Paxos Replicated State Machines as the Basis of a High-Performance Data Store

William J. Bolosky*, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters and Peng Li
Microsoft and *Microsoft Research
{bolosky, dexterb, rhaagens, norbertk, pengli}@microsoft.com

Abstract

Conventional wisdom holds that Paxos is too expensive to use for high-volume, high-throughput, data-intensive applications. Consequently, fault-tolerant storage systems typically rely on special hardware, semantics weaker than sequential consistency, a limited update interface (such as append-only), primary-backup replication schemes that serialize all reads through the primary, clock synchronization for correctness, or some combination thereof. We demonstrate that a Paxos-based replicated state machine implementing a storage service can achieve performance close to the limits of the underlying hardware while tolerating arbitrary machine restarts, some permanent machine or disk failures and a limited set of Byzantine faults. We also compare it with two versions of primary-backup. The replicated state machine can serve as the data store for a file system or storage array. We present a novel algorithm for ensuring read consistency without logging, along with a sketch of a proof of its correctness.

1. Introduction

Replicated State Machines (RSMs) [31, 35] provide desirable semantics, with operations fully serialized and durably committed by the time a result is returned. When implemented with Paxos [20], they also tolerate arbitrary computer and process restarts and permanent stopping faults of a minority of computers, with only very weak assumptions about the underlying system--essentially that it doesn't exhibit Byzantine [22] behavior. Conventional wisdom holds that the cost of obtaining these properties is too high to make Paxos RSMs useful in practice for applications that require performance. For instance, Birman [4] writes:

> Given that it offers stronger failure guarantees, why not just insist that all multicast primitives be dynamically uniform [his term for what Paxos achieves]? … From a theory perspective, it makes sense to do precisely this. Dynamic uniformity is a simple property to formalize, and applications using a dynamically uniform multicast layer are easier to prove correct.
>
> But the bad news is that dynamic uniformity is *very costly* [emphasis his].
>
> On the other hand, there are major systems (notably Paxos…) in which … dynamic uniformity is the default. … [T]he cost is so high that the resulting applications may be unacceptably sluggish.

We argue that at least in the case of systems that are replicated over a local area network and have operations that often require using hard disks, this simply is not true. The extra message costs of Paxos over other replication techniques are overwhelmed by the roughly two orders of magnitude larger disk latency that occurs regardless of the replication model. Furthermore, while the operation serialization and commit-before-reply properties of Paxos RSMs seem to be at odds with getting good performance from disks, we show that a careful implementation can operate disks efficiently while preserving Paxos' sequential consistency. Our measurements show that a Paxos RSM that implements a virtual disk service has performance close to the limits of the underlying hardware, and better than primary-backup for a mixed read-write load.

The current state of the art involves weakened semantics, stronger assumptions about the system, restricted functionality, special hardware support or performance compromises. For example, the Google File System [13] uses append-mostly files, weakens data consistency and sacrifices efficiency on overwrites, but achieves very good performance and scale for appends and reads. Google's Paxos-based implementation [8] of the Chubby lock service [5] relies on clock synchronization to avoid stale reads and restricts its state to fit in memory; its published performance is about a fifth of ours[1]. Storage-area network (SAN) based disk systems often use special hardware

[1] Though differences in hardware limit the value of this comparison.

such as replicated battery-backed RAM to achieve fault tolerance, and are usually much more costly than ordinary computers, disks and networks. There are a number of flavors of primary-backup replication [4], but typically these systems run at the slower rate of the primary or the median backup, and may rely on (often loose) clock synchronization for correctness. Furthermore, they typically read only from the primary, which at worst wastes the read bandwidth of the backup disks and at best is unable to choose where to send reads at runtime, which can result in unnecessary interference of writes with reads. Many Byzantine-fault tolerant (BFT) [1, 9, 18] systems do not commit operations to stable storage before returning results, and so cannot tolerate system-wide power failures without losing updates. In contrast, our Paxos-based RSM runs on standard servers with directly attached disks and an ordinary Ethernet switch, makes no assumptions about clock synchronization to ensure correctness, delivers random read performance that grows nearly linearly in the number of replicas and random write performance that is limited by the performance of the disks and the size of the write reorder buffer, but is not affected by the distributed parts of the system. It performs 12%-69% better than primary-backup replication on an online transaction processing load.

The idea of an RSM is that if a computation is deterministic, then it can be made fault-tolerant by running copies of it on multiple computers and feeding the same inputs in the same order to each of the replicas. Paxos is responsible for assuring the sequence of operations. We modified the SMART [25] library (which uses Paxos) to provide a framework for implementing RSMs. SMART stored its data in SQL Server [10]; we replaced its store and log and made extensive internal changes to improve its performance, such as combining the Paxos log with the store's log. We also invented a new protocol to order reads without requiring logging or relying on time for correctness. To differentiate the original version of SMART from our improved version, we refer to the new code as SMARTER[2]. We describe the changes to SMART and provide a sketch of a correctness proof for our read protocol.

Disk-based storage systems have high operation latency (often >10ms without queuing delay) and perform much better when they're able to reorder requests so as to minimize the distance that the disk head has to travel [39]. On the face of it, this is at odds with the determinism requirements of an RSM: If two operations depend on one another, then their order of execution will determine their result. Reordering across such a dependency could in turn cause the replicas' states to diverge. We address this problem by using IO parallelism both before and after the RSM runs, but by presenting the RSM with fully serial inputs. This is loosely analogous to how out-of-order processors [37] present a sequential assembly language model while operating internally in parallel.

This paper presents Gaios[3], a reliable data store constructed as an RSM using SMARTER. Gaios can be used as a reliable disk or as a stream store (something like the i-node layer of a file system) that provides operations like create, delete, read, (over-)write, append, extend and truncate. We wrote a Windows disk driver that uses the Gaios RSM as its store, creating a small number of large streams that store the data of a virtual disk. While it is beyond the scope of this paper, one could achieve scalability in both performance and storage capacity by running multiple instances of Gaios across multiple disks and nodes.

We use both microbenchmarks and an industry standard online transaction processing (OLTP) benchmark to evaluate Gaios. We compare Gaios both to a local, directly attached disk and to two variants of primary-backup replication. We find that Gaios exposes most of the performance of the underlying hardware, and that on the OLTP load it outperforms even the best case version of primary-backup replication because SMARTER is able to direct reads away from nodes that are writing, resulting in less interference between the two.

Section 2 describes the Paxos protocol to a level of detail sufficient to understand its effects on performance. It also describes how to use Paxos to implement replicated state machines. Section 3 presents the Gaios architecture in detail, including our read algorithm and its proof sketch. Section 4 contains experimental results. Section 5 considers related work and the final section is a summary and conclusion.

2. Paxos Replicated State Machines

A state machine is a deterministic computation that takes an input and a state and produces an output and a new state. Paxos is a protocol that results in an agreement on an order of inputs among a group of replicas, even when the computers in the group crash

[2] SMART, Enhanced Revision.

[3] Gaios is the capital and main port on the Greek island of Paxos.

and restart or when a minority of computers permanently fail. By using Paxos to serialize the inputs of a state machine, the state machine can be replicated by running a copy on each of a set of computers and feeding each copy the inputs in the order determined by Paxos.

This section describes the Paxos protocol in sufficient detail to understand its performance implications. It does not attempt to be a full description, and in particular gives short shrift to the view change algorithm, which is by far the most interesting part of Paxos. Because view change happens only rarely and is inexpensive when it does, it does not have a large effect on overall system performance. Other papers [20, 21, 23] provide more in-depth descriptions of Paxos.

2.1 The Paxos Protocol

As SMART uses it, Paxos binds requests that come from clients to *slots*. Slots are sequentially numbered, starting with 1. A state machine will execute the request in slot 1, followed by that in slot 2, *etc*. When thinking about how SMART works, it is helpful to think about two separate, interacting pieces: the Agreement Engine and the Execution Engine. The Agreement Engine uses Paxos to agree on an operation sequence, but does not depend on the state machine's state. The Execution Engine consumes the agreed-upon sequence of operations, updates the state and produces replies. The Execution Engine does not depend on a quorum algorithm because its input is already linearized by the Agreement Engine.

The protocol attempts to have a single computer designated as *leader* at any one time, although it never errs regardless of how many computers simultaneously believe they are leader. We will ignore the possibility that there is not exactly one leader at any time (except in the read-only protocol proof sketch in Section 3.3.2) and refer to **the** leader, understanding that this is a simplification. Changing leaders (usually in response to a slow or failed machine) is called a *view change*. View changes are relatively lightweight; consequently, we set the view change timeout in SMART to be about 750ms and accept unnecessary view changes so that when the leader fails, the system doesn't have to be unresponsive for very long. By contrast, primary-backup replication algorithms often have to wait for a lease to expire before they can complete a view change. In order to assure correctness, the lease timeout must be greater than the maximum clock skew between the nodes.

Figure 1 shows the usual message sequence for a Paxos read/write operation, leaving out the computation and disk IO delays. When a client wants to submit a read/write request, it sends the request to the leader (getting redirected if it's wrong about the current leader). The leader receives the request, selects the lowest unused slot number and sends a *proposal* to the computers in the Paxos group, tentatively binding the request to the slot. The computers that receive the proposal write it to stable storage and then acknowledge the proposal back to the leader. When more than half of the computers in the group have written the proposal (regardless of whether the leader is among the set), it is permanently bound to the slot. The leader then informs the group members that the proposal has been decided with a commit message. The Execution Engines on the replicas process committed requests in slot number order as they become available, updating their state and generating a reply for the client. It is only necessary for one of them to send a reply, but it is permissible for several or all of them to reply. The dotted lines on the reply messages in Figure 1 indicate that only one of them is necessary.

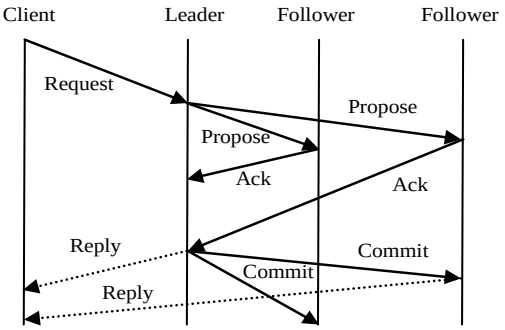

Figure 1: Read/Write Message Sequence

When the write to stable storage is done using a disk and the network is local, the disk write is the most expensive step by a large margin. Disk operations take milliseconds or even tens of milliseconds, while network messages take tens to several hundred microseconds. This observation led us to create an algorithm for read-only requests that avoids the logging step but uses the same number of network messages. It is described in section 3.3.2

2.2 Implementing a Replicated State Machine with Paxos

There are a number of complications in building an efficient replicated state machine, among them avoiding writing the state to disk on every operation. SMART and Google's later Paxos implementation [8] solve this problem by using periodic atomic checkpoints of the state. SMART (unlike Google) writes out only the changed part of the state. If a

node crashes other than immediately after a checkpoint, it will roll back its state and re-execute operations, which is harmless because the operations are deterministic. Both implementations also provide for catching up a replica by copying state from another, but that has no performance implication in normal operation and so is beyond the scope of this paper.

3. Architecture

SMARTER is at the heart of the Gaios system as shown in Figure 2. It is responsible for the Paxos protocol and overall control of the work flow in the system. One way to think of what SMARTER does is that it implements an asynchronous Remote Procedure Call (RPC) where the server (the state machine) runs on a fault-tolerant, replicated system.

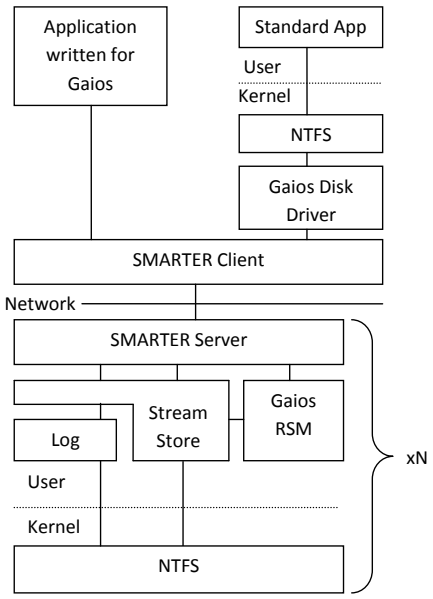

Figure 2: Gaios Architecture

Gaios's state machine implements a stream store. Streams are named by 128-bit Globally Unique IDs (GUIDs) and contain of a sparse array of bytes. The interface includes create, delete, read, write, and truncate. Reads and writes may be for a portion of a stream and include checksums of the stream data.

SMARTER uses a custom log to record Paxos proposals and the Local Stream Store (LSS) to hold state machine state and SMARTER's internal state. The system has two clients, one a user-mode library that exposes the functions of the Gaios RSM and the second a kernel-mode disk driver that presents a logical disk to Windows, and backs the disk with streams stored in the Gaios RSM.

3.1 SMARTER

Among the changes we made to SMART[4] were to present a pluggable interface for storage and log providers, rather than having SQL Server hardwired for both functions; to have a zero-copy data path; to allow IO prefetching at proposal time; to batch client operations; to have a parallel network transport and deal with the frequent message reorderings that that produces; to detect and handle some hardware errors and non-determinism; and to have a more efficient protocol for read-only requests. SMARTER performs the basic Paxos functions: client, leadership, interacting with the logging subsystem and RSM, feeding committed operations to the RSM, and managing the RSM state and sending replies to the client. It is also responsible for other functions such as view change, state transfer, log trimming, *etc*.

The SMARTER client pipelines and batches requests. Pipelining means that it can allow multiple requests to be outstanding simultaneously. In the implementation measured in this paper, the maximum pipeline depth is set to 6, although we don't believe that our results are particularly sensitive to the value. Batching means that when there are client requests waiting for a free pipeline slot, SMARTER may combine several of them into a single composite request.

Unlike in primary-backup replication systems, SMART does not require that the leader be among the majority that has logged the proposal; any majority will do. This allows the system to run at the speed of the median member (for odd sized configurations). Furthermore, there is no requirement that the majority set for different operations be the same. Nevertheless all Execution Engines will see the same binding of operations to slots and all replicas will have identical state at a given slot number.

The leader's network bandwidth could become a bottleneck when request messages are large. In this case SMARTER forwards the propose messages in a chain rather than sending them directly as shown in Figure 1. Because the sequential access bandwidth of a disk is comparable to the bandwidth of a gigabit Ethernet link, this optimization is often important.

[4] When we refer to "SMART" in the text, we mean either the original system, or to a part of SMARTER that is identical to it.

3.2 The Local Stream Store

Gaios uses a custom store called the Local Stream Store for its data (but not for its log). The LSS in turn uses a single, large file in NTFS against which it runs non-cached IO.

The LSS writes in a batch mode. It takes requests, executes them in memory, and then upon request atomically checkpoints its entire state. The LSS is designed so that it can overlap (in-memory) operation execution with most of the process of writing the checkpoint to disk, so there is only a brief pause in execution when a checkpoint is initiated.

The LSS maintains checksums for all stream data. The checksum algorithm is selectable; we used CRC32 [17] for all experiments in this paper, resulting in 4 bytes of checksum for 4K of data, or 0.1% overhead. The checksums are stored separately from the data so that all accesses to data and its associated checksum happen in separate disk IOs. This is important in the case that the disk misdirects a read or write, or leaves a write unimplemented [3]. No single misdirected or unimplemented IO will undetectably corrupt the LSS. Checksums are stored near each other and are read in batches, so few seeks are needed to read and write the checksums.

The LSS provides deterministic free space. Regardless of the order in which IOs complete and when and how often the store is checkpointed, as long as the set of requests is the same the system will report the same amount of free space. This is important for RSM determinism, and would be a real obstacle with a store like NTFS [28] that is subject to space use by external components and in any case is not deterministic in free space.

3.2.1 Minimizing Data Copies

Because SMART used SQL Server as its store, it wrote each operation to the disk four times. When logging, it wrote a proposed operation into a table and then committed the transaction. This resulted in two writes to the disk: one into SQL's transaction log and a second one to the table. The state machine state was also stored in a set of SQL tables, so any changes to the state because of the operation were likewise written to the disk twice.

For a service that had a low volume of operations this wasn't a big concern. However, for a storage service that needs to handle data rates comparable to a disk's 100 MB/s it can be a performance limitation. Eliminating one of the four copies was easy: We implemented the proposal store as a log rather than a table. Once the extra write in the proposal phase was gone, we were left with the proposal log, the transaction log for the final location and the write into the final location. We combined the proposal log and the transaction log into a single copy of the data, but it required careful thinking to get it right. Just because an operation is proposed does not mean that it will be executed; there could be a view change and the proposal may never get quorum. Furthermore, RSMs are not required to write any data that comes in an operation—they can process it in any way they want, for example maintaining counters or storing indices, so it's not possible to get rid of the LSS's transaction log entirely.

We modified the transaction log for the LSS to allow it to contain pointers into the proposal log. When the LSS executes a write of data that was already in the proposal log, it uses a special kind of transaction log record that references the proposal log and modifies the proposal log truncation logic accordingly. The necessity for the store to see the proposal log writes is why it's shown as interposing between SMARTER and the log in Figure 2. In practice in Gaios data is written twice, to the proposal log and to the LSS's store.

It would be possible to build a system that has a single-write data path. Doing this, however, runs into a problem: Systems that do atomic updates need to have a copy of either the old or new data at all times so that an interrupted update can roll forward or backward [14]. This means that, in practice, single-write systems need to use a write-to-new store rather than an overwriting store. Because we wanted Gaios efficiently to support database loads, and because databases often optimize the on-disk layout assuming it is in-order, we chose not to build a single-write system. This choice has nothing to do with the replication algorithm (or, in fact, SMARTER). If we replaced the LSS with a log-structured or another write-to-new store we could have a single-write path.

3.3 Disk-Efficient Request Processing

State machines are defined in terms of handling a single operation at a time. Disks work best when they are presented with a number of simultaneous requests and can reorder them to minimize disk arm movements, using something like the elevator (SCAN) algorithm [12] to reduce overall time. Reconciling these requirements is the essence of getting performance from a state-machine based data store that is backed by disks.

Gaios solves this problem differently for read-only and read-write requests. Read-write requests do their writes exclusively into in-memory cache, which is cleaned in large chunks at checkpoint time in a disk-efficient order. Read-only requests (ordinarily) run on only one replica. As they arrive, they are reordered and sent to the disk in a disk efficient manner, and are executed once the disk read has completed in whatever order the reads complete.

3.3.1 Read-Write Processing

SMART's handling of read-write requests is in some ways analogous to how databases implement transactions [14]. The programming model for a state machine is ACID (atomic, consistent, isolated and durable), while the system handles the work necessary to operate the disk efficiently. In both, atomicity is achieved by logging requests, and durability by waiting for the log writes to complete before replying to the user. In both, the system retires writes to the non-log portion of the disk efficiently, and trims the log after these updates complete.

Unlike databases, however, SMART achieves isolation and consistency by executing only one request at a time in the state machine. This has two benefits: It ensures determinism across multiple replicas; and, it removes the need to take locks during execution. The price is that if two read-write operations are independent of one another, they still have to execute in the predetermined order, even if the earlier one has to block waiting for IO and the later one does not.

SMARTER exports an interface to the state machine that allows it to inspect an operation prior to execution, and to initiate any cache prefetches that might help its eventual execution. SMARTER calls this interface when it first receives a propose message. This allows the local store to overlap its prefetch with logging, waiting for quorum and any other operations serialized before the proposed operation. It is possible that a proposed operation may never reach quorum and so may never be executed. Since prefetches do not affect the system state (just what is in the cache), incorrect prefetches are harmless.

During operation execution, any reads in read/write operations are likely to hit in cache because they've been prefetched. Writes are always applied in memory. Ordinarily writes will not block, but if the system has too much dirty memory SMARTER will throttle writes until the dirty memory size is sufficiently small. The local stream store releases dirty memory as it is written out to the disk rather than waiting until the end of a flush, so write throttling does not result in a large amount of jitter.

3.3.2 Read-Only Processing

SMARTER uses five techniques to improve read-only performance: It executes a particular read-only operation on only one replica; it uses a novel agreement protocol that does not require logging; it reorders the reads into a disk-efficient schedule, subject to ordering constraints to maintain consistency; it spreads the reads among the replicas to leverage all of the disk arms; and, it tries to direct reads away from replicas whose LSS is writing a checkpoint, so that reads aren't stuck behind a queue of writes.

Since a client needs only a single reply to an operation and read-only operations do not update state there is no reason to execute them on all replicas. Instead, the leader spreads the read-only requests across the (live), non-checkpointing replicas using a round-robin algorithm. By spreading the requests across the replicas, it shares the load on the network adapters and more importantly on the disk arms. For random read loads where the limiting factor is the rate at which the disk arms are able to move there is a slightly less than linear speedup in performance as more replicas are added (see Section 4). It is sub-linear because spreading the reads over more drives reduces read density and so results in longer seeks.

When a load contains a mix of reads and writes, they will contend for the disk arm. It is usually the case that on the data disk reads are more important than writes because SMARTER acknowledges writes after they've been logged and executed, but before they've been written to the data disk by an LSS checkpoint. Because checkpoints operate over a large number of writes it is common for them to have more sequentiality than reads, and so disk scheduling will starve reads in favor of writes. SMARTER takes two steps to alleviate this problem: It tries to direct reads away from replicas that are processing checkpoints, and when it fails to do that it suspends the checkpoint writes when reads are outstanding (unless the system is starving for memory, in which case it lets the reads fend for themselves). The leader is able to direct reads away from checkpointing replicas because the replicas report whether they're in checkpoint both in their periodic status messages, and also in the MY_VIEW_IS message in the read-only protocol, described immediately hereafter.

A more interesting property of read-only operations is that to be consistent as seen by the clients, they do not need to execute in precise order with respect to the read/write operations. All that's necessary is that they execute after any read/write operation that has completed before the read-only request was issued. That is, the state against which the read is run must

reflect any operation that any client has seen as completed, but may or may not reflect any subsequent writes.

SMARTER's read-only protocol is as follows:

1. Upon receipt of a read-only request by a leader, stamp it with the greater of the highest operation number that the leader has committed in sequence and the highest operation number that the leader re-proposed when it started its view.
2. Send a WHATS_MY_VIEW message to all replicas, checking whether they have recognized a new leader.
3. Wait for at least half of all replicas (including itself) to reply that they still recognize the leader; if any do not, discard the read-only request.
4. Dispatch the read-only request to a replica, including the slot number recorded in step 1.
5. The selected replica waits for the stamped slot number to execute, and then checks to see if a new configuration has been chosen. If so, it discards the request. Otherwise, it executes it and sends the reply to the client.

In practice, SMARTER limits the traffic generated in steps 2 & 3 by only having one view check outstanding at a time, and batching all requests that arrive during a given view check to create a single subsequent view check. We'll ignore this for purposes of the proof sketch, however.

SMARTER's read-only protocol achieves the following property: The state returned by a read-only request reflects the updates made by any writes for which any client is aware of a completion at the time the read is sent, and does not depend on clock synchronization among any computers in the system. In other words, the reads are never stale, even with an asynchronous network.

We do not provide a full correctness proof for lack of space. Instead we sketch it; in particular, we ignore the possibility of a configuration change (a change in the set of nodes implementing the state machine), though we claim the protocol is correct even with configuration changes.

Proof sketch: Consider a read-only request **R** sent by a client. Let any write operation **W** be given such that **W** has been completed to some client before **R** is sent. Because **W** has completed to a client, it must have been executed by a replica. Because replicas execute all operations in order and only after they've been committed, **W** and all earlier operations must have been committed before **R** was sent. **W** was either first committed by the leader to which **R** is sent (call it **L**), or by a previous or subsequent leader (according to the total order on the Paxos view ID). If it was first committed by a previous leader, then by the Paxos view change algorithm **L** saw it as committed or re-proposed it when **L** started; if **W** was first committed by **L** then **L** was aware of it. In either case, the slot number in step 1 is greater than or equal to **W**'s slot number.

If **W** was first committed by a subsequent leader to **L**, then the subsequent leader must have existed by the time **L** received the request in step 1, because by hypothesis **W** had executed before **R** was sent. If that is the case, then by the Paxos view change algorithm a majority of computers in the group must have responded to the new view. At least one of these computers must have been in the set responding in step 3, which would cause **R** to be dropped. So, if **R** completes then **W** was not first committed by a leader subsequent to **L**. Therefore, if **R** is not discarded the slot number selected in step 1 is greater than or equal to **W**'s slot number.

In step 5, the replica executing **R** waits until the slot number from step 1 executes. Since **W** has a slot number less than or equal to that slot number, **W** executes before **R**. Because **W** was an arbitrary write that completed before **R** was started SMARTER's read-only protocol achieves the desired consistency property with respect to writes. The protocol did not refer to clocks and so does not depend on clock synchronization■

3.4 Non-Determinism

The RSM model assumes that the state machines are deterministic, which implies that the state machine code must avoid things like relying on wall clock time. However, there are sources of non-determinism other than coding errors in the RSM. Ordinary programming issues like memory allocation failures as well as hardware faults such as detected or undetected data corruptions in the disk [3], network, or memory systems [30, 36] can cause replicas to misbehave and diverge.

Divergent RSMs can lead to inconsistencies exposed to the user of the system. These problems are a subset of the general class of Byzantine faults [22], and could be handled by using a Byzantine-fault-tolerant replication system [7]. However, such systems require more nodes to tolerate a given number of faults (at least *3f+1* nodes for *f* faults, as opposed to *2f+1* for Paxos [26]), and also use more network communication. We have chosen instead to anticipate a set

of common Byzantine faults, detect them and turn them into either harmless system restarts or to stopping failures. The efficacy of this technique depends on how well we anticipate the classes of failures as well as our ability to detect and handle them. It also relies on external security measures to prevent malefactors from compromising the machines running the service (which we assume and do not discuss further).

Memory allocation failures are a source of nondeterminism. Rather than trying to force all replicas to fail allocations deterministically, SMART simply induces a process exit and restart, which leverages the fault tolerance to handle the entire range of allocation problems.

In most cases, network data corruptions are fairly straightforward to handle. SMARTER verifies the integrity of a message when it arrives, and drops it if it fails the test. Since Paxos is designed to handle lost messages this may result in a timeout and retry of the original (presumably uncorrupted) message send. In a system with fewer than f failed components, many messages are redundant and so do not even require a retransmission. As long as network corruptions are rare, message drops have little performance impact. As an optimization, SMARTER does not compute checksums over the data portion of a client request or proposal message. Instead, it calls the RSM to verify the integrity of these messages. If the RSM maintains checksums to be stored along with the data on disk (as does Gaios), then it can use these checksums and save the expense of having them computed, transported and then discarded by the lower-level SMARTER code.

Data corruptions on disk are detected either by the disk itself or by the LSS's checksum facility as described in Section 3.2. SMARTER handles a detected, uncorrectable error by retrying it and if that fails declaring a permanent failure of a replica and rebuilding it by changing the configuration of the group. See the SMART paper [25] for details of configuration change.

In-memory corruptions can result in a multitude of problems, and Gaios deals with a subset of them by converting them into process restarts. Because Gaios is a store, most of its memory holds the contents of the store, either in the form of in-process write requests or of cache. Therefore, we expect at least those memory corruptions that are due to hardware faults to be more likely to affect the store contents than program state. These corruptions will be detected as the corrupted data fails verification on the disk and/or network paths.

4. Experiments

We ran experiments to compare Gaios to three different alternatives: a locally attached disk and two versions of primary-backup replication. We ran microbenchmarks to tease out the performance differences for specific homogeneous loads and an industry standard online transaction processing benchmark to show a more realistic mixed read/write load. We found that SMARTER's ability to vector reads away from checkpointing (writing) replicas conveyed a performance advantage over primary-backup replication.

4.1 Hardware Configuration

We ran experiments on a set of computers connected by a Cisco Catalyst 3560G gigabit Ethernet switch. The switch bandwidth is large enough that it was not a factor in any of the tests.

The computers had three hardware configurations. Three computers ("old servers") had 2 dual core AMD Opteron 2216 processors running at 2.4 GHz, 8 GB of DRAM, four Western Digital WD7500AYYS 7200 RPM disk drives (as well as a boot drive not used during the tests), and a dual port NVIDIA nForce network adapter, with both ports connected to the same switch. A fourth ("client") had the same hardware configuration except that it had two quad-core AMD Opteron 2350 processors running at 2.0 GHz. The remaining two ("new servers") had 2 quad-core AMD Opteron 2382 2.6 GHz processors, 16 GB of DRAM, four Western Digital WS1002FBYS 7200 RPM 1 TB disk drives, and two dual port Intel gigabit Ethernet adapters. All of the machines ran Windows Server 2008 R2, Enterprise Edition. We ran the servers with a 128 MB memory cache and a dirty memory limit of 512 MB. We used such artificially low limits so that we could hit full-cache more quickly so that our tests didn't take as long to run, and so that read-cache hits didn't have a large effect on our microbenchmarks.

4.2 Simulating Primary-Backup

In order to compare Gaios to a primary-backup (P-B) replication system, we modified SMARTER in three ways:

1. Reads are dispatched without the quorum check in the SMARTER read protocol, on the assumption that a leasing mechanism

would accomplish the same thing without the messages.

2. Read/Write operation quorums must include the leader, so for example in a 3-node configuration if the two non-leader nodes finish their logging first the system will still wait for the leader.
3. All read/write replies come only from the leader.

Because we didn't implement a leasing mechanism, the modified SMARTER might serve stale reads after a view change. We simply ignored this possibility for performance testing.

Because P-B systems read only from the primary, they cannot take advantage of the random read performance of their backup nodes. The consequences of this may be limited by having many replication groups that spread primary duties (and thus read load) over all of the nodes. In the best case, they will uniformly spread their reads over all of the nodes as SMARTER does.

To capture the range of possible read spreading in P-B systems we implemented two versions: worst and best cases. The worst case version is called PB1 because it reads from only one node. It assumes that spreading is completely ineffective and sends all reads to the primary. The best case is called PBN and simulates perfect spreading by sending reads to all N nodes. Rather than implementing multiple groups, we simply used SMARTER's existing read distribution algorithm, but without the quorum check and without the check to avoid sending reads to nodes that are checkpointing.

The latter point is the crucial difference between the two systems. While PBN is able to use all of the disk arms for reads, it can't dynamically select which arm to use for a particular read because it must send reads to the primary, and it achieves spreading only by distributing the work of the primaries for many groups. Moving a primary is far too heavy-weight to do on each read. SMARTER, on the other hand, tries to move reads away from checkpointing replicas so that writes don't interfere with reads. It also adds some randomness into the decision about when to checkpoint to avoid having replicas checkpoint in lockstep. In the mixed read/write transaction processing load measured in section 4.4 Gaios achieves 12% better performance tan PBN because of this ability (and is 68% faster than PB1).

4.3 Microbenchmarks

We ran microbenchmarks on Gaios and P-B replication as well as directly on an instance of each of the two types of disks used in our servers, varying the number of servers from 1 to 5. We expect that most applications would want to run with a group size of 3, though a requirement for greater fault tolerance or improved read performance argues for more replicas. In all of the experiments where we varied the degree of replication, we used the three old servers first followed by the two new servers, so for instance the 4 replica data point has three old and one new server.

We used the sqlio [33] tool running on NTFS over the Gaios disk driver (or directly on the local drive, as appropriate). Gaios exported a 20 GB drive to NTFS and sqlio used a 10GB file. Gaios used two identical drives on each replica, one for log and one for the data store. Each data point is the mean of 10 measurements and was taken over a five minute period, other than the burst writes shown in Figure 4, which ran for 10 seconds. We ran all tests with the disks set to write through their cache, so all writes are durable. We ran the P-B variants only on two or more nodes because they're identical to Gaios on one node, and we ran only one P-B variant on the write tests, since PB1 and PBN differ only for reads.

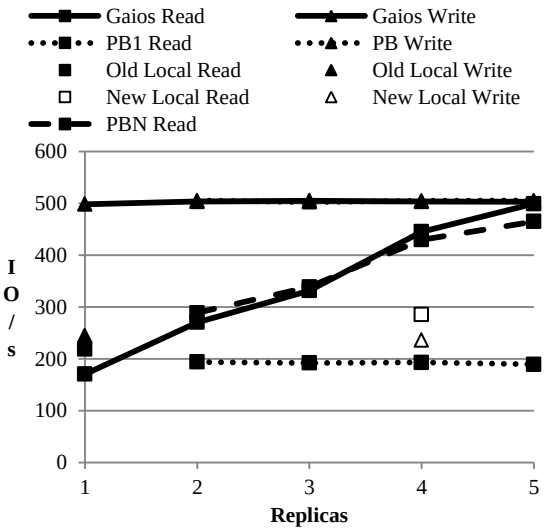

Figure 3: Random IO Performance

Figure 3 shows the performance of 8 kilobyte random reads and writes. In this and the other microbenchmark figures, we show the results for the new server disks at the 4 replica position both to provide visual separation from the old replica disks and to help point out that at 4 replicas we started adding new servers to the mix.

The writes were measured with a dirty cache. Write performance does not vary much with degree of replication or Gaios vs. P-B and is roughly 500 IO/s, a little more than twice the local disk's. This is because the server is able to reorder the writes in a disk-efficient manner over its 512MB of write buffer without the possibility of loss because the data is already logged, while the raw disks can reorder only over the simultaneously outstanding operations. The overhead of replication and checkpoints is negligible compared to disk latency, and performance is increased by SMARTER's batching.

A simple back-of-the-envelope computation shows how fast we expect the disk to be able to retire random writes, and demonstrates that SMARTER achieves that bound, meaning that (at least for random writes) the bottleneck is at the disk, not elsewhere. The disks we used have tracks about ¾ of a megabyte in size, so the 10GB sqlio file was around 14K tracks. SMARTER is using 512MB of cache, which is 64K 8KB-sized individual writes, or about 4.7 writes/track. The 7200 RPM disk takes 8.3ms for a complete rotation. 4.7 writes per each 8.3ms rotation is about 570 writes/s, which is just a little more than Gaios' performance.

The random read test used 35 simultaneous outstanding reads. Gaios' and PBN's random reads (also shown in Figure 3) scale slightly sub-linearly with the number of replicas. They improve with the number of replicas because SMARTER is able to employ the disk arms on the replicas separately, but the improvement is less than linear because as it scales each replica has fewer simultaneous reads over which to reorder. Single replica Gaios has a read rate about 14% lower than the local disk. PB1 didn't vary in the count of replicas since it only reads from one node.

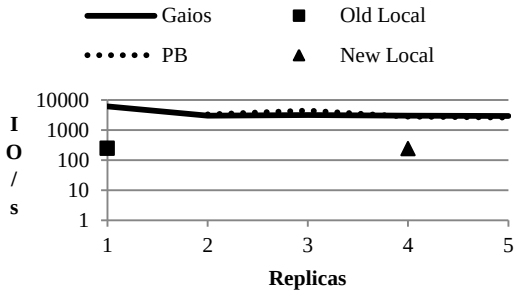

Figure 4: Burst Write Performance

Figure 4 shows the write rates for 10 second bursts of 8K random writes with 200 writes outstanding at a time. In this test, Gaios and PB logged and executed the writes and returned the replies to the client, but because the volume of data written was smaller than the 512MB dirty cache limit, it was bounded only by logging not by the seek rate of the data disk. Because SMARTER answers writes when they're written to the log, it does random write bursts at the rate of sequential writes, while the local disk does them at the rate of random writes.

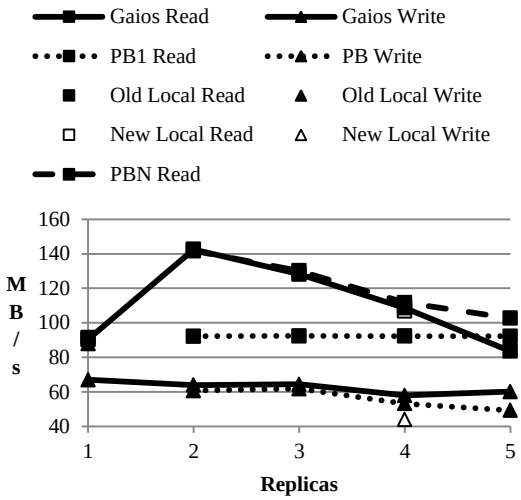

Figure 5: Sequential Bandwidth

Figure 5 shows Gaios' performance for sequential IO. This test used megabyte size requests with 40 simultaneously outstanding for writes and 10 eight megabyte requests for reads. It's difficult to see on the graph, but the (old) local disk writes at about 88 MB/s, while Gaios is at 67 MB/s. The difference is due to a difficulty in getting the data through the network transport. Writes for both Gaios and PB slow down marginally as they're distributed across more nodes (and as they need to write the slower new disks at 4 and 5 replicas). PBN and Gaios' reads are more interesting: unlike random IO, sequential IO is harder to parallelize because distributing sequential IO requests adds seeks, which reduces efficiency, sometimes more than the increase in bandwidth that's achieved by adding extra hardware. This shows up in the PBN and Gaios lines, which perform at the local disk rate on a single replica, peak at 2 replicas (but at only 1.3 times the rate of a local disk) and drop off roughly linearly after. SMARTER probably would benefit from getting hints from the RSM about how to distribute reads.

Figure 6 shows the operation latency for 8K reads and writes. Unlike the other microbenchmarks, this test only allowed a single operation to be outstanding at a time. For reads, Gaios is about 8% slower than a

local disk in the single replica case and 20% slower for 2-3 replicas. The difference in going from one to two replicas is that there is extra network traffic in the server to execute the read-only algorithm (see Section 3.3.2). Both versions of PB are about 2% faster than Gaios at 2 nodes, and 10-15% faster at 5 (where Gaios has to touch three nodes for its quorum check).

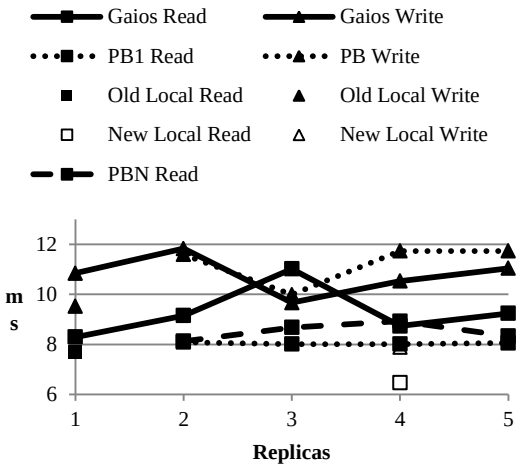

Figure 6: Single Operation Latency

Write latency is more interesting. In Gaios and P-B, the main contributor to latency is writing into the log, because the write rate is slow enough that the system doesn't throttle behind the replica checkpoint even for a 5 minute run. Writing one item to the log, waiting a little while and the writing again causes the log disk to have to take an entire 8.3ms rotation before being able to write the next log record, which accounts for the bulk of the time in Gaios. Latency goes down at three replicas because only 2 of three of them need to complete their log write for the operation to complete. As the replication grows PB gets slower than Gaios because of its requirement that the primary always be in every quorum.

The reason for storing data in an RSM is to achieve fault tolerance. To measure how Gaios performs when a fault occurs we ran a 60 second version of the 3 replica sequential read test and induced the failure of a replica half way through each of the runs. The resultant bandwidth was 127 MB/s, roughly equivalent to the 128MB/s of the non-faulty three node case. However, the maximum operation latency increased from 1500ms to 1960ms, because requests outstanding at the time of the failure had to time out and be retried. The large max latency in the non-failure case was due to the disk scheduling algorithm starving one request for a while and because of queuing delay (which is substantial with 10 8MB reads simultaneously outstanding).

4.4 Transaction Processing

In order to observe Gaios in a more realistic setting (and with a mixed read/write load), we ran an industry standard online transaction processing (OLTP) benchmark that simulates an order-entry load. We selected the parameters of the benchmark and configured the database so that it has about a 3GB log file and a 53GB table file. We housed the log and tables on different disks. In Gaios (and P-B) we ran each virtual disk as a separate instance of Gaios sharing server nodes, but using distinct data disks on the server. SMARTER shared a single log disk, so each server node used three disks: the SMARTER log, the SQL log and the SQL tables.

This benchmark does a large number of small transactions of several different types, and generates a load of about 51% reads and 49% writes to the table file by operation count, with the average read size about 9K and the average write about 10K. We configured the benchmark to offer enough load that it was IO bound. The CPU load on the client machine running SQL Server was negligible.

We used 64-bit Microsoft SQL Server 2008 Enterprise Edition for the database engine. For each data point, we started by restoring the database from a backup, which resulted in identical in-file layout. We then ran the benchmark for three hours, discarded the result from the first hour in order to avoid ramp-up effects and used the transaction rate for the second two hours. This benchmark is sensitive to two things: write latency to the SQL Server log, and read latency to the table file. The writes are offered nearly continuously as SQL Server writes out its checkpoints and are mixed with the reads.

Even though the load is half writes, the replicas spent significantly less than half of their time writing. This is because the writes were more sequential than the reads because they came from SQL's database cleaner which tries to generate sequential writes, and they were further grouped by SMARTER's checkpoint mechanism. Because of this, Gaios usually had one or more replicas that were not in checkpoint to which to send reads. Even though the load at the client was about half reads and half writes, at the server nodes it was ¾ writes because each write ran on all three nodes, while reads ran only on one. This limited the effect of the increased random read performance of Gaios and PBN.

Figure 7 shows the performance of Gaios and the two PB versions running on a three node system in transactions per second normalized to the local-machine performance. Each bar is the mean of ten runs. Gaios runs a little faster than the local node because its increased random read performance more than compensates for the added network latency and checksum IO. Because PBN is unable to direct its reads away from checkpointing nodes it is somewhat slower, while PB1 suffers even more due to its inability to extract read parallelism.

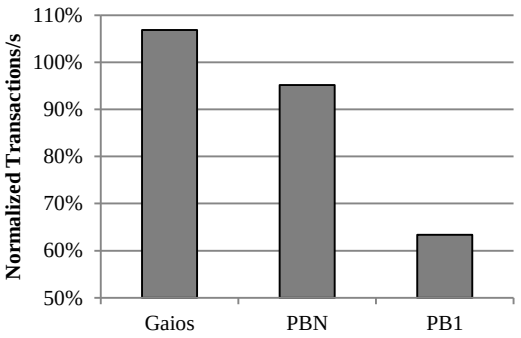

Figure 7: OLTP Performance

5. Related Work

Google [8] used a Paxos replicated state machine to re-implement the Chubby [5] lock service. They found that it provided adequate performance for their load of small updates to a state that was small enough to fit in memory (100MB). It serviced all reads from the leader (there being no need to take advantage of parallel disk access because of in-memory state), and used a time-based leasing protocol to prevent stale reads, similar to primary-backup. Their highest reported update rate was 640 small operations per second and 949 KB/s on a five node configuration, about one fifth and one sixtieth respectively of Gaios' comparable performance on 5 nodes, though because the hardware used was different it's not clear how meaningful this comparison is.

Petal [24] was a distributed disk system from DEC SRC that used two-copy primary-backup replication to implement reliability. It used a Paxos-based RSM to determine group membership, but not for data. Data writes happened in two phases, first taking a lock on the data and then writing to both copies. Only when the writes to both copies completed was the lock released and the operation completed to the user. Much like Gaios, Petal used write-ahead logging and group commit to achieve good random write performance. Castro and Liskov [7] implemented a version of NFS that stored all of its data in a BFT replicated state machine. However, their only performance evaluation was with the Andrew Benchmark [16], which has been shown [38] to be largely insensitive to underlying file system performance. BFT replication differs from Paxos in that it tolerates arbitrary, potentially malicious failures of less than a third of its replicas. It uses many more messages and a number of cryptographic operations to achieve this property.

Several BFT agreement protocols [1, 9, 18] have much lower latency than Gaios. They achieve this by not logging operations before executing them and returning results to the client. Because of this, these systems cannot tolerate simultaneous crashes of too many nodes (such as would be caused by a datacenter power failure) without permanently failing or rolling back state. As such, they do not provide sufficiently tight semantics to implement tasks that require write through such as the store for a traditional database. They also are not evaluated on state that is larger than memory. Furthermore, because they tolerate general Byzantine faults, they need at least $3f+1$ (and sometimes more) replicas to tolerate f faults (though f of these replicas can be witnesses that do not hold execution state [40]). Gaios tolerates many non-malicious (hardware or programming-error caused) Byzantine faults without the extra complexity of dealing with peers that are trying to corrupt the system.

The Federated Array of Bricks (FAB) [34] built a store out of a set of industry-standard computers and disks, much like Gaios. It used a pair of custom replication algorithms, one for mirrored data and one for erasure-coded. Unlike Paxos, it did not have a leader function or views; rather (in the mirroring case), it took a write lock over a range of bytes using a majority algorithm. Once the write lock was taken, it sent the write data to all nodes, and updated both the data and a timestamp. After a majority of the nodes completed the write, it completed the operation back to the caller. To read data, it sent the read to all replicas, with one designated to return the data. The other nodes returned only timestamps; if the returned data did not have the latest timestamp, it retried the read. This scheme achieves serializability without needing to achieve a total order of operations as happens in an RSM. However, because its read algorithm requires accessing a per-block timestamp, it employed NVRAM to avoid the need to move the disk arms to read the timestamps; SMARTER's algorithm simply asks for a copy of in-memory state from all of the replicas, and does the disk IO on only one and so does not need NVRAM.

Oceanstore [19] was designed to store the entire world's data. It modified objects by generating updates locally and then running conflict resolution in the background, in the style of Bayou [11]. Oceanstore used a Byzantine-agreement protocol to serialize and run conflict resolution, but stored the data using simple lazy replication (or replication of erasure coded data).

The Google File System [13] is designed to hold very large files that are mostly written via appends and accessed sequentially via reads. It relaxes traditional file system consistency guarantees in order to improve performance. In particular, write operations that fail because of system problems can leave files in an "inconsistent" state, meaning that the values returned by reads depend on which replica services the read. Furthermore, concurrent writes can leave file regions in an "undefined" state, where the result is not consistent with any serialization of the writes, but rather is a mixture of parts of different writes. After a period of time, the system will correct these problems. GFS uses write-to-all, so faults require the system to reconfigure before writes can proceed.

Berkeley's xFS [2] and Zebra [15] file systems placed a log structured file system [32] on top of a network RAID. They worked by doing write-to-all on the RAID stripes, and then using a manager to configure out failed storage nodes. The xFS prototype described in the paper did not "implement the consensus algorithm needed to dynamically reconfigure manager maps and stripe group maps."

Boxwood [27] offered a set of storage primitives at a higher level than the traditional array of blocks, such as B-trees. It used Paxos only to "store global system state such as the number of machines."

Everest [29] is a system that offloads work from busy disks to smooth out peak loads. When off-loading, it writes multiple copies of data to any stores it can find and keeps track of where they are in volatile memory. After a crash and restart, the client scans all of the stores to find the most up-to-date writes, and as long as one copy of each write is available, it recovers. This protocol works because there is only ever one client for a particular set of data.

TickerTAIP [6] was a parallel RAID system that distributed the function of the RAID controller in order to tolerate faults in the controller. It used two-phase commit [14] to ensure atomicity of updates to the RAID stripes.

6. Summary and Conclusion

Conventional wisdom holds that while Paxos has theoretically desirable consistency properties, it is too expensive to use for applications that require performance. We argue that compared to disk access latencies, the overhead required by Paxos on local networks is trivial and so the conventional wisdom is incorrect. While replicated state machines' in-order requirement seems to be at odds with the necessity of doing disk operation scheduling, careful engineering can preserve both.

We presented Gaios, a system that provides a virtual disk implemented as a Paxos RSM. Gaios achieves performance comparable to the limits of the hardware on which it's implemented on various microbenchmarks and the OLTP load, while providing tolerance of arbitrary machine restarts, a sufficiently small set of permanent stopping failures and some types of Byzantine failures. We compared Gaios to primary-backup replication and found that it performs comparable to or in some cases better than P-B's best case. We presented a novel read-only algorithm for SMARTER, and showed that because it allows reads to run on any node SMARTER can often avoid having reads and writes contend for a particular disk, giving significant performance improvements over even the best case of primary-backup replication for the mixed read/write workload of the OTLP benchmark.

Bibliography

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. SOSP*, 2005.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli and R. Wang. Serverless network file systems. In *Proc. SOSP*, 1995.

[3] L. Bairavasundaram, G. Goodson, B. Schroeder, A. Arpaci-Dusseau and R. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In *Proc. FAST*, 2008.

[4] K. Birman. *Reliable Distributed Systems Technologies, Web Services and Applications*. Springer, 2005

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

[6] P. Cao, S. Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP parallel RAID architecture. In *Proc. ISCA*, 1993.

[7] M. Castro and B. Liskov, Practical Byzantine fault tolerance. In *Proc. OSDI*, 1999.
[8] T. Chandra, R. Griesemer and J. Redstone. Paxos made live: an engineering perspective. In *Proc. PODC*, 2007. Invited talk.
[9] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shira. HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In *Proc. OSDI*, 2006.
[10] K. Delaney, P. Randal, K. Tripp and C. Cunningham. *Microsoft SQL Server 2008 Internals*. Microsoft Press, 2009.
[11] A. Demers, K. Peterson, M. Spreitzer, D. Terry, M. Theimer and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proc. IEEE Workshop on Mobile Computing Systems & Applications*, 1994.
[12] R. Eager and A. Lister. *Fundamentals of Operating Systems*. Springer-Verlag, 1995.
[13] S. Ghemawat, H. Gobioff and S-T. Leung. The Google file system. In *Proc. SOSP*, 2003.
[14] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
[15] J. Harman and J. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3), 1995.
[16] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), 1988.
[17] IEEE 802.3 Standard, 1983-2008.
[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proc. SOSP*, 2007.
[19] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. ASPLOS*, 2000.
[20] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), 1998.
[21] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), 2001.
[22] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*. 4(3), 1982.
[23] B. Lampson. The ABCD's of Paxos. In *Proc. PODC*, 2001.
[24] E. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proc. ASPLOS*, 1996.
[25] J. Lorch, A. Adya, W. Bolosky, R. Chaiken, J. Douceur and J. Howell. The SMART way to migrate replicated stateful services. In *Proc. Eurosys*, 2006.
[26] N. Lynch. *Distributed Algorithms*. Morgan Kauffman, 1996.
[27] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. OSDI*, 2004.
[28] R. Nagar. *Windows NT File System Internals*. O'Reilly, 1997.
[29] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through I/O off-loading. In *Proc. OSDI*, December, 2008.
[30] E. Nightingale, J. Douceur and V. Orgovan. Cycles, Cells and Platters: An empirical analysis of hardware failures on a million commodity PCs. To appear in *Proc. EuroSys*, 2011.
[31] B. Oki. Viewstamped replication for highly available distributed systems. Ph.D. thesis. Technical Report MIT/LCS/TR-423, MIT, 1988.
[32] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), 1992.
[33] M. Ruthruff. SQL Server best practices article: predeployment I/O best practices. In IEEE Computer, 27(3), 1994.
[34] Y. Saito, S. Frølund, A. Veitch, A. Merchant and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proc. ASPLOS*, 2004.
[35] F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4), 1990.
[36] B. Schroeder, E. Pinheiro, and W-D. Weber. DRAM Errors in the wild: A large-scale field study. In *Proc. SIGMETRICS/Performance*, 2009.
[37] R. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM Journal of Research and Development*, 11(1), 1967.
[38] A. Traeger, E. Zadok, N. Joukov and C. Wright. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage*, 4(2), 2008.
[39] B. Worthington, G. Ganger, and Y. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proc. SIGMETRICS*, 1994.
[40] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. SOSP*, 2003.