Robustness in the Salus scalable block store

Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam,
Lorenzo Alvisi, and Mike Dahlin
The University of Texas at Austin

Abstract: This paper describes Salus, a block store that seeks to maximize simultaneously both scalability and robustness. 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.). Such increased protection does not come at the cost of scalability or performance: indeed, Salus often actually outperforms HBase (the codebase from which Salus descends). For example, Salus' active replication allows it to halve network bandwidth while increasing aggregate write throughput by a factor of 1.74 compared to HBase in a well-provisioned system.

1 Introduction

The primary directive of storage—not to lose data—is hard to carry out: disks and storage sub-systems can fail in unpredictable ways [7, 8, 18, 23, 34, 37], and so can the CPUs and memories of the nodes that are responsible for accessing the data [33, 38]. Concerns about robustness become even more pressing in cloud storage systems, which appear to their clients as black boxes even as their larger size and complexity create greater opportunities for error and corruption.

This paper describes the design and implementation of Salus,[1] a scalable block store in the spirit of Amazon's Elastic Block Store (EBS) [1]: 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 [13, 16, 31], but these systems are not scalable—they require each correct node to process at least a majority of updates. Conversely, scalable distributed storage systems [3, 4, 6, 11, 14, 20, 25, 30, 43] typically protect some subsystems like disk storage with redundant data and checksums, but fail to 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), and leverages an architecture similar to scalable key-value stores like Bigtable [14] and HBase [6] towards scaling these guarantees to thousands of machines and tens of thousands of disks.

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 §5.1).

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.

Pipelined commit. Salus' new pipelined commit protocol allows writes to proceed in parallel at multiple disks but, by tracking the necessary dependency information during failure-free execution, 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 computation node from corrupting data, Salus replicates both the storage and the computation layer. Salus applies an update to the system's persistent state only if the update is agreed upon

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

by *all* of the replicated computation nodes. 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 a Merkle tree [32] 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 [19, 26, 31, 41], 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.

2 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.

To minimize cost, a typical server in existing storage deployments is relatively storage heavy, with a total capacity of up to 24 TB [5, 42]. We expect a storage server in a Salus deployment to have ten or more SATA disks and two 1 Gbit/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.

2.1 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 [35] 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 fail-

[2] More precisely, *ordered commit* (defined in §2.2) which for multiple clients implies FIFO-compliant linearizability.

ures, 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 [16], 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* [36] 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 [12] 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 [10].

2.2 Consistency model

To be usable as a virtual disk, Salus tries to preserve 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.

During normal operation (up to two commission or omission failures), Salus guarantees both freshness and a property we call *ordered-commit*: by the time a request *R* is committed, all requests that were received before *R* have committed. 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.

Under severe failures Salus provides the weaker *prefix semantics*: in these circumstances, a client that crashes and restarts may observe only a prefix of the committed writes; a tail of committed writes may be lost. This semantics is not new to Salus: it is the semantics familiar to every client that interacts with a crash-prone server that acknowledges writes immediately but logs them asynchronously; it is also the semantics to which every other geo-replicated storage systems we know of [11, 29, 31] retreats when failures put it under duresse. The reason is simple: while losing writes is always disappointing, prefix semantics has at least the merit of leaving the disk in a legal state. Still, data loss should be rare, and Salus falls back on prefix semantics only in the following scenario: the client crashes, one or more of the servers suffer at the same time a commission failure, and the rest of the servers are unavailable. If the client does not fail or at least one server is correct and available, Salus continues to guarantee standard disk semantics.

Salus mainly focuses on tolerating arbitrary failures 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 paper.

3 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, because it provides a key-value interface that can be easily modified to support a block store; second, because it has a large user base that includes companies such as Yahoo!, Facebook, and Twitter; and third because, unlike other successful large-scale storage systems with similar architectural features, such as Windows Azure [11] and Google's Bigtable/GFS [14, 20], HBase/HDFS is open source.

HDFS HDFS [39] is an append-only distributed file system. It stores the system metadata 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 [6] is a distributed key-value store. 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

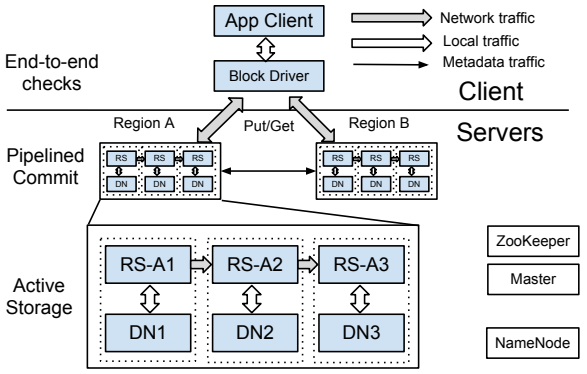

Fig. 1: The architecture of Salus. Salus differs from HBase in three key ways. First, Salus' block driver performs end-to-end checks to validate the GET reply. Second, Salus performs pipelined commit across different regions to ensure ordered commit. Third, Salus replicates region servers via active storage to eliminate spurious state updates. For efficiency, Salus tries to co-locate the replicated region servers with the replicated datanodes (DNs).

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

4 The design of Salus

The architecture of Salus, as Figure 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 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 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 §4.3): 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 §4.1, 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 §4.2): 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 data.

4.1 Pipelined commit

The goal of the pipelined commit protocol is to allow clients to concurrently issue requests to multiple regions, 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 at region 2. Untimely crashes, even transient ones, 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 now violates both standard disk semantics and the weaker prefix semantics; further, V is left in an invalid state that can potentially cause severe data loss [15, 35].

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 §5.2.4, performance would suffer significantly. Instead, we would like to achieve the good performance that comes with issuing multiple oustanding requests, without compromising the ordered-commit property. To achieve this goal, 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.

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 unless the previous request is also guaranteed to eventually commit. Similarly, during recovery, these sequence numbers are used to ensure that a consistent prefix of issued requests are recovered (§4.4).

Salus' apporach to ensure ordered-commit for GETs is simple. Like other systems before it [9], 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 2. The client, as in HBase, issues requests in batches. Unlike HBase, each client is allowed to issue multiple outstanding batches. Each batch is committed using a 2PC-like protocol [21, 24], consisting of the phases described below. Compared to 2PC, pipelined commit reduces the overhead of the failure-free case by eliminating the disk write in the commit phase and by pushing complexity to the recovery protocol, which is usually a good trade-off.

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.

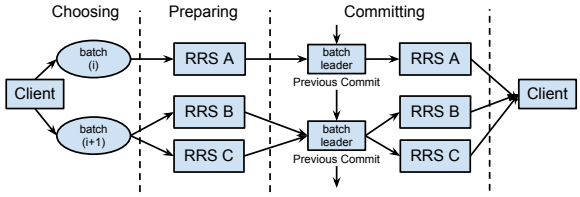

Fig. 2: Pipelined commit (each batch leader is actually replicated to tolerate arbitrary faults.)

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 request (which is now *prepared*) and sends its YES vote to 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.

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 (§4.4) ensures that all these prepared PUTs eventually commit without violating ordered-commit.

The pipelined commit protocol enforces ordered-

[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.

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

commit assuming the abstraction of (logical) region servers that are correct. It is the *active storage* protocol (§4.2) that, from physical region servers that *can* lose committed data and suffer arbitrary failures, provides this abstraction to the pipelined commit protocol.

4.2 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 a key limitation of existing scalable storage systems: 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.

The design of Salus 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.

Salus implements this basic principle using *active storage*. In addition to storing data, storage nodes in Salus also coordinate to attest data and perform checks to ensure that only correct and attested data is being replicated. Perhaps surprisingly, in addition to improving fault-resilience, active storage also enables us to improve performance by trading relatively cheap CPU cycles for expensive network bandwidth.

Using active storage, Salus can provide strong availability and durability guarantees: a data block with a quorum of size n will remain available and durable as long as no more than $n-1$ nodes fail. 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 two key ideas—(1) moving computation to data, and (2) using unanimous consent quorums—to ensure that active storage does not incur more network cost or storage cost compared to existing approaches that do not replicate computation.

4.2.1 Moving computation to data to minimize network usage

Salus implements active storage by blurring the boundaries between the storage layer and the compute layer. Existing storage systems [6, 11, 14] 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 block. Using this modified interface, Salus can efficiently implement active storage by colocating a compute 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 (§5.2.2)—an attractive trade-off for bandwidth starved data-centers. In particular, because a region server can now update the colocated datanode without requiring the network, the bandwidth overheads of flushing (§3) and compaction (§3) 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 (§5.2.2): 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.

4.2.2 Using unanimous consent to reduce replication overheads

To control the replication and storage overheads, we use unanimous consent quorums for PUTs. Existing systems replicate data to three nodes to ensure durability despite two permanent omission failures. Salus provides the same durability and availability guarantees despite two failures of either omission *or* commission without increasing the number of replicas. To tolrate f commission faults with just $f+1$ replicas, Salus requires the replicas to reach unanimous consent prior to performing any operation that updates the state and to store a certificate proving the legitimacy of the update.

Of course, 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 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

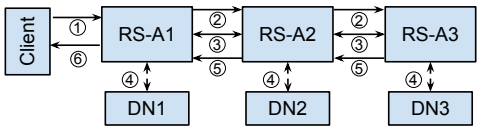

Fig. 3: Steps to process a PUT request in Salus using active storage.

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 §4.4.

4.2.3 Active storage protocol

To provide to the other components of Salus 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) issues a *proposal*, which is forwarded 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 shows how active storage refines the pipelined commit protocol for PUT requests. The PUT issued by a client is received by the primary region server as part of a sub-batch (①). Upon receiving a PUT, each replica validates it and forwards it down the chain of replicas (②). The region servers then agree on the location and order of the PUT in the append-only logs (③) and create a PUT-*log certificate* that attests to that location and order. Each region server sends the PUT and the certificate to its corresponding datanode to guarantee their persistence and waits for the datanode's confirmation (④) before marking the request as prepared. Each region server then independently contacts 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 PUT_SUCCESS certificate (⑤); on receiving ABORT the region servers generate instead a PUT_FAILED certificate. In either case, the primary then forwards the certificate to the client (⑥).

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.

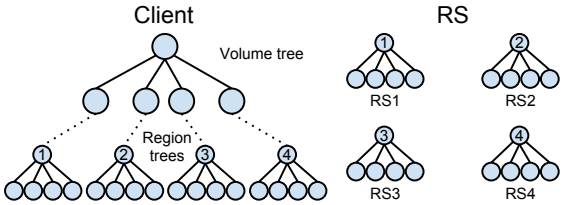

Fig. 4: Merkle tree structure on client and region servers

4.3 End-to-end verification

Local file systems fail in unpredictable ways [35]. 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 [19, 26, 31, 41], Salus' mechanism for end-to-end checks leverages Merkle trees 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 robustness, Salus keeps a copy of the volume tree stored distributedly 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 latest known hash for the root of the full volume tree, together with the sequence number of the PUT request that produced it.

Figure 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. On receiving a response, the client verifies it using the locally stored volume tree. If the check fails (because of a commission failure) or if the client times out (be-

cause 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(§4.4). 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 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 (§4.4). 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. If the two hashes match, then the client considers the mount to be complete; otherwise it reports an error indicating that a RRS is returning a potentially stale tree. In such cases, the client reports an error to the Master to trigger the replacement of the servers in the corresponding RRS, as described in §4.4.

4.4 Recovery

The recovery protocol ensures that, despite comission or permanent omission failures in up to f pairs of corresponding region servers and datanodes, Salus continues to provide the abstraction of a virtual disk with *standard disk semantics*, except in the extreme failure scenario in which the client crashes *and* one or more of the region server/datanode pairs of a region experience a commission failure *and* all other region server/datanode pairs of that region are unavailable: in this case, Salus' recovery protocol guarantees the weaker *prefix 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 §4.1 that every PUT for which the client received a PUT_SUCCESS must appear in the log of at least one correct replica in the region that processed that PUT. Hence, if a correct replica is available for each of the volume's regions, P_C will contain all PUT requests for which the client received a PUT_SUCCESS, thus guaranteeing standard disk semantics. If however, because of a confluence of commission and transient omission failures, the only available replicas in a region are those who have suffered commission failures, then the P_C that the recovery protocol collects may include only a prefix (albeit ordered-commit-compliant) of those PUT requests, resulting in the weaker prefix 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.

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

Fig. 5: Pseudocode for the recovery protocol.

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. Worse, because of transient omission failures, some region may temporarily have no operational correct replica when the recovery protocol attempts to collect logged PUTs.

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 5 describes the protocol that Salus uses to recover faulty datanodes and region servers. The first two phases describe the recovery of individual region logs, while the last two phases describe how the RRSs coordinate to identify commitable requests.

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 choose, among the instances (one for each operational datanode) of that region's old region logs, the longest available log that is *valid*. A log is *valid* if it is a prefix of PUT requests issued to that region. [5] We use the PUT-*log certificate* attached to each PUT record to separate *valid* logs from *invalid* ones: each region server independently replays the log and checks if each PUT record's location and order matches the location and order included in that PUT record's PUT-*log certificate*. Having found a valid log, the servers in the RRS agree on the longest prefix and advance to the next stage.

[5] Salus' approach for truncating logs is similar to how HBase manages checkpoints and is discussed in an extended TR [44].

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), [6] 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`). The goal of this phase is to ensures that all PUTs in the LCP are committed and available. The first half 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 (§4.3) to determine whether the current root of the volume tree matches the one stored in ZooKeeper during Phase 3.

As mentioned above, a confluence of commission and transient omission failures could cause a RRS to recover only a prefix of its region log. In that case, the above checks could fail, since some PUTs within the LCP could be ignored. If the checks fail, recovery starts again from Phase 1.[7] Note, however, that all the ignored PUTs must have been prepared and so, as long as the number of permanent omission or commission failures does not exceed f, a correct datanode will eventually become available and a consistent volume will be recovered.

5 Evaluation

We have implemented Salus by modifying HBase [6] and HDFS [39] 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 [16] to replicate NameNode, ZooKeeper, and Master.

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

[7] For a more efficient implementation that leverages version vectors, see [44].

Salus ensures *freshness, ordered-commit, and liveness* when there are no more than 2 failures within any RRS and the corresponding datanodes.	§5.1
Salus achieves comparable or better single-client throughput compared to HBase with slightly increased latency.	§5.2.1
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.	§5.2.2
Salus' overhead over HBase does not grow with the scale of the system.	§5.2.3

Fig. 6: Summary of main results.

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 6 summarizes the main results.

5.1 Robustness

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

Salus 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 determine 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 crash, temporarily or permanently, 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 7 summarizes our results.

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

Affected nodes	Faults	HBase		Salus	
		GET	PUT	GET	PUT
Client	Crash and restart	Fresh	Not ordered	Fresh	Ordered
DataNode	1 or 2 permanent crashes	Fresh	Ordered	Fresh	Ordered
	Corruption of 1 or 2 replicas of log or checkpoint	Fresh	Ordered	Fresh	Ordered
	3 arbitrary failures	Fresh*	Lost	Fresh*	Lost
Region server+DataNode	1 (for HBase) or 3 (for Salus) region server permanent crashes	Fresh	Ordered	Fresh	Ordered
	1 (for HBase) or 2 (for Salus) region server arbitrary failures that potentially affect datanodes	Corrupted	Lost	Fresh	Ordered
	3 (for Salus) region server arbitrary failures that potentially affect datanodes	-	-	Fresh*	Lost
Client+Region server+DataNode	Client crashes and restarts, 1 (for HBase) or 2 (for Salus) region server arbitrary failures causing the corresponding datanodes to not receive a suffix of data	Corrupted	Lost	Fresh	Ordered

Fig. 7: 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.

when barriers are frequent (see §5.2.4). 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 7: 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.

5.2 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 bandwidth 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 and validation. Compared to the cost of disk-accesses for data, however, we expect these ovrheads to be modest.

This section quantifies these tradeoffs using sequential- and random-, read and write microbenchmarks. 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 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 an 16-core AMD Opteron 4282 @3.0GHz, 64GB of memory, and 10 WDC 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 [2]. The detailed configuration is shown in §5.2.3.

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.

5.2.1 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 overhead of checksum computation and message transfer of Salus increases 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 8 shows the single client throughput. For sequential read, Salus outperforms the HBase system with a speedup of 2.5. The reasons are that Salus' active replication's three region servers increase parallelism for reads and 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 write and random write, 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 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 (likely because it is utilizing more disk arms), and that it substantially outperforms EBS's random write throughput (likely because it transforms random writes into sequential ones.)

Figure 9 shows the 90th-percentile latency for random reads and writes. In both cases, Salus' latency is within two or three milliseconds or of HBase's. This is reasonable considering Salus' additional work to perform Merkle tree calculation, certificate generation and validation, and network transfer. One thing should be noted

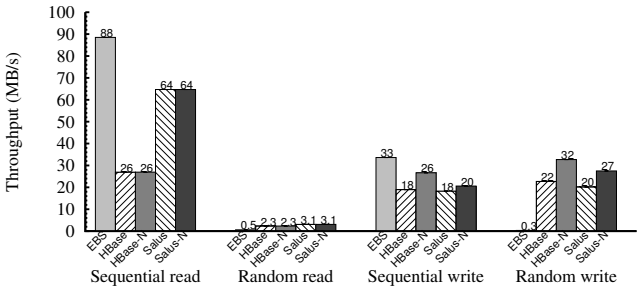

Fig. 8: Single client throughput on small nodes. HBase-N and Salus-N disable compactions. EBS's numbers are measured on different hardwares and are included for reference.

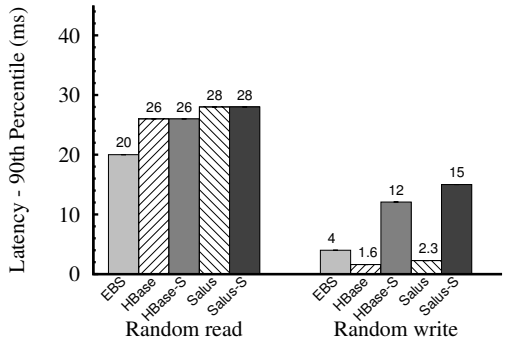

Fig. 9: Single client latency on small nodes. HBase-S and Salus-S enable sync. EBS's numbers are measured on different hardwares and are included for reference.

about the random write latency experiment: the HBase datanode does not call *sync* when performing disk write and that's why its write latency is small. This may be a reasonable design decision when the probability of three simultaneous crashes is small [28]. 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.

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. This is 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.

5.2.2 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 ac-

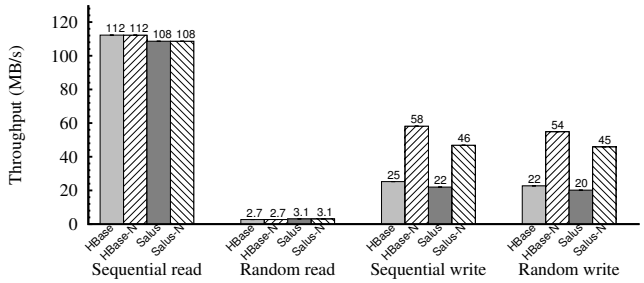

Fig. 10: Aggregate throughput on small nodes. HBase-N and Salus-N disable compactions.

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

Fig. 11: Aggregate sequential write throughput and network bandwidth usage with fewer server machines but more disks per machine.

tive replication of region servers, introduced to improve robustness, can reduce network bandwidth and significantly improve performance when the total disk bandwidth exceeds the aggregate network bandwidth.

Figure 10 reports experiments on our small-server testbed with nine nodes acting as combined region server and datanode servers and we increase the number of clients until the throughput does not increase.

For sequential read, both systems can 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 slowdown of Salus versus HBase for sequential and random writes is respectively of 11.1% to 19.4% and significantly lower 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 in our system because disk and network bandwidth are comparable. Even so, we find this to be an acceptable price for the stronger guarantees provided by Salus.

Figure 11 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 11 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;

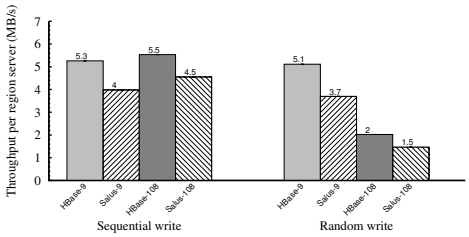

Fig. 12: Write throughput per server with 9 servers and 108 servers (compaction disabled).

logging, flushing, and compaction are performed locally. The actual number is slightly higher than 2, because of Salus's 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.

5.2.3 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 [2] 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 valuable 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 experi-

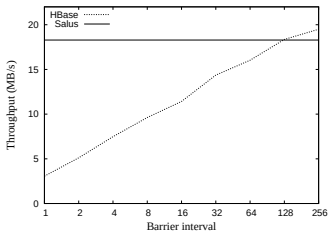

Fig. 13: Single client sequential write throughput as the frequency of barriers varies.

ments 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 12 shows the per-server throughput of the sequential and random write workloads in configuration with 9 and 108 servers. For the sequential write workload, the throughput per server remains almost unchanged in both HBase and Salus as we move from 9 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 (28%) in both the 9-server and the 108-server experiments, the letter that Salus' overhead does not grow with the scale of the system.

5.2.4 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 13 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 1 and 14 MB/s with a batch size of 32.

6 Related work

Scalable and consistent storage. Many existing systems provide the abstraction of scalable distributed storage [6, 11, 14, 27] with strong consistency. Unfortunately, these systems do not tolerate arbitrary node failures. While these systems 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 (§ 5.1). In contrast, Salus is designed to be robust (safe and live) even if nodes fail arbitrarily.

Protections in local storage systems Disks and storage sub systems can fail in various ways [7, 8, 18, 23, 34, 37], are so can memories and CPUs [33, 38] with disastrous consequences [35]. Unfortuantely, end-to-end protection mechanisms developed for local storage systems [35, 41] are inadequate for protecting the full path from a PUT to a GET in complex systems like HBase.

End-to-end checks. ZFS [41] incorporates an on-disk Merkle tree to protect the file system from disk corruptions. SFSRO [19], SUNDR [26], Depot [31], and Iris [40] 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.

BFT systems. While some distributed systems tolerate arbitrary faults (Depot [31], SPORC [17], SUNDR [26], BFT RSM [13, 16]), they require a correct node to observe all writes to a given volume, preventing a volume from scaling with the number of nodes.

Supporting multiple writers. We are not aware of any system that can support multiple writers while achieving ordered-commit, 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 [26], which of course hurts scalability, or by using weaker consistency models like Fork-Join-Casual [31] or fork* [17].

7 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.

Acknowledgements

We thank our shepherd Arvind Krishnamurthy and the anonymous reviewers for their insightful comments. This work was supported in part by NSF grants CiC-FRCC-1048269 and CSR-0905625.

References

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

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

[3] D. Anderson, J. Chase, and A. Vahdat. Interposed Request Routing for Scalable Network Storage. In *OSDI*, 2000.

[4] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, 14(1):41–79, Feb 1996.

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

[6] Apache HBASE. http://hbase.apache.org/.

[7] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. *TOS*, 2008.

[8] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *SIGMETRICS*, 2007.

[9] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li. Paxos Replicated State Machines as the Basis of a High-Performance Data Store. In *NSDI*, 2011.

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

[11] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[12] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *OSDI*, 2000.

[13] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *TOCS*, 2002.

[14] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[15] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency Without Ordering. In *FAST*, 2012.

[16] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché. UpRight Cluster Services. In *SOSP*, 2009.

[17] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *OSDI*, 2010.

[18] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *OSDI*, 2010.

[19] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *TOCS*, 2002.

[20] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[21] J. Gray. Notes on Database Systems. IBM Research Report RJ2188, Feb. 1978.

[22] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[23] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *TOS*, 2008.

[24] B. Lampson and H. Sturgis. Crash Recovery in a Distributed System. Xerox PARC Research Report, 1976.

[25] E. Lee and R. A. Thekkath. Petal: Distributed virtual disks. In *ASPLOS*, 1996.

[26] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *OSDI*, 2004.

[27] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *SOSP*, 2011.

[28] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the Harp File System. In *SOSP*, 1991.

[29] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[30] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, 2004.

[31] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: cloud storage with minimal trust. In *OSDI*, 2010.

[32] R. Merkle. Protocols for public key cryptosystems. In *Symposium on Security and Privacy*, 1980.

[33] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs. In *Eurosys*, 2011.

[34] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST*, 2007.

[35] V. Prabhakaran, L. Bairavasundaram, N. Agrawal, H. Gunawi, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. IRON file systems. In *SOSP*, 2005.

[36] R. L. Rivest, A. Shamir, and L. M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems (Reprint). *CACM*, 1983.

[37] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *FAST*, 2007.

[38] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS*, 2009.

[39] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[40] E. Stefanov, M. van Dijk, A. Oprea, and A. Juels. Iris: A Scalable Cloud File System with Efficient Integrity Checks. In *ACSAC*, 2012.

[41] I. Sun Microsystems. ZFS on-disk specification. Technical report, Sun Microsystems, 2006.

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

[43] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *SOSP*, 1997.

[44] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the Salus scalable block store. Technical Report TR-12-24, The University of Texas at Austin, Department of Computer Science, 2012.