FAB: Building Distributed Enterprise Disk Arrays from Commodity Components

Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, Susan Spence
Hewlett-Packard Laboratories
firstname.lastname@hp.com

ABSTRACT

This paper describes the design, implementation, and evaluation of a Federated Array of Bricks (FAB), a distributed disk array that provides the reliability of traditional enterprise arrays with lower cost and better scalability. FAB is built from a collection of *bricks*, small storage appliances containing commodity disks, CPU, NVRAM, and network interface cards. FAB deploys a new majority-voting-based algorithm to replicate or erasure-code logical blocks across bricks and a reconfiguration algorithm to move data in the background when bricks are added or decommissioned. We argue that voting is practical and necessary for reliable, high-throughput storage systems such as FAB. We have implemented a FAB prototype on a 22-node Linux cluster. This prototype sustains 85MB/second of throughput for a database workload, and 270MB/second for a bulk-read workload. In addition, it can outperform traditional master-slave replication through performance decoupling and can handle brick failures and recoveries smoothly without disturbing client requests.

Categories and Subject Descriptors

D.4.5 [**Software**]: Operating systems—*Reliability*; C.5.5 [**Computer system implementation**]: Servers; H.3.4 [**Information storage and retrieval**]: Systems and software—*Distributed systems*

General Terms

Algorithms, Management, Performance, Reliability

Keywords

Storage, disk array, replication, erasure coding, voting, consensus

1. INTRODUCTION

A *Federated Array of Bricks* (FAB) is a distributed disk array that provides reliable accesses to logical volumes using only commodity hardware. It solves the two problems, scalability and cost, associated with traditional monolithic disk arrays.

Traditional disk arrays drive collections of disks using centralized controllers. They achieve reliability via highly customized, redundant and hot swappable hardware components. They do not scale well, because there is a high up-front cost for even a minimally configured array, and a single system can only grow to a limited size. These limitations force manufacturers to develop multiple products for different system scales, which multiplies the engineering efforts required. These issues, coupled with relatively low manufacturing volumes, drive up their cost—high-end arrays retail for many millions of dollars, at least 20 times more than the price of consumer-class systems with equivalent capacity.

FAB consists of a collection of *bricks*—small rack-mounted computers built from commodity disks, CPU, and NVRAM—connected by standard networks such as Ethernet. Bricks autonomously distribute data and functionality across the system to present a highly available set of logical volumes to clients through standard disk-access interfaces such as iSCSI [32]. FAB can scale incrementally, starting from just a few bricks and adding more bricks as demand grows, up to several hundred bricks. It is also cheaper than traditional arrays: due to the economies of scale inherent in high-volume production, a brick with 12 SATA disks and 1GB of NVRAM can be built for less than $2000, with a total system cost of about 20% to 80% of traditional arrays, even with three-way replication.

Commodity hardware is, of course, far less reliable than its enterprise counterparts. Using the reliability figures reported in [4, 3], we expect the mean time between failures of a typical network switch to be 4 years, and that of a typical brick to be 4 to 30 years, depending on the quality of disks and the internal disk organization (e.g., RAID-5 is more reliable than RAID-0). FAB inevitably faces frequent changes to the system, including brick failures or additions, and network partitioning.

The FAB project tries to achieve two goals in such environments. First, FAB should provide *continuous service*, masking failures transparently and ensuring stable performance over diverse workloads. Second, it should ensure *high reliability*, comparable to that of today's high-end disk arrays: 10,000+ mean years before the first data loss, tolerating the failures of disks, CPUs, or networks.

The key idea behind FAB to achieve these goals is replication and erasure coding by *voting*. Acting on behalf of a client, a read or write request coordinator communicates with a subset (quorum) of bricks that store the data. Voting allows FAB to tolerate failed bricks and network partitioning safely without blocking. It also enables *performance decoupling* [24]—tolerating overloaded bricks by simply ignoring them, as long as others are responsive. This is especially effective in systems like FAB, in which brick response times fluctuate due to the randomness inherent in disk-head mechanisms. Voting-based replication is not new, but it has seen little use in high-throughput systems, because of concerns about inefficiency, as reading data must involve multiple remote nodes [35]. In

ASPLOS'04 October 9–13, 2004, Boston, Massachusetts, USA.

this paper, we show that voting is indeed practical and often necessary for reliable, high throughput storage systems. Specifically, our contributions are:

New replication and erasure-coding algorithms: We present asynchronous voting-based algorithms that ensure strictly linearizable accesses [17, 2] to replicated or erasure-coded data. They can handle any non-Byzantine failures, including brick failures, network partitioning, and slow bricks. Existing algorithms [5, 27], in contrast, not only lack erasure-coding support, but also could break consistency when a brick that coordinates a request crashes in the middle.

A new dynamic quorum reconfiguration algorithm: can adjust quorum configurations dynamically, while allowing I/O requests from clients to proceed unimpeded. It improves reliability by allowing the system to tolerate more failures than in a system with fixed-quorum voting, and by adding a new brick after another brick is decommissioned.

Efficient implementation and evaluation of FAB: present several techniques that improve the efficiency of these algorithms and implement them in FAB.

We have implemented a FAB prototype on a 22-node Linux cluster. As we show in Section 7, this prototype sustains 85MB/second of throughput for a database workload, and 270MB/second for a bulk-read workload. In addition, it can outperform traditional master-slave replication through performance decoupling and can handle brick failures and recoveries smoothly without disturbing client requests.

2. RELATED WORK

Today's standard solution for building reliable storage systems are centralized disk arrays employing RAID [7], such as EMC Symmetrix, Hitachi Lightning, HP EVA, and IBM ESS. To ensure reliability, these systems incorporate tightly synchronized, hardware-level redundancy at each layer of the system's functionality, including processing, cache, disk controllers and RAID control. As reviewed in the previous section, this architecture limits their capacity, throughput, and availability. FAB distributes the functionality of array controllers across bricks while maintaining the consistency semantics of a single disk.

The idea of distributed, composable disk arrays was pioneered by TickerTAIP [6] and Petal [22]. Petal uses a master-slave replication protocol, which cannot tolerate network partitioning. In addition, it has a period (∼30 seconds) of unavailability during fail-over, which can cause clients to take disruptive recovery actions, such as database-log or file-system scanning. In contrast, FAB can mask failures safely and instantaneously using voting, and it supports Reed-Solomon erasure coding in addition to replication. Recently, LeftHand Networks [23] and IBM [19] have proposed FAB-like storage systems, but no details about them have been published.

Network-attached secure disks (NASD) [13] let clients access network-attached disks directly and safely. Both FAB and NASD try to build scalable distributed storage, but with different emphases: FAB focuses on availability and reliability through redundancy, whereas NASD focuses on safety through access-control mechanisms. These systems complement each other.

The ability of voting algorithms to tolerate failures or slow nodes has led to their recent adoption in storage systems. FarSite [1] is a distributed serverless file system that uses voting-based algorithms to tolerate Byzantine failures. *Self-** is also a serverless file system that uses quorum-based erasure-coding algorithms [12, 16].

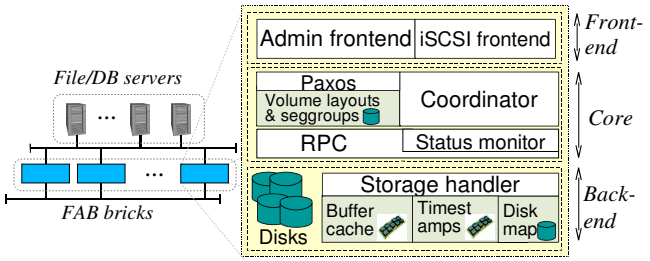

Figure 1: The structure of a FAB system. Bricks are connected to each other and to clients by commodity networks. All bricks run the same set of software modules, shown in the right-hand picture. Volume layouts, seggroups, and diskmaps are on-disk data structures, normally cached in memory. The buffer cache and timestamp table are stored in NVRAM.

FAB

OceanStore [31] is a wide-area file system that uses voting to tolerate Byzantine failures and erasure coding for long-term, space-efficient data storage. Unlike these systems, FAB is designed as a high-throughput local-area storage system. It tolerates only stopping failures, but it ensures consistent data accesses without changing the clients or exploiting file-system semantics. Ling [24] and Huang [18] use voting to build a high-throughput storage system, but they support only replication, with only single-client accesses, and require a special protocol to run on each client.

We

Consistent reconfiguration has been studied in viewstamped replication [29], which uses two-phase commits for updating data and Paxos [20, 21] for transitioning views. More recently, RAMBO [27] proposed the idea of concurrent active views and background state synchronization. This idea is used in FAB as well, but whereas RAMBO is based on single register (logical block) emulation, FAB runs more efficient voting algorithms over multiple logical blocks.

3. OVERVIEW

Figure 1 shows the structure of a FAB system. FAB is a symmetrically distributed system—each brick runs the same set of software modules and manages the same types of data structures. FAB clients, usually file or database servers, use iSCSI [32] for reading and writing logical blocks, and a proprietary protocol for administrative tasks, such as creating and deleting logical volumes. At a high level, a read or write request is processed as follows:

1. The client sends an iSCSI request of the form ⟨*volume-id, offset, length*⟩ to a *coordinator*, that is, a brick that acts as a gateway for the request. Because of FAB's symmetric structure, the client can choose any brick as the coordinator to access any logical volume. Different requests, even from the same client, can be coordinated by different bricks. In practice, the client uses either hard-wired knowledge or a protocol such as iSNS [33] (a name service for iSCSI) to pick a coordinator.
2. The coordinator finds the set of bricks that store the requested blocks. These are the *storage bricks* for the request.
3. The coordinator runs the replication or erasure-coding protocol against the storage bricks, passing the tuple ⟨*volume-id, offset, length*⟩ to them.
4. Each storage brick converts the tuple ⟨*volume-id, offset, length*⟩ to physical disk offsets and accesses the requested data.

3.1 Key data structures and software modules

The steps described above are carried out using the following key data structures:

- *Volume layout* maps a logical offset to a *seggroup* at segment granularity for each volume. A segment, set to 256MB, is the unit of data distribution.
- *Seggroup* describes the layout of a segment, including the set of bricks that store the segment. The volume layout and seggroups are used in step 2 to locate the set of storage bricks for a request. A seggroup is also the unit of reconfiguration, as we discuss further in Section 5.
- *Diskmap* maps a logical offset to the tuple ⟨disk-number, disk-offset⟩ at *page* granularity for each logical volume. A page, set to 8MB, is the unit of disk allocation. Diskmap contents are unique to each brick. Diskmaps are used in step 4.
- *Timestamp table* stores timestamp information for recently modified blocks. The contents of this table are unique to each brick. This data structure is used in steps 3 and 4 to access replicated or erasure-coded blocks in a consistent fashion. We discuss FAB's replication and erasure-coding algorithms and their use of timestamp tables in more detail in Section 4.

Figure 2 shows an example of I/O request processing. Volume layouts and seggroups are called the *global metadata*, because they are replicated on every brick and are read by the request coordinator. Following the approach pioneered by Petal [22], we use Paxos [20, 21], an atomic broadcast protocol, to maintain the consistency of the global metadata across bricks. Paxos allows bricks to receive exactly the same sequence of metadata updates, even when updates are issued concurrently and bricks fail and recover. Thus, by letting bricks initially boot from the same (empty) global metadata and use Paxos for updates, they can keep their metadata consistent. As discussed further in Section 5.2, FAB is designed to withstand stale global metadata, so long as bricks eventually receive metadata updates. As such, reading global metadata is done directly against the local copy.

These data structures are managed by software modules that are roughly divided into three groups. The *frontend* receives requests from clients (step 1). The *core* contains modules needed to locate logical blocks and maintain data consistency (steps 2 and 3). In particular, the *coordinator* module is responsible for communicating with the backend modules of remote bricks to access blocks consistently. The *status monitor* keeps track of the disk usage and load of other bricks. It is used to assign less-utilized segment groups to volumes while creating volumes (Section 3.2), and to pick a brick in the quorum that reads data from disk (Section 4.4). It currently deploys two mechanisms. First, the status information is piggybacked on every message exchanged between bricks; this gives a timely view of the status of a small set of bricks. Second, we use a variation of the gossip-based failure detector [**?**] to advertise the status to a random brick every three seconds; this gives an older, but more comprehensive, view of the system. Finally, the *backend* modules are responsible for managing and accessing NVRAM and physical disks (step 4).

3.2 Data layout and load balancing

All the segments assigned to a seggroup must use the same redundancy policy: replication of the same degree or erasure coding with the same layout. FAB's policy is to create, for each redundancy policy, an average of four seggroups that contain a specific brick. Logical volume segments are assigned to seggroups semi-randomly when the volume is created, favoring seggroups containing bricks with less utilized disks (the status monitor is consulted for this purpose). The assignment of physical disk blocks to pages (i.e., diskmap) is done randomly by each brick when the page is written for the first time.

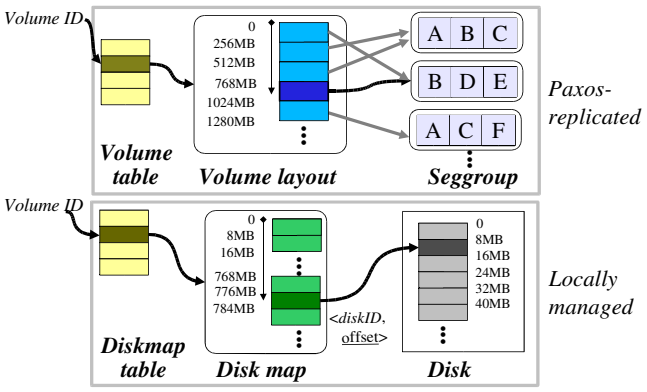

Figure 2: Example of locating a logical 1KB block at offset 768MB of a volume. The client sends a request of the form ⟨volume-id, 768MB, 1KB⟩ to a random coordinator. In the top half of the diagram, the coordinator locates the volume layout from the local copy of the global metadata and finds the seggroup for the offset 768MB. The seggroup shows that the data is stored on bricks B, D, and E. The coordinator then executes the replication or erasure-coding protocol against bricks B, D and E. In the bottom half of the diagram, each of the bricks B, D, and E consult the local diskmap to convert the offset 768MB to disk addresses.

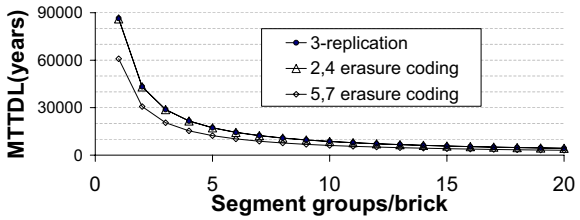

Figure 3: Mean time to data loss (MTTDL) of FAB in systems with 256 TB logical capacity.

The choice of number of seggroups per brick reveals a tension between load balancing and reliability. After a brick b fails, the "read" requests normally handled by b are now served by the other bricks in the seggroups that b belongs to. Thus, the more seggroups per brick, the more evenly the extra load is spread. Creating too many seggroups, however, reduces the system's reliability, since this increases the number of combinations of brick failures that can lead to data loss. Figure 3 shows how the reliability changes with the number of seggroups per brick. This analysis is based on a Markov model assuming bricks with twelve 256GB SATA disks. Failures are assumed to be independent. We assume a disk mean time to failure (MTTF) of 57 years, based on manufacturers' specifications and a brick (enclosure) MTTF of 30 years, based on data from [4]. The time to repair a failure depends on the failure type and is based on the time required to copy the data to spare space — we assume that spare space is always available. Based on this, we pick an average of four seggroups per brick because this meets our goal of a 10,000 year MTTDL, while still allowing the load to be spread evenly.

The choice of segment and page sizes involves several trade-offs. A larger segment size reduces the global-metadata management overhead, but at the cost of less storage allocation freedom, because bricks in a seggroup must store all its segments. The page is chosen to be smaller than the segment to reduce the storage waste for erasure-coded volumes (Section 4.2), or for logical volumes whose

size is not segment-aligned. Too small a page size, however, could also hurt performance by increasing disk-head movement. We find that the current setting of 256MB segments and 8MB pages offers a good balance for the next few years—even with bricks with 10TB raw capacity and one thousand 1TB logical volumes in the system, the size of the global metadata and diskmaps would be only 5MB and 10MB, respectively.

4. VOTING-BASED REPLICATION AND ERASURE CODING

FAB provides two redundancy mechanisms, replication and erasure-coding. Both are based on the idea of voting: each request makes progress after receiving replies from a (random) quorum of storage bricks. Our protocols require no persistent state on the request coordinator. This feature allows any brick to act as a coordinator and helps FAB become truly decentralized without changing clients.

Section 4.1 describes our basic replication protocol for a single logical block, and Section 4.2 describes how it can be extended for erasure coding. Multi-block requests are logically handled by running multiple instances of these algorithms in parallel, but in practice, we batch and run them as efficiently as single-block requests. We discuss this and other implementation-related issues in later sections.

4.1 Replication

The task of a request coordinator is straightforward in theory: when writing, it generates a new unique timestamp and writes the new block value and timestamp to a majority of storage bricks; when reading, it reads from a majority and returns the value with the newest timestamp. The challenge lies in the handling of the failure of the participants in the middle of a "write" request: the new value may end up on only a minority of bricks. A storage system must ensure *strict linearizability* [2, 17]—it must present a single global ordering of (either successful or failed) I/O requests, even when they are coordinated by different bricks. Put another way, after a "write" coordinator fails, future "read" requests to the same block must all return the old block value or all return the new value, until the block is overwritten by a newer "write" request. Prior approaches, e.g., Gifford's use of two-phase commits [14] cannot ensure a quick fail-over, and Ling et al.'s use of end-to-end consistency checking [24] conflicts with our goal of leaving the client interface (iSCSI) unchanged.

FAB takes an alternative approach, performing recovery lazily when a client tries to read the block after an incomplete write. Figure 4 shows the pseudocode of FAB's algorithm. Each replicated block keeps two persistent timestamps: *valTs* is the timestamp of the block currently stored, and *ordTs* is the timestamp of the newest ongoing "write" request. An incomplete "write" request is indicated by $ordTs > valTs$ on some brick. A "write" runs in two phases. First, in the Order phase, the replicas update their *ordTs* to indicate a new ongoing update and ensure that no request with an older timestamp is accepted. In the second, Write, phase, the replicas update the actual disk block and *valTs*. A "read" request usually runs in one phase, but takes two additional phases when it detects an incomplete past "write"—the coordinator first discovers the value with the newest timestamp from a majority, and then writes that value back to a majority with a timestamp greater than that of any previous writes. In this protocol, a "write" request still tries to write to *all* the bricks in the seggroup; the coordinator just does not wait for all the replies. Thus, a read-recovery phase usually happens only when there is an actual failure. Figure 5 shows an example of I/Os using this algorithm.

```
// I/O coordinator code.
proc write(val)
  ts ← NewTimestamp()
  send [Order, {}, ts] to bricks in the seggroup
  if a majority reply "yes"
    send [Write, val, ts] to bricks in the seggroup
    if a majority reply "yes" return OK
  return ABORTED
proc read()
  send [Read] to bricks in the seggroup
  if a majority reply "yes" and all timestamps are equal
    return the val in a reply.
  ts ← NewTimestamp()  // Slow "recover" path starts
  send [Order, "all", ts] to bricks in the seggroup
  if a majority reply "yes"
    val ← the value with highest valTs from replies
    send [Write, val, ts] to bricks in the seggroup
    if a majority reply "yes" return val
  return ABORTED

// Storage handler code. Variable val stores the block contents.
when Receive [Read]
  status ← (valTs ≥ ordTs)
  reply [status, valTs, val]
when Receive [Order, targets, ts]
  status ← (ts > max(valTs, ordTs))
  if status ordTs ← ts
  if targets = "all" or this block ∈ targets reply [valTs, val, status]
  else reply [valTs, status]
when Receive [Write, newVal, ts]
  status ← (ts > valTs and ts ≥ ordTs)
  if status val ← newVal; valTs ← ts
  reply [status]
```

Figure 4: FAB's replication algorithm for a single logical block. The function NewTimestamp generates a locally monotonically increasing timestamp by combining the real-time clock value and the brick ID (used as a tie-breaker).

One unusual feature of our protocol is that a request may abort when it encounters a concurrent request with a newer timestamp. In this case it is up to the client or the coordinator to retry. In practice, abortion is rare, given that protocols such as NTP can synchronize clocks with sub-millisecond precision [28, 10]. Being able to abort requests, however, offers two benefits. First, it allows for an efficient protocol—a "read" request can complete in a single round as opposed to two in previous algorithms [5, 27], skipping the round to discover the latest timestamp. Second, abortion enables *strict linearizability*—that is, only by sometimes aborting requests can an algorithm properly linearize requests whose coordinators could crash in the middle. A theoretical treatment of this issue appears in separate papers [11, 2].

4.2 Erasure coding

FAB also supports generic m,n Reed-Solomon erasure coding. Reed-Solomon codes have two characteristics. First, they generate $n-m$ parity blocks out of m data blocks, and can reconstruct the original data blocks from any m out of n blocks. Second, they provide a simple function, which we call Delta, that enables incremental update of parity blocks [30]. Using this function, when writing to a logical block X, the new value of any parity block can be computed by xor(*old-parity*, Delta(*old-x*, *new-x*)), where *old-parity* is the old parity block value, and *old-x* and *new-x* are the old and new values of block X.

Figure 7 shows our data-access algorithm for erasure-coded volumes. Supporting erasure-coded data requires three key changes to the basic replication protocol: segment layout, quorum size, and update logging.

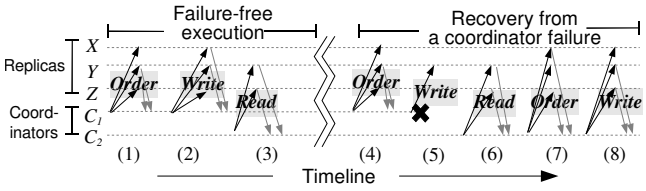

Figure 5: A logical block is replicated on bricks X, Y, and Z. In steps (1) and (2), coordinator C_1 writes to the block in two rounds. Coordinator C_2 reads from $\{Y, Z\}$, discovers that the timestamps are consistent and finishes (in practice, C_2 reads the block value from only one replica; Section 4.4). Steps (4) to (8) show why a write needs two rounds. C_1 tries to write, but crashes after sending *Write* to only Y. Later, while trying to read, C_2 discovers the partial write by observing *valTs*<*ordTs* on Z. C_2 discovers the newest value in step (7) and writes it back to a majority (in fact, all) in step (8), so that future requests will read the same value. In a different scenario, C_2 could contact only $\{X, Z\}$ in step (6), and C_2 would find and write back the old value. This causes no problem—when a write fails, the client cannot assume its outcome.

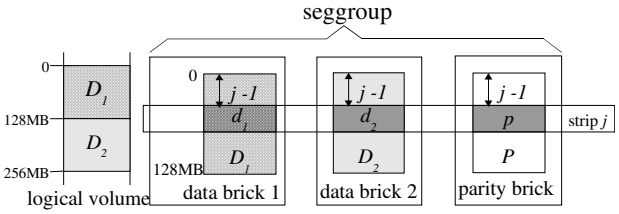

Figure 6: An example of 2,3 erasure-coded segment. An m, n erasure-coding scheme splits one segment into m equal-size chunks (D_1, D_2), and adds $m-n$ parity chunks. A horizontal, block-size-height slice is called a "strip". Bricks in the seggroup maintain the set of timestamps and the update log for each strip. In this example, with a 1KB logical block, the 3rd strip of the segment will occupy regions $\{$(2KB, 3KB), (131074KB, 131075KB)$\}$ of the segment.

We currently use the entire segment as the erasure-code chunk, as shown in Figure 6, unlike typical RAID systems that use smaller chunk sizes such as 64KB. We chose this layout because it lets a large logical sequential request be translated into a large sequential disk I/O at each brick. The downside is that it may abort writes spuriously, when two blocks that happen to be in the same strip are updated concurrently. With a database transaction workload (Section 7.3), however, the conflict rate is measured to be $< 0.001\%$, and we consider that the benefits outweigh the downsides.

As in replication, each request contacts a subset of the bricks that store the segment. However, with m, n erasure coding, a coordinator must collect replies from $m + \lceil (n-m)/2 \rceil$ bricks—that is, the intersection of any two quorums must contain at least m bricks—to be able to reconstruct the strip value during a future "read". We call this quorum system an *m-quorum.* For instance, the m-quorum size is 3 for a $2, 4$ erasure code, and 8 for a $6, 10$ erasure code.

The final change involves the need for strip recovery. Suppose that a "write" coordinator crashes after writing the new value to less than m bricks in the second round. The subsequent "read" request must recover the old value, which might become impossible if the "write" request simply overwrote the blocks and if $n < 2m$ (which is a rather common setting). We solve this situation by *update logging*— a storage brick merely logs the new value in the second round of the "write". A read request, when recovering the old value, scans the log on an m-quorum of bricks and finds the newest strip value that can be fully reconstructed. The "write" coordinator, after it replies to the client, instructs the bricks to overwrite the old block value, and thus compress their log, in an asynchronous Commit phase. In practice, the log is implemented in each brick's NVRAM cache, and the third round—replacing the block value with the log entry—is performed simply by modifying the cache index. Thus, logging does not create any additional disk-I/O or memory-copying traffic in the common case when no brick fails during request processing.

```
// I/O coordinator code. "idx" is the block number within the strip.
proc write(val, idx)
  ts ← NewTimestamp()
  send [Order, {idx}, ts] to bricks in the seggroup
  if an m-quorum reply "yes" and idx'th brick replies with oldval
    delta ← Delta(oldval, val, idx)
    send [Write-EC, val, ts] to the idx'th brick.
    send [Write-EC, NULL, ts] to other data bricks.
    send [Write-EC, delta, ts] to parity bricks
    if an m-quorum reply "yes"
      send [Commit, ts] to bricks in the seggroup
      return OK
  return ABORTED
proc read(idx)
  send [Read] to bricks in the seggroup
  if an m-quorum and idx reply "yes" and all timestamps are equal
    return the val returned by idx'th brick.
  ts ← NewTimestamp() // Slow recovery path begins
  send [Order&ReadLog, ts] to bricks in the seggroup
  ts' ← Pick the largest timestamp that appears in at least m replies.
  strip ← Reconstruct the original strip for ts'
  send [Write, strip[i], ts] to i'th brick, for each i in the seggroup
  if an m-quorum returns "yes"
    send [Commit, ts] to bricks in the seggroup
    return strip[idx]
  return ABORTED

// Storage handler code
when Receive [Write-EC, newval, ts]
  status ← (ts > valTs and ts ≥ ordTs)
  if status
    if this brick is for parity, add [xor(newval, val), ts] to the log.
    elseif newval ≠ NULL, add [newval, ts] to the log.
    else add [val, ts] to the log
  reply status
when Receive [Order&ReadLog, ts]
  status ← (ts > max(valTs, ordTs))
  reply [status, all the log entries]
when Receive [Commit, ts]
  Wait for a while to reject requests with stale timestamps.
  if there is a log entry for ts
    val ← the associated log value.
    Remove log entries with timestamps ts or smaller.
```

Figure 7: Erasure coding algorithm for a single strip. Procedure "write" is invoked by the I/O coordinator to write to the *idx*'th block in the strip. Procedure "read" reads from the *idx*'th block in the strip.

4.3 Reducing the overhead of timestamp management

One challenge of FAB is the timestamp management overhead: for every 1 TB of data, with 24 byte timestamps recorded for every 512B block, 48 GB of space could be required for timestamps. This information must be kept persistently, yet this amount of NVRAM is infeasible. We employ two techniques to reduce the overhead of timestamp management.

First, we observe that timestamps are used only to disambiguate concurrent updates and to recover from previous failures. Thus, when all replicas of a logical block are functional, timestamps can be discarded after all of them have acknowledged an update. Replies

to the client are made as soon as a majority of the replicas have acknowledged an update. The coordinator, in the background, sends a GC (garbage collect) message to bricks only after *all* bricks in the seggroup reply; for erasure-coded volumes, this message is piggybacked onto the Commit message when possible. Each recipient of this message removes the corresponding entry in the timestamp table after waiting for a short period (10 seconds), just long enough to detect out-of-order requests with older timestamps. This period is conservatively chosen to be larger than the maximum clock skew plus the maximum possible scheduling delay on any brick [25].

Another improvement can be made by observing that a single "write" request usually updates multiple blocks, and that each of the blocks affected will have the same timestamp. We thus organize the timestamp table as an ordered tree, with a set of timestamps kept for a range of blocks rather than per-block. When a new request arrives for a part of an existing range in the timestamp table, we then split the range into two (or three) and replace only the part overwritten by the new request.

The combination of these techniques can reduce the timestamp overhead substantially. In the non-failure case, a brick needs to keep timestamps only for blocks that are actively updated. Steady-state size of the timestamp table per brick is measured to be 10KB, which can easily be kept in NVRAM. When a brick fails, the timestamps need to be kept until the reconfiguration protocol removes it from the segment group usually in less than an hour (Section 5). However, simulation results with real workloads show that the timestamp-table size increases by at most 4MB per brick per hour even after brick failure [10]. It is extremely unlikely that the number of timestamps will exceed what a brick can store in memory.

4.4 Improving the efficiency of voting

One of the criticisms of majority voting is its inefficiency, because "read" requests must contact multiple remote nodes [35]. This problem, however, does not apply to FAB for two reasons.

First, we apply an "optimistic read" technique for the common case scenario of reading from a logical block that is already consistent. Here, the coordinator reads the actual block contents (*val*) from an idle, live replica and reads only timestamps from others in the quorum. This technique, in effect, reduces the number of disk accesses to one per "read" request, as timestamps are kept in NVRAM. Second, FAB is naturally a disk-I/O-bound system; the CPU spends much of the time waiting for disk I/Os to complete, so the CPU overhead of timestamp processing does not slow the system down.

4.5 Handling coordinator failures

When a coordinator fails, it is up to the client to connect to a different coordinator and retry. Most enterprise-class storage clients already have such a fail-over capability. Moreover, because of FAB's strict linearizability guarantee, a client can fail over as quickly as it wishes—in fact, it allows a single client to use multiple coordinators concurrently, e.g., in a round-robin fashion.

5. RECONFIGURATION

FAB's reconfiguration protocol changes the quorum configuration of segment groups. It is activated, for example, when a brick failure, recovery, decommissioning, or addition is detected. This protocol and the data-access protocol complement each other—the data-access protocol enables transparent masking of failures or slow bricks, whereas the reconfiguration protocol enables long-term improvement of the system's reliability by allowing the system to tolerate more failures than would otherwise be possible using a fixed-quorum algorithm. For example, Figure 8 shows how a

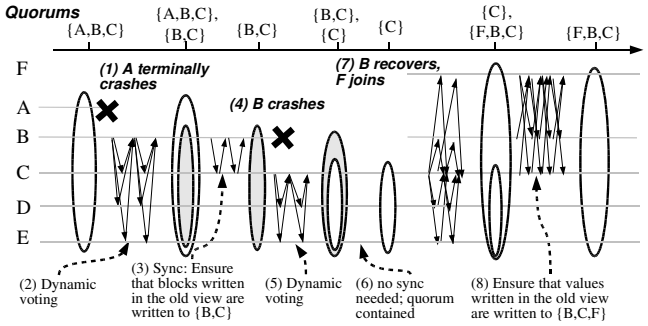

Figure 8: Reconfiguration example. This seggroup initially replicates data on bricks A, B, C, with witnesses D, E participating only in view transition. At the top, the set of active quorums formed at each moment is shown. After A and B crash, C is still able to form a singleton view with help from the witnesses. After B recovers and F is added, they ensure that they store values written to the seggroup before removing the old view $\{C\}$.

3-way replicated seggroup can handle two failures over time using the reconfiguration protocol.

This protocol runs independently for each seggroup in the system. The list of live bricks agreed upon by the members of the seggroup form a *view*; until the view changes, read and write requests that happen in the view must contact an *m*-quorum of the bricks in the view. Figure 8 overviews the reconfiguration protocol. First, a view-agreement protocol lets bricks agree on a new view after brick failure or addition (step (2)). A new view is superposed on the existing view (step (3)), forcing all new requests to collect replies from an *m*-quorum of each of the old and new views. The old view is removed after ensuring that values written in the old view are also written to an *m*-quorum of the bricks in the new view (*state synchronization*; steps (4) and (5)). In the rare event in which more than two views are formed in a short period, they are removed in the FIFO order. By decoupling view formation and state synchronization from foreground request processing, FAB allows client requests to be processed undisturbed. The following sections describe these steps in more detail.

5.1 View agreement via dynamic voting

FAB's view agreement protocol lets bricks in a seggroup agree on a single sequence of primary views; i.e., it ensures that disjoint, concurrent views (a "split-brain" situation) never happen. We use dynamic voting [26][1]—a protocol similar to Paxos [20, 21], but optimized for view agreement—for this purpose.

The participants of the dynamic voting protocol are the set of bricks that store blocks in the seggroup, plus at least two additional *witness* bricks that participate only in the view-agreement protocol (witnesses are chosen randomly when the seggroup is created). Witnesses allow the seggroup to transition views safely, in particular when there are only two storage bricks in the view. We use the phrase *vote view* to refer to this extended set of bricks to distinguish them from a *view*, which is a subset that contains only storage bricks.

This protocol consists of three phases. First, a brick that detects the failure or recovery of another brick becomes a leader and computes a new "candidate" vote view. FAB uses a three-round membership protocol [8] because it settles a new view quickly, but alternatives, e.g., pairwise heartbeats, could also be used. The rest

[1]Caution: The dynamic voting protocol is unrelated to FAB's voting-based data access protocols.

of the protocol ensures that the candidate view indeed ensures a global total order. This is done by having each brick keep the list of *ambiguous views* that are attempted, but not yet fully formed. In the second phase, the leader proposes the candidate view to its members. A recipient accepts the view only if it is a majority of the current view as well as each of the ambiguous views. The recipient also adds the candidate view to the ambiguous-view list. Upon receiving acceptance from all bricks in the candidate view, the leader sends another message to let them update their current view and empty the ambiguous-views lists. When the leader or any other participant dies during this process, another brick becomes a leader and re-runs the protocol.

5.2 Logical-block synchronization

Just forming a new view is not sufficient to ensure consistent accesses to volumes. Before removing the old view, bricks must perform *state synchronization*. Consider a seggroup replicated on five bricks, b_1 to b_5 (witnesses are immaterial in this scenario). The initial view contains all five bricks. Write request W completes, storing the value on bricks b_3, b_4 and b_5. Bricks b_4 and b_5 then fail simultaneously, and a new view $\{b_1, b_2, b_3\}$ is formed. Here, the value of W must be written to at least a majority of the new view before the old view is discarded. Otherwise, a read request might contact only b_1 and b_2 and miss W.

Figure 9 shows the basic state synchronization algorithm (due to space constraints, we show it only for replicated volumes). This protocol resembles "recovery read" that runs after an incomplete write is found (Figure 4), with one difference: it leaves *ordTs* unchanged in the first phase, because this operation itself need not be linearized. This change also avoids aborting new I/O requests by clients.

After the state synchronization finishes, the reconfiguration leader sends out a RemoveView message to let bricks discard the old view. When the reconfiguration leader dies during state synchronization, another brick will restart the view-agreement protocol. However, the blocks already synchronized by the former leader need not be re-synchronized again, and the total amount of synchronization needed after a failure stays constant even when the protocol restarts.

An I/O coordinator learns the list of active views in the seggroup by initially assuming that all bricks in the seggroup are alive. When a storage brick notices that the coordinator's knowledge of the views is stale, it piggybacks its own view list on the reply. The coordinator updates its active-view list transitively, until it receives replies for the I/O request from an m-quorum of every view in the list.

5.3 Streamlining synchronization

The basic algorithm described so far can, in fact, be vastly optimized in many of the common situations. We describe two techniques used in FAB.

5.3.1 Exploiting the quorum containment property

Quorum containment happens when every quorum in the old view is a superset of another quorum in the new view. We can skip block synchronization altogether if this condition is satisfied. This happens, in particular, when a brick fails in a two-brick view, as exemplified in step (6) of Figure 8.

5.3.2 Embedding the respondents in the timestamp table

We piggyback additional information on the optional third background phase of the "write" request (Section 4.3) to let each storage brick remember the set of bricks that have successfully executed the second Write phase. This set is stored in the timestamp table, in-line with the timestamps for the block.

This information can be used to distinguish blocks in the timestamp table that need to be synchronized before the old view can be removed (called the *"must" blocks*), and those blocks that could wait (*"may" blocks*). Specifically, in the FindBlocks phase in Figure 9, each brick returns a block as "must" only when the respondents-set is not a quorum of the new view. If the set is a quorum of, but not the superset of the new view, then the block is returned as "may" ("may" block are still synchronized so that bricks can remove entries from the timestamp tables; Section 4.3). Otherwise, the block need not be synchronized at all. This technique often allows the system to remove an old view very quickly and then synchronize "may" blocks at a leisurely speed. We will examine the effect of this technique in Section 7.5.

```
proc synchronize(sgid, newView, oldView)
  blocks ← findBlocksInTimestampTable(sgid)
  foreach block in blocks
    send [SyncPoll, block] to bricks in oldView
    Wait until an m-quorum in the oldView reply
    maxValTs, maxVal ← Pick the maximum valTs
        and corresponding value from the replies
    maxOrdTs ← Pick the maximum ordTs
        from the replies.
    send [SyncWrite, block, maxValTs, maxOrdTs, maxVal]
        to bricks in newView
    Wait until an m-quorum in newView reply.
proc findBlocksInTimestampTable(sgid, oldView)
  send [FindBlocks, sgid] to bricks in oldView
  Wait until an m-quorum in the oldView reply
  return the union of all blocks in the replies

when Receive [SyncPoll, block]
  return [valTs, ordTs, val] for the block
when Receive [SyncWrite, newValTs, newVal, newOrdTs]
  if newOrdTs > ordTs then ordTs ← newOrdTs
  if newValTs > valTs then
    valTs ← newValTs
    val ← newVal
when Receive [FindBlocks, sgid]
  return block numbers in the timestamp table for seggroup sgid.
```

Figure 9: State synchronization after a view change. This protocol runs independently for each segment group in the system.

5.4 Handling permanent changes

The mechanisms described in the previous section can also be used to remove bricks permanently or add bricks to the system. To handle such events, the system administrator chooses a random brick as the reconfiguration leader and informs it that a failed brick has no hope of recovering. For each affected seggroup, the leader runs the dynamic voting protocol and creates a new view that excludes the dead brick and adds a new brick. After the old view is removed, the leader issues a Paxos update to change the seggroup entry of the global metadata. The newly added brick performs the whole-seggroup synchronization, copying every block, not just those in the timestamp tables.

6. CHOOSING THE RIGHT REDUNDANCY SCHEMES

The main trade-offs between replication and erasure-coding involve reliability, capacity efficiency, and performance. Figure 10 compares the expected mean-time to data loss (MTTDL) of a cluster composed of bricks with 3TB capacity each. In order to achieve

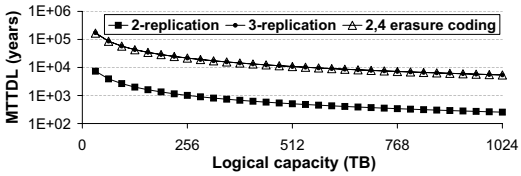

Figure 10: Mean time to first data loss in storage systems using 2-way replication, 3-way replication and 2,4 erasure coding. With 2-way replication, MTTDL is adequate for very small systems but drops rapidly as system size grows. 3-way replication and 2,4 erasure coding have similar MTTDL (the lines are superposed). These provide adequate reliability for commercial use.

our goal of 10,000 years MTTDL, we need at least 3 bricks per logical block using replication. The primary reasons the system requires such high a degree of replication are the use of failure-prone commodity components [4, 3], and the size of the system. A FAB system with a 256TB logical capacity can have over 100 bricks, and the number of combinations of brick failures that can lead to data loss increases with the number of bricks.

Erasure coding can gain higher capacity efficiency than replication, since an m,n erasure coding provides reliability similar to $(n-m+1)$-way replication. For example, a system based on $2,4$ erasure coding provides similar reliability to 3-way replication, but uses the same raw capacity as 2-way replication. The capacity efficiency of erasure-coding-based systems comes at some cost in performance for four main reasons. First, Reed-Solomon encoding and decoding itself consumes CPU cycles. Second, there are fewer disk spindles per logical capacity. Third, a small (strip) write engenders $2(n-m+1)$ disk I/Os in m,n erasure coding, as opposed to $(n-m+1)$ I/Os for the comparable $(n-m+1)$-way replication. Fourth, each request must collect replies from an m-quorum, and the latency is determined by the slowest bricks in the quorum. We will quantify the erasure-coding overhead in the next section.

7. EVALUATION

We have implemented FAB on Linux. The prototype consists of 80,000 lines of C++ code, of which 25,000 lines are for the core replication, erasure coding, and reconfiguration protocols. The global metadata and diskmap tables are implemented as in-memory tables backed up by Berkeley DB. We emulate NVRAM using a memory-mapped file. This simulated NVRAM is used for two purposes: the timestamp table (Section 4), and the write-back buffer cache. The buffer cache size is set to 512MB.

FAB is a user-space single-threaded program. It uses non-blocking I/O (`poll`) and the SCSI-generic driver [15] to multiplex low-level network and disk I/O requests. This design can control resource usage more precisely than, say, using kernel threads. In particular, we run a lottery scheduler [34] for disk-request queue management to ensure that potentially bursty state-synchronization traffic uses only a fraction (5%) of the disk throughput. With our hardware, FAB is disk-bound; thus, ensuring fair-share accesses to disks suffices to ensure end-to-end fair share between different classes of traffic. We examine the effect of this mechanism in Section 7.5.

7.1 System configurations

A cluster of PCs is used as bricks. Each machine is equipped with two 1GHz Pentium 3 CPUs,[2] 2GB of memory, three Seagate Cheetah 32GB SCSI disks (15K rpm, 3.6ms average seek time),and two Intel Gigabit Ethernet interfaces. They run Debian 3.0 with the Linux 2.4.24 kernel. On each brick, the first 6GB of one disk is used by the host Linux file system, and the remaining 90GB is used for FAB data. Up to 22 machines are used as FAB bricks, and an additional 7 machines are used to generate workloads.

[2]Only one CPU per brick is actively used during the evaluation, because FAB is single-threaded.

7.2 Application performance

We first examine FAB's baseline performance by running applications on a single client on seven different storage platforms. A run of the benchmark consists of three phases: (1) "untar" the Linux 2.6.1 source code, 177MB in size, to a target (ext3) file system [*bulk write*]; (2) "tar" the files back to the local file system [*bulk read*]; and (3) compile Linux on the target file system [*a mix of computation, reads and writes*]. To exclude the effect of the client-side buffer cache, we unmounted the target volume after each step (the unmount latency is included in the numbers).

Table 1 shows the results. Overall, the performance of FAB with 3-way replication is comparable with iSCSI+raw disk, proving that FAB's extra protocol processing adds only a marginal overhead to end-to-end performance. Erasure coded volumes are slower than replication for the reasons discussed in Section 6. The 2,4 code is slower than the 4,5 code because of the cost of erasure encoding and decoding: whereas the 4,5 (i.e., RAID-5) code is a simple bitwise XOR, the 2,4 code involves GF(2^8) arithmetic that requires multiple table lookups for each byte. On our hardware, encoding or decoding 1KB of $2,4$ erasure-coded blocks consumes 50μs of CPU time. Bulk reading (tar) over iSCSI is significantly slower than local disks. We believe that this is because the iSCSI client on Linux (we use the Cisco iSCSI initiator) does not prefetch data aggressively enough to keep reading disks sequentially.

	Untar	Tar	Compile
Local disk	21.76	14.80	318.9
Local RAID 1	22.32	14.64	319.2
iSCSI+raw disk	24.21	24.32	323.9
FAB (3way repl.)	21.57	24.61	316.0
FAB (2,4 erasure code)	38.22	27.81	322.0
FAB (4,5 erasure code)	33.33	26.22	319.5
FAB (3way repl., no cache)	28.34	26.13	327.0

Table 1: End-to-end latency of application programs. The numbers are an average over three runs. "Local disk" and "Local RAID-1" use disks locally attached to the client. "iSCSI+raw disk" uses a remote iSCSI server accessing a local raw disk. "FAB" accesses data through FAB's iSCSI gateway. "FAB (no cache)" shows FAB with its NVRAM buffer cache turned off.

7.3 Scalability

To study how FAB's throughput grows with size, we ran three types of synthetic workloads, because none of the real-world applications that we have can exert enough stress on FAB. The workload *DB*, modeled after SPC-1 [9],[3] simulates a database transaction workload. DB uses three volumes. The first two are data volumes that receive uniformly random as well as database-index-accessing 4KB reads and writes. The third volume, whose size is 1/3 of the other two, receives sequential log writes of size 8KB to

[3]The primary difference between DB and SPC-1 is that SPC-1 defines an open-queue workload with a fixed request arrival rate. DB changes it to run in a closed queue with zero think time to stress the system.

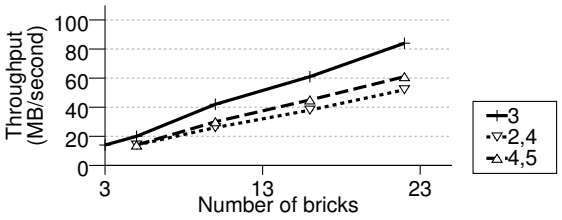

Figure 11: Aggregate throughput of FAB clusters with the DB workload. "3" means three-way replication, "2,4" means 2,4 erasure coding.

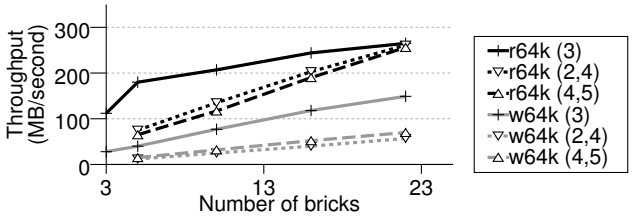

Figure 12: Throughput of FAB with random large read/write workload. The numbers in parentheses show the redundancy policy.

64KB. Overall, DB issues requests with a read:write ratio of 4:6 and an average size of 8KB. We scaled the total logical volume size to be 10*N* GB (*N* is the number of bricks in the cluster)—e.g., in a 22-brick cluster, the two data volumes are 94GB each, and the log volume is 32GB. The other two workloads, r64k and w64k, are random 64KB read and write requests over a volume of size 25*N*GB. The request size of 64KB is taken from SPC-2's proposed data mining and video-on-demand workloads [9]. Each workload is generated by a total of 30*N* threads running in a closed queue with zero think time on seven client machines.

Figures 11 and 12 show the results. Overall, as expected, FAB's throughput scales linearly with the cluster size. The exception is 64KB random reads, which hit a ceiling due to the capacity limits of our Ethernet switches. Erasure-coded volumes sustain much lower throughput than their replicated counterparts, for the reasons discussed in Section 6.

7.4 Performance decoupling

This section compares our replication protocol to the master-slave protocol, the traditional method for replicating data across a network. We have built a variation of FAB that runs a master-slave protocol similar to Petal's [22]. In this protocol, the dynamic voting protocol is used to let bricks agree on the single master for each seggroup. Each I/O coordinator forwards the request to this master. For read requests, the master simply reads its local disk and returns the data to the coordinator. For write requests, the master broadcasts the new value to the replicas in the current view, waits for the replies from all of them, and then returns control back to the coordinator. Freed from timestamp maintenance, this protocol is far simpler than FAB's.

Figure 13 shows the throughput of the two systems on the 22-brick cluster with 3-way replication. Interestingly, for both DB and 64KB-random-write workloads, FAB outperforms the master-slave protocol. This is due to the performance decoupling effect of the voting protocols [24]—specifically, FAB can ignore slow bricks by collecting replies only from a majority. Performance decoupling is especially effective in a disk-bound system like FAB in which

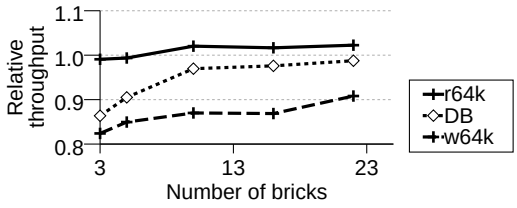

Figure 13: The throughput of the master slave protocol with 3-way replication. The FAB protocol is normalized to 1.0.

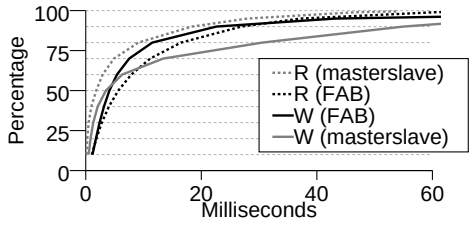

Figure 14: CDF of end-to-end request latency under high load for the DB workload. The master-slave protocol experiences many high-latency "write" requests.

disk accesses, especially NVRAM flushing, often generate bursty disk traffic that slows the brick down for a short period of time. The performance-decoupling effect is visible especially for smaller clusters, in which a single overloaded brick can have a large impact on the overall performance. On the other hand, for 64KB-random-read workloads, the master-slave protocol slightly outperforms FAB in a large cluster due to its simplicity, although this is offset by FAB's ability to read blocks from idle bricks (Section 4.4). These effects can also be observed in the latency distribution as shown in Figure 14.

7.5 Handling changes

This section studies how FAB handles changes to the system. We start a 22-brick cluster with 3-way replication, run the DB workload, and artificially introduce brick failures or recoveries. Figure 15 shows the throughput transition when one brick fails and recovers three minutes later. The brick failure causes a reconfiguration protocol to run, which causes bricks in the affected seggroup to scan their timestamp tables. The CPU overhead of this timestamp-table scan is the reason for the small drop in throughput. No state synchronization is required, however, because for every seggroup affected by the failure, the two bricks that form the new view are already consistent. The system throughput does not decrease noticeably during the crash period, because DB is a write intensive workload—each remaining brick handles the same amount of write traffic per request. After the recovery, another timestamp-table scan happens. Virtually no "must" blocks (Section 5.3.2) will be found, however, as the "write" requests issued during the crash period to an affected seggroup will be written to the remaining two, and these two form a quorum in the new, full view. Thus, the old-view removal happens nearly instantaneously after recovery. The synchronization of the "may" blocks, i.e., copying blocks written during the crash period to the recovered block, happens slowly in the background over the next 2.5 minutes, due to the lottery scheduling. Overall, no client-visible I/O error happens during the run. Note that our current client software cannot handle session termination gracefully; thus, for the experiments in

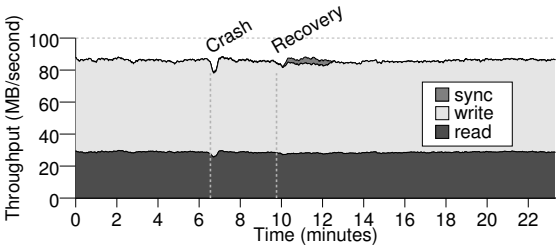

Figure 15: A brick fails then recovers in a 22-brick FAB cluster running the 3-way quorum-based replication protocol under the DB workload. FAB can mask the failure without causing any I/O errors.

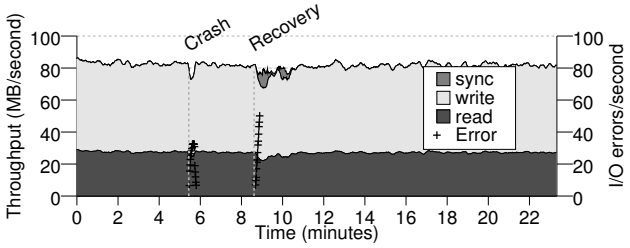

Figure 16: A brick fails and then recovers in a 22-brick FAB cluster running the 3-way master-slave replication protocol. The "error" marks show the number (not megabytes) of I/O errors encountered by clients.

this section, we set up the clients not to use failed brick(s) as I/O coordinators.

In contrast, Figure 16 shows the same scenario, but using the master-slave replication protocol. After the failure and recovery, the throughput drops, not because of timestamp scanning but because "write" requests to seggroups that contain the failed brick abort until the new view is formed 10 seconds later. This is evident from the "error" marks in the graph. The performance drop is suppressed in this graph, because our DB workload generator does not initiate the recovery activities, e.g., device resetting and database log recovery, that usually happen after I/O failures—the clients simply retry after waiting for a second.

Figure 17 shows a double-failure scenario for FAB's quorum protocol. Two bricks fail within two minutes and then recover. After the second failure, there is a single seggroup in the system whose view size changes from two to one. This causes requests to this seggroup to abort until the new view is formed (the quorum size of a two-brick view is always two). Recovery causes a little more disruption, because the amount of state that needs to be synchronized doubles. However, after about 5 minutes, state synchronization finishes and the throughput is restored back to the original level.

Figure 18 shows FAB's reaction to permanent failures. A brick fails, and is declared permanently dead four minutes later. For each seggroup that includes the dead brick, another brick replaces the dead one. These newly added bricks need to copy the existing data, which consumes a steady portion of the disk traffic. No I/O errors occur during this scenario. With DB running at full speed, as in this picture, it takes about 1.5 hours to fully bring the new bricks up to date. Without any foreground traffic, the disk synchronization finishes after 25 minutes.

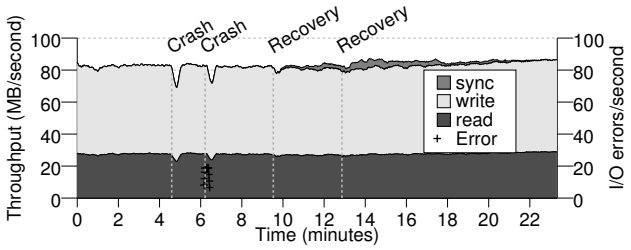

Figure 17: Handling double failures in a 22-brick FAB cluster. The second failure causes an I/O error on a segment group that contains both the failed bricks. After the new view settles, these segment groups can continue handling requests with one remaining brick.

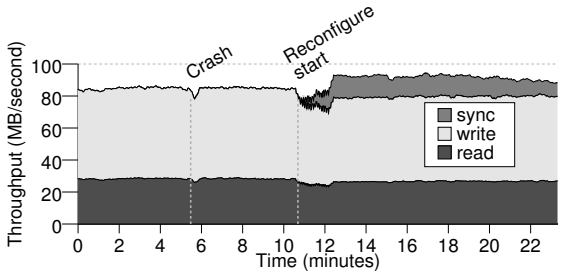

Figure 18: A brick fails, and five minutes later, it is declared dead and the affected segment groups are re-balanced across surviving bricks.

8. CONCLUSION

This paper has described the design, implementation, and evaluation of FAB. FAB achieves two key requirements of enterprise storage systems, stable, continuous service and high reliability, using two new mechanisms. First, it uses a voting-based protocol to guarantee linearizable accesses to replicated or erasure-coded logical blocks. This protocol transparently masks failures, and offers better throughput than traditional master-slave replication by masking temporary overload conditions. Second, FAB deploys a dynamic quorum-reconfiguration protocol to allow the system to react to brick additions or decommissioning without disrupting clients.

Acknowledgements

We thank Marcos Aguilera, Minwen Ji, Beth Keer, Hernan Laffitte, Craig Soules, and John Wilkes for their help and input to the project.

9. REFERENCES

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *5th Symp. on Op. Sys. Design and Impl. (OSDI)*, Boston, MA, USA, December 2002.

[2] Marcos K. Aguilera and Svend Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Labs, December 2003.

[3] Dave Anderson, John Dykes, and Erik Riedel. More than an interface—SCSI vs. ATA. In *USENIX Conf. on File and*

Storage Technologies (FAST), San Francisco, CA, March 2003.

[4] Satoshi Asami. *Reducing the cost of system administration of a disk storage system built from commodity components*. PhD thesis, University of California, Berkeley, May 2000. Tech. Report. no. UCB-CSD-00-1100.

[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[6] Pei Cao, Swee Boon Lin, Shivakumar Venkataraman, and John Wilkes. The TickerTAIP parallel RAID architecture. *ACM Trans. on Comp. Sys. (TOCS)*, 12(3):236–269, 1994.

[7] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[8] Flaviu Christian and Frank Schmuck. Agreeing on processor group membership in asynchronous distributed systems. Technical Report CSE95-428, UC San Diego, 1995.

[9] Storage Performance Council. SPC Benchmark 1 specification. http://www.storageperformance.org/, 2003.

[10] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: Enterprise storage systems on a shoestring. In *8th Workshop on Hot Topics in Operating Systems (HOTOS-VIII)*, pages 169–174, Kauai, HI, USA, May 2003.

[11] Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. A decentralized algorithm for erasure-coded virtual disks. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 125–134, Florence, Italy, June 2004.

[12] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman. Self-* storage: Brick-based storage with automated administration. Technical Report CMU-CS-03-178, Carnegie Mellon University, August 2003.

[13] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *8th Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS-VIII)*, pages 92–103, San Jose, CA, USA, October 1998.

[14] David Gifford. Weighted voting for replicated data. In *7th Symp. on Op. Sys. Principles (SOSP)*, pages 150–162, Pacific Grove, CA, USA, December 1979.

[15] Douglas Gilbert. The Linux SCSI generic HOWTO. http://www.torque.net/sg/p/sg_v3_ho.html, 2003.

[16] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Efficient consistency for erasure-coded data via versioning servers. Technical Report CMU-CS-03-127, Carnegie Mellon University, April 2003.

[17] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Prog. Lang. and Sys. (TOPLAS)*, 12(3):463–492, July 1990.

[18] Andy Huang and Armando Fox. Dstore: self-managing, crash-only persistent hash table. http://swig.stanford.edu/public/projects/dstore/, 2004.

[19] IBM. IceCube: storage server for the Internet age. http://www.almaden.ibm.com/cs/storagesystems/IceCube/, 2003.

[20] Leslie Lamport. The part-time parliament. *ACM Trans. on Comp. Sys. (TOCS)*, 16(2):133–169, 1998.

[21] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[22] Edward K. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. In *7th Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS-VII)*, pages 84–92, Cambridge, MA, USA, October 1996.

[23] LeftHand Networks. IP-based storage area networks. http://www.lefthandnetworks.com/downloads/ip-san_wp.pdf, 2002.

[24] Benjamin C. Ling, Emre Kiciman, and Armando Fox. Session state: beyond soft state. In *1st Symp. on Network Sys. Design and Impl. (NSDI)*, pages 295–308, San Francisco, CA, USA, March 2004.

[25] Barbara Liskov, Liuba Shrira, and John Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Trans. on Comp. Sys. (TOCS)*, 9(2):125–142, 1991.

[26] Esti Yeger Lotem, Idit Keidar, and Danny Dolev. Dynamic voting for consistent primary components. In *16th Symp. on Princ. of Distr. Comp. (PODC)*, pages 63–71, Santa Barbara, CA, USA, August 1997.

[27] Nancy A. Lynch and Alex A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *16th Int. Conf. on Dist. Computing (DISC)*, pages 173–190, Toulouse, France, October 2002.

[28] David L. Mills. Improved algorithms for synchronizing computer network clocks. In *ACM SIGCOMM*, pages 317–327, London, United Kingdom, September 1994.

[29] Brian Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly available distrbuted systems. In *7th Symp. on Princ. of Distr. Comp. (PODC)*, pages 8–17, Toronto, ON, Canada, August 1988.

[30] James S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software—Practice and Experience*, 27(9):995–1012, 1997.

[31] Sean Reah, Patrik Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore prototype. In *USENIX Conf. on File and Storage Technologies (FAST)*, pages 1–14, San Francisco, CA, March 2003.

[32] Julian Satran, Kalman Meth, Constantine Sapuntzakis, Mallikarjun Chadalapaka, and Efri Zeidner. RFC3720: Internet small computer systems interface (iSCSI). http://www.faqs.org/rfcs/rfc3720.html, 2004.

[33] Josh Tseng, Kevin Gibbons, Franco Travostino, Curt Du Laney, and Joe Souza. Internet storage name service (iSNS), draft version 18. http://www.diskdrive.com/reading-room/standards.html, March 2003.

[34] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible propotional-share resource management. In *1st Symp. on Op. Sys. Design and Impl. (OSDI)*, pages 1–11, Monterey, CA, USA, November 1994.

[35] Avishai Wool. Quorum systems in replicated databases: science or fiction? *Bull. IEEE Technical Committee on Data Engineering*, 21(4):3–11, December 1998.