Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication

Yang Wang, Lorenzo Alvisi, and Mike Dahlin
The University of Texas at Austin
{*yangwang, lorenzo, dahlin*}*@cs.utexas.edu*

Abstract: This paper describes Gnothi, a block replication system that separates data from metadata to provide efficient and available storage replication. Separating data from metadata allows Gnothi to execute disk accesses on subsets of replicas while using fully replicated metadata to ensure that requests are executed correctly and to speed up recovery of slow or failed replicas.

Performance evaluation shows that Gnothi can achieve 40-64% higher write throughput than previous work and significantly save storage space. Furthermore, while a failed replica recovers, Gnothi can provide about 100-200% higher throughput, while still retaining the same recovery time and while guaranteeing that recovery eventually completes.

1 Introduction

An ideal storage system should provide good availability and durability, strong correctness guarantees, low cost, and fast failure recovery. Existing replicated storage systems make different trade-offs among these properties: 1) synchronous primary-backup systems [10, 13, 15] require $f+1$ replicas to tolerate f crash faults, but they risk data loss if there are timing errors; 2) asynchronous full replication systems [4, 5, 8, 18] use asynchronous agreement [20, 21] 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 [22, 29] 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 are promising for replicating services with small amounts of state, they are not well-suited for replicating a block storage service 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 paper describes Gnothi[1], a new storage block system that achieves all these properties. 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 [20, 21, 27]. 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 ($2f+1$ 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 those 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 lineariz-

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

able [17]; 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 [18]. 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 [5]. 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 that 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 Design

2.1 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, but this is not discussed or evaluated in this paper.

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. It 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 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.

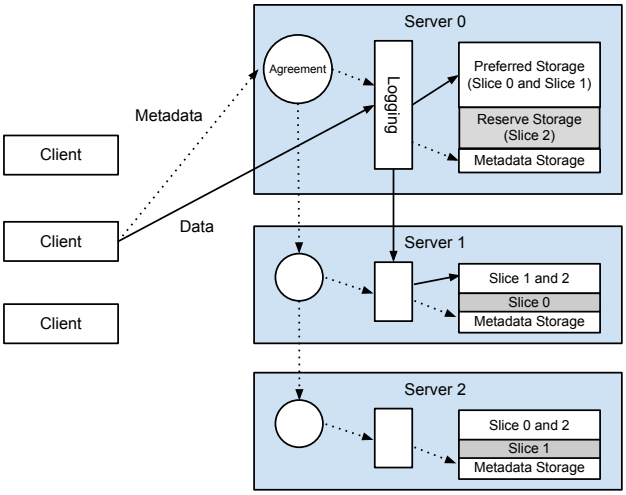

Figure 1: Data and metadata flow for a request to update a block in slice 1.

2.2 Architecture

As shown in Figure 1, Gnothi uses the Replicated State Machine (RSM) approach [27]: agreement modules on different replicas work together to guarantee that all replicas process the same client update requests in the same order. Requests 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.
- 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 3.3 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

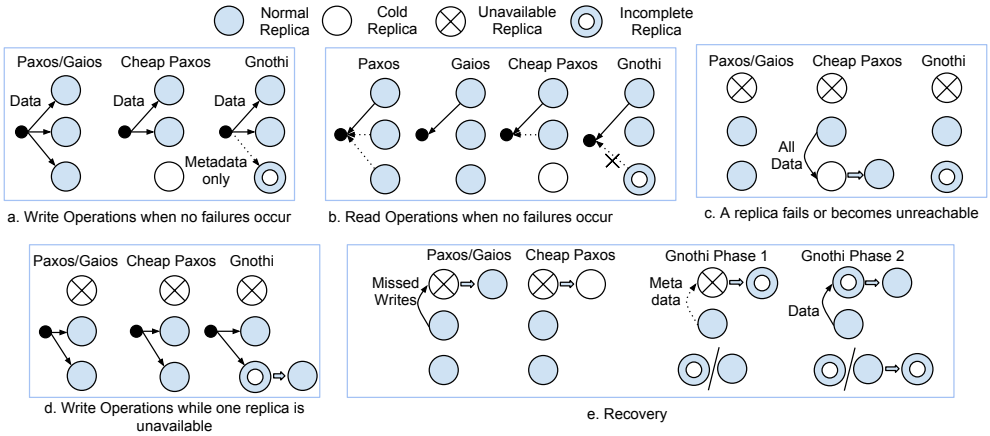

Figure 2: Gnothi protocols. We only show the logical flow of data and metadata in the figure, not actual messages. Consensus messages among servers are omitted, and we only show the flow and state for a single block. Multiple blocks are distributed among servers, so that each replica holds both *COMPLETE* and *INCOMPLETE* blocks.

on $f+1$ preferred replicas, from replica n 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.

2.3 Protocol Overview

This section presents an overview of Gnothi's protocol and compares Gnothi with the asynchronous full replication used by Paxos [20, 21], the asynchronous partial replication used by Cheap Paxos [22], and the state-of-the-art Paxos-based block replication system Gaios [5].

Figure 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 [1]. Cheap Paxos adopts this idea by activating two replicas and leaving the other one as a cold backup [22, 29]. 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.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 [9]. 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.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. In Gnothi, 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. Therefore, Gnothi also does not need any special handling when a replica becomes unavailable.

Figure 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.e shows how recovery works. Paxos and Gaios both need to fetch all missing data before process-

ing 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 others. Since metadata is updated on all replicas, this phase of recovery proceeds as in a traditional RSM. After this phase is complete, 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 not necessary to allocate a large reserve storage.

In the second phase, the recovering replica rereplicates 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.

2.4 Summary

Table 1 summarizes 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, and in Cheap Paxos, the whole system must halt until the transfer completes.

3 Detailed Design

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

3.1 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 3.5, 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 have. 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. Replicas run an agreement protocol to guarantee that WriteData messages are processed in the same order by all correct replicas.

Note that a PrepareData may never be consumed by a replica. For example, a client could fail after sending the PrepareData but before sending the WriteData. To garbage-collect unused PrepareDatas, a client includes a client sequence number with each PrepareData and WriteData it sends, and a replica discards an unused PrepareData if it receives a WriteData 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 PrepareDatas in the previous epoch. Notice that if the failed client does not recover, the replica cannot discard unused 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, 1 day) to detect the disconnected client and clear its buffer.

Gnothi keeps metadata for each block: an 8-byte version number assigned by agreement to identify the

Protocol	Write	Read	Space	Failure	Recovery (Disk survived)	Recovery (Disk replaced)
Paxos	2f+1	2f+1	2f+1	0	O(NB)	O(S)
Gaios	2f+1	1	2f+1	0	O(NB)	O(S)
Cheap Paxos	f+1	f+1	f+1+f (Cold)	O(S) (Blocking)	0	0
Gnothi	f+1	1	f+1+Δf $0 < \Delta \leq 1$	0	O(Nb)+O(NB)	O(Nb)+O($\frac{f+1}{2f+1}$S)

Table 1: Cost of Gnothi and previous work; 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.

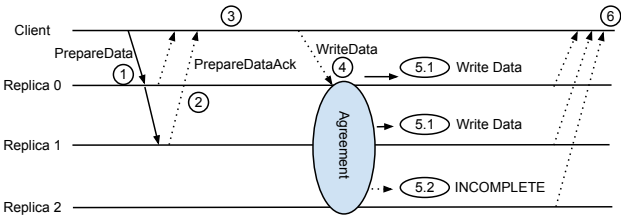

Figure 3: Write Protocol

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.

3.2 Write Protocol

The write protocol is illustrated in Figure 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 targets the block's preferred quorum, and when 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 [16, 26]: the client sends the data to one replica which forwards it to the next, and so on, in turn.

(2) A replica receiving the PrepareData puts it into a PrepareData buffer, logs it to disk, and sends an 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 [5, 12] could be used.

(5) When receiving a WriteData message, a replica 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 WriteData and PrepareData, 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*.

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

(5.3) The replica has both WriteData and PrepareData, 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 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 [9]; 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 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 [20, 21] 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.

3.3 Read Protocol

For reads, we use the Gaios read protocol [5], 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 proposed. This number is used later 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 [5], but a client can prevent it by blocking the later write when there is an outstanding read operation to the same block.

3.4 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.

3.4.1 Phase 1: Metadata recovery

Gnothi replicates metadata on each node, so metadata recovery proceeds as it would in traditional RSMs: 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 number. 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.

3.4.2 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 the number of blocks 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 re-

covers 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: if a *RESERVED* block is *COMPLETE* on its preferred replicas, then the replica can safely delete the block from its reserve storage.

3.5 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 amplifies storage costs by a factor of $2f+1$, 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. This means that when $f=1$ a replica must allocate one third of its storage space for reserve storage, and more when f is larger. This is the same space overhead as in 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 availiability for many workloads.

Specifically, we analyze two sets of traces from Microsoft: one is collected by Microsoft Research Cambridge [24] and it consists of 23 1-week disk traces under different workloads; the other is collected on Microsoft's production servers [19] and consists of 44 disk traces, whose lengths vary from 6 hours to 1 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 2 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.

3.6 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 [10, 23]. 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.

4 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 3.4.

We apply several additional modifications to improve performance: first, Gnothi modifies ZooKeeper's agreement module to incorporate batching [9, 12], 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 collections 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.

5 Evaluation

5.1 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 5 machines as servers and 5 machines as clients. We run our performance evaluation experiments for two configurations: $f = 1$ (3 servers) and $f = 2$ (5 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, 3 clients with 200 requests each are enough, but we need to run the experiments for 3 hours to measure the stable throughput; and for the 1MB sequential write workloads, it takes just 5 clients with 60 outstanding requests each to saturate the system.

5.2 I/O Throughput

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

Figure 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 [5], 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

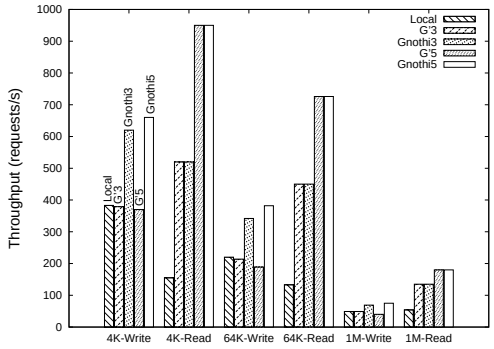

Figure 4: Random I/O with 3 (f=1) and 5 (f=2) servers.

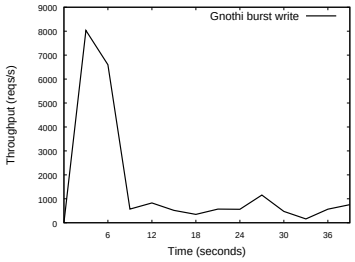

Figure 5: Burst writes. In the default configuration, Linux starts to flush dirty data to disk if 10% of total system memory pages are dirty.

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 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 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 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 about 15K updates per second. For 1MB requests, our profiler shows that the bottleneck is

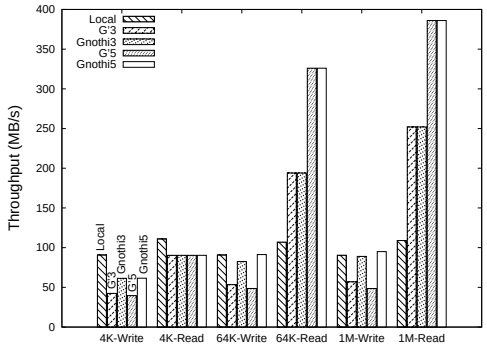

Figure 6: Sequential I/O with 3 (f=1) and 5 (f=2) servers.

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 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.

5.3 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 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 7 and 8 look at two recovery scenarios. Figure 7 shows the case when a node temporarily fails and then recovers by fetching just the updated blocks it missed. Figure 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 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 as a result of chain replication. 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.

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 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 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.

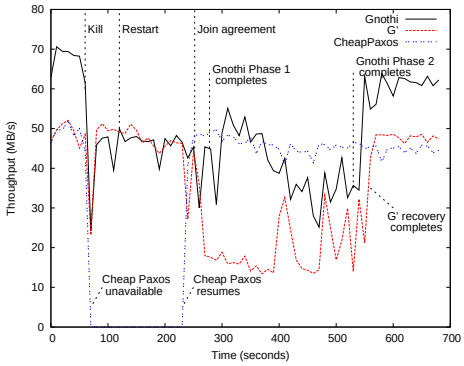

Figure 7: Failure recovery (catch up).

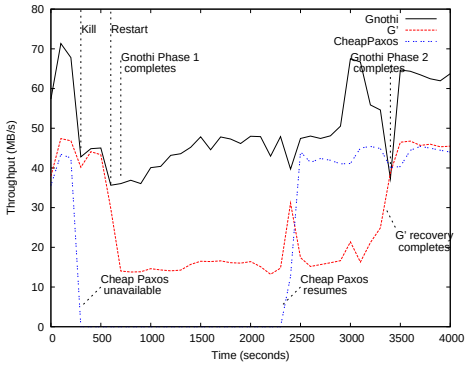

Figure 8: Failure recovery (re-replicate).

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 the re-replication is complete, but since it uses all its bandwidth to perform recovery, it can complete re-replication at time 2400.

Comparing Figure 7 and Figure 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 a higher client throughput but longer recovery time for full replication.

Both Gnothi and G' can tune a parameter to divide resources between servicing new requests and fetching state for recovery. The parameter is the time interval (ms) for a replica to issue a 16 MB state fetch request, where a smaller number means more aggressive recovery. In Figures 7 and 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 9 and 10, we show the effect of different configurations.

In Figure 9, we can see that Gnothi can always catch up, so the administrator can tune this parameter to bal-

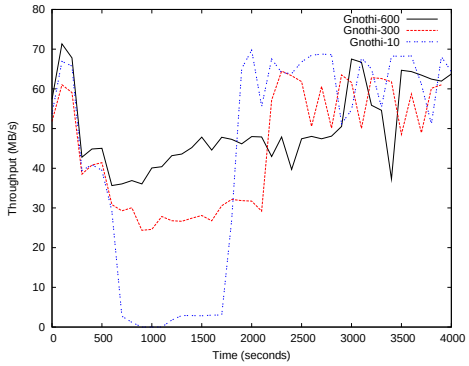

Figure 9: Gnothi with different recovery values.

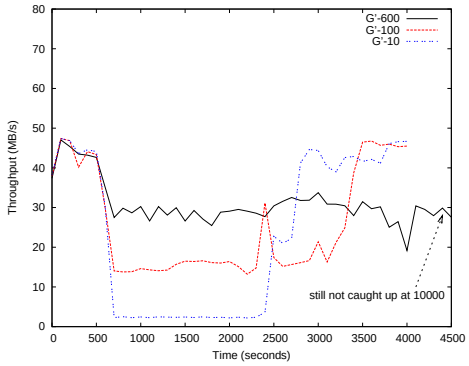

Figure 10: G' with different recovery values.

ance 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 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 throughput is similar, Gnothi can recover faster.

6 Related Work

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

In synchronous replication [6, 7, 13], 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 [5]: 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 [20, 21] 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 [5]. ZooKeeper [18] includes a fast read protocol that executes on a single replica, but it does not provide Paxos's linearizability guarantee.

On-demand instantiation (ODI) [22, 29] 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. Distler et al. [14] 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.

Separating data and metadata Paris et al. [25] reduce the storage overhead of voting using volatile witnesses. Yin et al. [30] separate agreement from execution to reduce the number of execution nodes required for Byzantine replication, and Clement et al. [12] refine these techniques, but these separation techniques exploit a type of redundancy fundamentally different than that exploited by Gnothi. In particular, they exploit the redundancy

between tolerating Byzantine and omission faults; if u ("up") is the number of failures tolerated while ensuring liveness and r ("right") is the number tolerated while ensuring safety, then execution must be replicated to at least $u+max\{u,r\}+1$ nodes [11]. Gnothi "breaks" this bound by waiting for state updates to be successfully stored on $f+1$ of $2f+1$ nodes ($u+1$ of $2u+1$ in UpRight) and by using metadata to identify which nodes completed the last writes to which objects.

Several scalable cluster file systems [2, 3, 15, 28] are architected to separate data and metadata to allow one or more metadata managers to coordinate access to large numbers of storage servers by tracing where each object is stored. Gnothi, instead, focuses on scaling Paxos-based replication and providing strong consistency (linearizability) for arbitrary read/write workloads, and therefore maintains different types of metadata (block versions and requestID rather than mappings of objects/locations).

7 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.

Acknowledgement

We thank Manos Kapritsos for his unique blend of Σοφία and Φρόνησις. We also thank our shepherd, Jon Howell, and the anonymous reviewers for their insightful comments. This work was supported by NSF grant NSF-CiC-FRCC-1048269.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-Scalable Byzantine Fault-Tolerant Services. In *SOSP*, 2005.

[2] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *OSDI*, 2002.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. *ACM Trans. Comput. Syst.*, 14(1), 1996.

[4] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, 2011.

[5] 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.

[6] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault-tolerance. *ACM Trans. Comput. Syst.*, 14(1), 1996.

[7] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Primary-Backup Protocols: Lower Bounds and Optimal Implementations. In *CDCCA*, 1992.

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

[9] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.*, 20(4), 2002.

[10] 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.

[11] A. Clement. *UpRight Fault Tolerance*. PhD thesis, 2010.

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

[13] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*, 2008.

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

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, 2003.

[16] R. Guerraoui, R. Levy, B. Pochon, and V. Quema. Throughput Optimal Total Order Broadcast for Cluster Environments. *ACM Trans. Comput. Syst.*, 28(2), 2010.

[17] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

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

[19] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *IISWC*, 2008.

[20] L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

[21] L. Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4), 2001.

[22] L. Lamport and M. Masa. Cheap Paxos. In *DSN*, 2004.

[23] M. Mammarella, S. Hovsepian, and E. Kohler. Modular Data Storage with Anvil. In *SOSP*, 2009.

[24] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *FAST*, 2008.

[25] J.-F. Paris and D. Long. Voting with Regenerable Volatile Witnesses. In *ICDE*, 1991.

[26] R. Renesse and F. Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, 2004.

[27] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4), 1990.

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

[29] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the Art of Practical BFT Execution. In *Eurosys*, 2011.

[30] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *SOSP*, 2003.