. In *SIGMETRICS*, 2009.

[94] Konstantin Shvachko. HDFS scalability: the limits to growth. http://c59951.r51.cf2.rackcdn.com/5424-1908-shvachko.pdf.

[95] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[96] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: A Scalable Cloud File System with Efficient Integrity Checks. In *ACSAC*, 2012.

[97] Texas Advanced Computing Cener (TACC). https://www.tacc.utexas.edu/.

[98] The DiskSim Simulation Environment. http://www.pdl.cmu.edu/DiskSim/.

[99] HDFS Usage in Yahoo! `http://www.aosabook.org/en/hdfs.html`.

[100] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *SOSP*, 1997.

[101] The Truman Show. http://www.imdb.com/title/tt0120382/.

[102] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, 2004.

[103] Ke Wang, Kevin Brandstatter, and Ioan Raicu. SimMatrix: SIMulator for MAny-Task Computing Execution fabRIc at eXascale. In *HPC*, 2013.

[104] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication. In *USENIX ATC*, 2012.

[105] Yang Wang, Manos Kapritsos, Lorenzo Alvisi, and Mike Dahlin. Exalt: Empowering Researchers to Evaluate Large-Scale Storage Systems. In *NSDI*, 2014.

[106] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the Salus scalable block store. In *NSDI*, 2013.

[107] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *OSDI*, 2006.

[108] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. ZZ and the Art of Practical BFT. In *Eurosys*, 2011.

[109] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *SOSP*, 2003.

[110] Dongfang Zhao, Da Zhang, Ke Wang, and Ioan Raicu. Exploring Reliability of Exascale Systems Through Simulations. In *HPC*, 2013.

[111] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[112] Jacob Ziv and Abraham Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.

Vita

Yang Wang was born in Nanjing, a beautiful ancient city in China. He lived there until he graduated from Nanjing Foreign Language School in 2001. He attended Tsinghua University in Beijing, where he received his bachelor's degree in 2005 and master's degree in 2008, both in Computer Science and Technology. In the Fall of 2008, he joined the Department of Computer Science at the University of Texas at Austin as a Ph.D. student.

Permanent Address: Room 307, Building 4,
No. 186 Fenghuangxijie,
Nanjing, Jiangsu 210036,
P.R. China

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.