FAB: enterprise storage systems on a shoestring

Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence and Alistair Veitch
Storage Systems Department, Hewlett-Packard Laboratories, Palo Alto, CA
{frolund,arif,ysaito,suspence,aveitch}@hpl.hp.com

Abstract—A Federated Array of Bricks (FAB) is a logical disk system that provides the reliability and performance of enterprise-class disk arrays, at a fraction of the cost and with better scalability. The unit of deployment in FAB is a *brick*, a small rack-mounted storage appliance built from commodity components including disks, a CPU, NVRAM, and network cards. Bricks federate themselves in a completely decentralized manner to provide users with a set of logical volumes. This paper motivates FAB and introduces our data replication algorithm based on majority-voting. We argue that majority voting is practical for ultra-reliable, high-throughput storage systems like FAB, and present several techniques that improve both the performance and space overhead of our protocol.

1 Introduction

Disk arrays are today's standard solution for enterprise-class storage systems. The key requirement that separates disk arrays from consumer-class storage systems is their absolute reliability: a disk array must never lose data or stop serving data, under any circumstances short of complete disaster. To fulfill this requirement, disk arrays are constructed from customized, very reliable, hot swappable hardware components. Designing and building the hardware components is time-consuming and expensive, and this, coupled with relatively low manufacturing volumes, is a major factor in the high price of storage systems—high-end arrays retail for many millions of dollars.

Another cost factor, and problem for customers, is the lack of scalability of a single system. There is a high up-front cost for even a minimally configured array, and a single system is limited in both capacity and throughput. Many customers exceed these limits, resulting in poor performance or a requirement to purchase multiple systems, both of which increase management costs. The lack of scalability forces manufacturers to build multiple products, or even entire product lines, each targetted at different system scales. For example, Hewlett-Packard sells three different array lines, each of which effectively multiplies the engineering effort required — for hardware and firmware development, for integration and testing, etc, costs which are reflected in the price paid for each system.

A Federated Array of Bricks (FAB) is a low-cost alternative to disk arrays, and is designed to be scalable from very small to very large systems. FAB achieves this by composing together storage *bricks*, where each brick is a small rack-mounted storage appliance built from commodity components including disks, a CPU, NVRAM, and network cards. FAB systems cost much less than disk arrays to manufacture and develop, due to the economies of scale inherent in volume production, and because FAB can replace entire array product lines (amortizing development costs). Because of these factors, we anticipate that a FAB system can be built for far less than the equivalent high-end system. FAB provides comparable reliability, achieved through replication: we store the same disk block on multiple bricks, and we create redundant paths between all components of the system. FAB performance scales by completely distributing all functionality (no centralized bottlenecks) across the set of available bricks.

1.1 FAB: challenges and overview of solutions

We have identified the following key challenges to building a large, completely distributed storage system:

Failure tolerance: FAB is built from commodity hardware, which has empirically been found to be less reliable than the hardware used for enterprise systems [3, 2]. Every component—disks, bricks, networks can and will fail. FAB must seamlessly handle failures without data loss or delays in response to client requests.

Single-copy consistency: We must ensure that a replicated disk block logically looks like a single, highly available block to the client, even though there is no centralized software that oversees the I/O activities of the entire system.

Asynchronous coordination: We cannot rely on disks and operating systems to always act in a timely manner—e.g., an I/O request to a busy disk is known to sometimes take more than 5 seconds to complete (i.e., stuttering failures). Thus, we must coordinate replicas without any assumptions about their speed or network connectivity.

Hardware heterogeneity: As disk and CPU technologies evolve over time, the design of bricks will also evolve. We must let customers deploy different types of bricks incrementally as their demand grows. FAB must assign resources to volumes in a way that maximizes overall performance and reliability.

FAB uses a quorum-based replication scheme, as described in Section 3, to address the first three challenges. In FAB, a “read” or “write” request completes when a majority of the replicas are functional. We recover from failures lazily, repairing the replicas during the next “read” request without any lock-step synchronization. Our protocol does not rely on failure detection—it just ignores dysfunctional components. It tolerates non-Byzantine failures, including network partitioning and stuttering failures. FAB uses dynamic load balancing and background reconfiguration, as discussed further in Section 4, to address the fourth challenge.

1.2 Related work

The idea of building a distributed logical disk from a decentralized collection of smaller components was pioneered by DataMesh [13] and Petal [10]. FAB extends Petal’s ideas with better replication, volume layout, and load balancing algorithms. IBM’s IceCube [8] builds innovative hardware for a FAB-like composable storage system, but we do not know their software structure yet.

Many high-throughput data systems, including Petal and most relational database systems, use some form of primary-backup replication. They fail to solve the challenges outlined in the previous section. In these systems, a failure of the primary renders its data unavailable until a new primary is elected. The actual fail-over time in these systems can be quite substantial. Having too short a fail-over time increases the chances of electing a new primary before the old primary has actually failed, consequences of which range from severe performance degradation (as in some group membership protocols) to outright data corruption (as in a naïve timeout-based failure detection scheme). Thus, in practice, these systems must conservatively choose a large fail-over period, often longer than 30 seconds, which actually causes the clients to time out.

The goal of [1] is to allow clients of a storage-area network to directly execute RAID encodings across distributed storage devices. This algorithm relies on the ability of clients to accurately detect the failure of storage devices. Moreover, the algorithm in [1] can result in data loss when certain combinations of client and device failures occur. In contrast, our algorithm can tolerate the simultaneous crash of all bricks, and it can make progress whenever a majority recover and are able to communicate.

Numerous replication protocols use majority voting. The protocol by Thomas [12] is similar to ours in that it uses timestamps to order “write” requests against a majority of replicas. However, Thomas’ protocol only guarantees convergence, which is too weak for a distributed logical disk. FAB guarantees linearizability [7]. Several state-machine replication algorithms, such as Paxos [9], use majority voting to achieve a total order for requests. Our replication protocol exploits the semantics of read and write operations to achieve the same thing in fewer rounds, using less space. Messaging-based atomic register algorithms [4, 11] resemble our algorithm the most; they use majority voting and exploit read and write operation semantics. These algorithms, however, require more rounds (especially in the common case) than ours and lack support for process recovery.

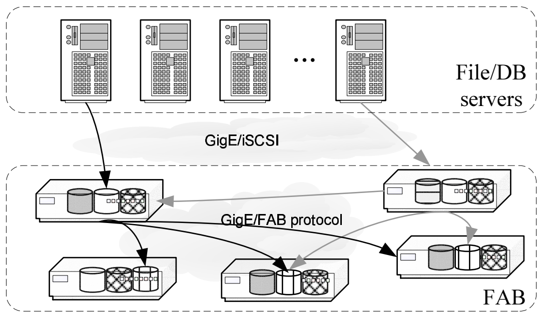

Figure 1: A typical FAB structure. Client computers connect to the FAB bricks using standard protocols. Clients can issue requests to any brick to access any logical volume. The bricks communicate among themselves using our replication protocol.

2 Structure of FAB

Figure 1 shows the structure of FAB. Client systems connect to FAB bricks using standard protocols such as Fibre Channel or iSCSI. Bricks are connected to each other using standard local-area networks, such as 1 Gbps Ethernet. FAB presents the clients with a number of logical volumes, each of which may be accessed transparently as if it were a single disk. Since FAB is a decentralized system without a central management node, a client can ask any brick to create, resize, or access a logical volume. Bricks use a custom protocol, described in the next section, to coordinate among themselves and provide a consistent view of volumes. A single FAB system is anticipated to contain up to 5000 bricks with a logical capacity of 2 petabytes.

FAB internally splits each logical volume into fixed-size *segments*. Each segment contains a number of *blocks*, the minimum unit of access. The segment size and block size default to 8GB and 1KB, respectively. A number of segments are gathered into *groups*. Each group is replicated over several bricks (three by default: see below), chosen randomly out of the set of bricks with available space. The use of segments and groups enables efficient metadata management; segments are used in layout management and groups for replication and availability.

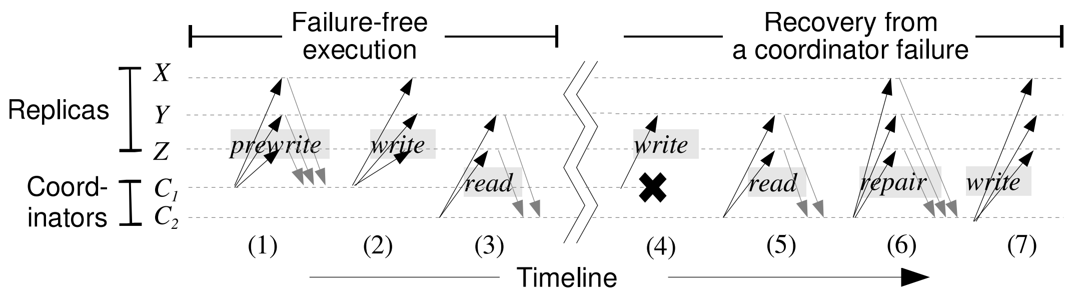

Figure 2: In this example, a disk block is replicated on three bricks, X, Y, and Z. Two coordinators, C_1 and C_2, issue requests to the replicas. The first scenario (steps 1 to 3) shows a failure-free execution. Brick C_1 writes to the block in two rounds. In the *prewrite* round, the replicas update their *logTs* to indicate a new ongoing update and promise not to accept any request older than this request. In the *write* round, the replicas actually write the new value to their disks and sets *ts* to indicate that the update is complete. In step (3), node C_2 reads blocks from a majority of $\{X,Y,Z\}$, discovers that the block contents are consistent and finishes (in practice, C_2 reads the block value from only one replica; see Section 3.1.) The second scenario (steps 4 to 7) shows why the *prewrite* round is needed. Here, C_1 tries to write, but crashes just after sending the *prewrite* to only Y. Later, while trying to read, C_2 discovers that $ts \neq logTs$ on Y; i.e., the replicas are inconsistent because of an incomplete write request. C_2 runs the *repair* round in step (6) on a majority of the replicas to discover the newest value, and writes it back to (at least) the majority of the nodes in step (7), so that future requests will never read older values.

Each brick internally runs three software modules: the *coordinator* module that receives client requests and coordinates disk read or write requests on behalf of clients, the *block-management* module that actually reads and writes disk blocks, and the *configuration-management* module that oversees administrative changes. The next section describes the interaction between the coordinator and block-management modules. The configuration-management module uses the Paxos distributed consensus algorithm [9] to replicate configuration information—e.g., the set of bricks that exist in the system, and the name and layout of logical volumes—on all bricks. We plan to investigate a more efficient configuration management scheme in the future.

The choice of a redundancy scheme to provide high reliability at an acceptable cost is an important consideration for FAB. We considered erasure coding and replication. Erasure coding has high space efficiency and high reliability, but poor performance when bricks fail (reads access multiple bricks and writes require a read before writing). Instead, FAB uses replication of data across multiple bricks to overcome brick failures. We compared the reliability of 2-way and 3-way replication schemes by computing their mean unavailability and MTTDL — the mean time, starting from a system with no failures, that some data is lost. We used component failure rates from Asami [3] and assumed that data is lost when all bricks holding replicas of any segment fail. We considered a FAB system of 256 bricks; each brick uses RAID-5 across 12 SATA disks holding replicas of 128 data segments. Each segment group contains 32 segments. For 2-way replication, we estimated a MTTDL of 267 years and a mean unavailability of 0.02% (1.8 hours/year) which is inadequate for critical applications. For 3-way replication, we computed a MTTDL of 1.3 million years and a mean unavailability of 3×10^{-6}% (1 second/year), which is acceptable. Based on these considerations, we chose 3-way replication as the default policy, but we also allow administrators to choose other replication factors.

Creating three replicas for each logical block sounds expensive, but is only 33% more capacity intensive than RAID-10; and FAB systems are build from cheaper components than existing high-end disk arrays, which reduces the cost substantially. In Section 5, we discuss our plan to use "witness" replicas to reduce the effective storage consumption, while maintaining an acceptable level of reliability.

3 The FAB replica-management protocol

This section describes FAB's approach to replica consistency. We first introduce the basic protocol for maintaining replicated blocks. Later sections describe performance optimizations and extensions to optimize the system's performance and memory consumption.

In FAB, an I/O request to logical blocks is handled by the coordinator module of any brick (from a clients view, every brick can act as a disk array controller). FAB runs a variation of a timestamp-based majority-voting protocol [12]. The full details of the protocol and a correctness proof are presented in [5].

Figure 2 shows an example. The task of the coordinator is straightforward in theory: when writing, it generates a new timestamp and writes the value and timestamp to the majority of replicas; when reading, it reads from the majority and returns the value with the newest timestamp.

The failure of the coordinator itself, however, causes a

Workload	*date*	*length*	*volume size*	*#writes*	*#reads*	*data written*	*data read*	*unique data written*
Cello	9/2002	1 day	1.4 TB	5,250,126	6,766,002	67.4 GB	160 GB	27.1 GB
SAP	1/2002	15 min	5 TB	150,339	4,835,793	1.75 GB	55.4 GB	1.36 GB
OpenMail	10/1999	1 hr	7 TB	931,979	355,962	61.3 GB	2.47 GB	1.64 GB

Table 1: Workload characteristics. *Date* shows when the trace was collected. *Unique data written* is the amount of data written once overlapping writes are removed.

problem, because it may leave a new value on a sub-majority of the replicas. A logical disk system must ensure *linearizability* [7]—roughly speaking, all clients must see a single global ordering of (either successful or failed) read and write requests for each logical block, even when these requests are coordinated by different bricks. Thus, after a coordinator failure, future "read" requests on the same block must all return the old block value or all return the value attempted by the failed coordinator (unless the block is overwritten by a newer "write" request). Previous approaches, such as those using two-phase commits [6], cannot ensure a quick fail-over. FAB takes an alternative approach, performing recovery in a lazy manner when a client tries to read the block for the first time after the failure.

To detect the partial writes that result from failures, each replica of a logical block keeps two timestamps: the *ts* is the timestamp of the value currently stored, whereas the *logTs* is the timestamp of the newest ongoing "write" request. As illustrated in Figure 2, a "write" request runs in two phases, using the timestamps to ensure linearizability. A "read" request usually runs in one phase, but takes three phases when timestamp state indicates the past failure of a (different) coordinator: the value with the highest timestamp is stored in a majority with a timestamp greater than that of any previous writes, including any partial writes.

We assume that clients have multi-path capability, so the failure of a coordinator does not stop them issuing requests to FAB. Since FAB requires no change to clients, the clients' own software and the standard protocol used between clients and FAB dictate client reaction to coordinator failure.

3.1 Improving the efficiency of majority voting

Majority voting has been proposed as a simple yet robust replication method for quite a while [6, 12], but no system has used it in a high-throughput environment. The often-cited reason is that it is inefficient, because "read" requests must contact multiple remote replicas [14]. This reason, however, does not apply to FAB for the 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 from one idle 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 the vast majority of timestamps will be cached in main memory for the reasons described in the next section.

Secondly, FAB is a naturally disk-I/O-bound system; the CPU and network spend most of the time waiting for disk I/Os to complete. The overhead of extra timestamp processing does not slow the system down.

3.2 Reducing the overhead of timestamp management

One challenge that FAB faces is the timestamp management overhead: for every 1 TB of data, with a 12 byte timestamp recorded for every 1KB block, 12 GB of space could potentially be required to store timestamps. This information must be kept persistently, yet this amount of NVRAM is infeasible. We employ several techniques to reduce the overhead of timestamp management substantially.

First, we observe that timestamps are used only to disambiguate concurrent updates and to "repair" the results of previous failures. Thus, in the case where all replicas of a logical block are functional, timestamps can be discarded once all the replicas 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, runs a third phase to write processing in which it lets replicas remove their timestamps once all have replied. In the normal case, a brick needs to keep timestamps only for blocks that are actively updated; these timestamps can easily be kept in NVRAM. After one of the replicas fails, other replicas must keep timestamps for blocks that are updated, until the replica recovers or a reconfiguration starts. However, as we show below, it is extremely unlikely that the number of these timestamps exceeds what a brick can store in memory.

A second optimization can be made by observing that a single "write" request almost always updates multiple blocks, and that each of the blocks affected will have the same timestamp. We thus keep timestamp information on ranges of blocks, rather than per-block.

To investigate the impact these optimizations would have, and to determine the actual amount of memory needed, we have analyzed several real-world I/O traces, summarized

Workload	*raw*	*written*	*multiple*
Cello	16.8 GB	26.4 MB	2.28 MB
SAP	60 GB	128 MB	10.3 MB
OpenMail	84 GB	38.4 MB	4.00 MB

Table 2: Timestamp memory requirements. *Raw* is the amount of memory required with a timestamp for every 1KB block, *written* is the amount required if timestamps are only kept for blocks written during the trace, *multiple* is the amount if a timestamp covers multiple blocks. For sparse structures, we assume a doubling over the raw timestamp size to allow for the data structure overhead. All numbers except for *raw* (which does not vary over time) have been normalized to be per-hour, i.e., the amount of memory required for every hour of operation while a replica is unavailable. This normalization results in pessimistic estimates—as significant spatial locality is shown over time, the rate of timestamp generation decreases when longer time periods are observed. For a longer, 3 day "Cello" trace, the rate of timestamp generation is half of that shown by the 3rd day.

in Table 1.

Cello: A file system managed by an 8 processor HP9000 N4000 for 20–30 researchers, with 16 GB of RAM, and an HP XP512 disk array.

SAP: A SAP ISUCCS 4.5B and Oracle supporting 3000 users and several background batch jobs running on an HP V2500 with an HP XP512 disk array.

OpenMail: An HP9000 K580 server with 6 CPUs, 3.75 GB of RAM, and an EMC 3700 Symmetrix disk array. Approximately 2000 users access their email during the course of the trace.

Table 2 shows the amount of memory required for timestamp information under various circumstances. These results show that even if a system is down for several days, even a relatively modest amount of memory will be sufficient to store all the timestamps needed.

4 Data layout and load balancing

As discussed in Section 2, FAB chooses the set of replicas for each segment randomly. The randomized layout has many advantages over a deterministic mapping such as chained declustering [10]: the load is uniformly distributed over the bricks; when bricks leave FAB, reassigning the segment replicas on these bricks to other bricks is straightforward; similarly, when new bricks join FAB, reassigning segment replicas from heavily loaded bricks to the new bricks is easy to do; and non-homogeneous bricks with different capacities and performance characteristics can be handled. Moreover, by using reasonably large segments (say, 8GB) the size of the layout table can be kept small.

The FAB replica-management protocol permits actual data reads to be made from any replica; the other replicas only provide timestamp information. By having coordinators perform the data read from the least loaded replica, load can be shifted away from a heavily loaded brick to its neighbors (bricks holding replicas of the segments on the heavily loaded one), which can further shift read load to their neighbors, and so on, until the entire FAB shares the load. This mechanism makes it possible to accommodate bricks with heterogeneous performance. It also makes FAB highly resilient to load imbalance due to brick failures, since the read load from failed bricks is automatically spread over the entire FAB. More intelligent load balancing may be considered in future work.

5 Current status and future work

We have implemented a prototype and are studying its behavior under various situations, including failures and overloads.

We identify two major areas of future work. One is dynamic volume reconfiguration after failures or to improve performance. The requirement remains the same: linearizability, asynchronous coordination, and no service stoppage during reconfiguration. We plan to adapt the technique described in [11], by superimposing a new quorum configuration using Paxos, transferring contents to new bricks, and garbage collecting old quorum configurations in the background.

The other is reducing the storage overhead of quorum-based replication using *witnesses* and *witness promotion*. We adapt the timestamp-discarding scheme introduced in Section 3.2 to create "witness" replicas that only keep timestamps, but no actual block values (at least in the long term). By replicating a logical segment on only $f+1$ normal replicas and f additional witnesses, the segment can tolerate f failures with little space overhead. A witness participates in the block-I/O protocol exactly like other replicas—it actually stores block contents in a scratch disk area. When it receives a "discard timestamps" request for a block, however, it recycles the disk area. In the common case where all replicas are functional, a witness only consumes disk blocks for ongoing updates. When a witness accumulates too many disk blocks for outstanding updates after a failure of another replica, it eventually promotes itself into a full-fledged replica using the aforementioned reconfiguration algorithm.

In the longer term, we also need to evaluate the full cost of ownership of a FAB system, including maintenance in the face of the higher expected rate of component failures, power and cooling expenses, and to address the potential engineering problems of a FAB system at the scale of 5000 bricks.

6 Acknowledgements

We would like to thank other members of the HP Labs Storage Systems Department for feedback, particularly John Wilkes. Our shepherd, Jeff Chase, provided valuable comments that improved the paper.

References

[1] Khalil Amiri, Garth A. Gibson, and Richard Golding. Highly concurrent shared storage. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS 2000)*, Taipei, Taiwan, April 2000.

[2] D. Anderson, J. Dykes, and E. Riedel. More than an interface—SCSI vs. ATA. In *USENIX Conf. on File and Storage Technologies (FAST)*, San Francisco, CA, March 2003.

[3] S. 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.

[4] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[5] Svend Frolund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. Building storage registers from crash-recovery processes, May 2003. Tech report HPL–SSP–2003–14, available at http://www.hpl.hp.com/research/ssp/papers/.

[6] D. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th. Symposium on Operating Systems Principles*, 1979.

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

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

[9] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001. http://research.microsoft.-com/users/lamport/pubs/pubs.html.

[10] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th Int. Conf. on Architectural Support for Prog. Lang. and Op. Systems*, pages 84–92, Cambridge, MA, 1996.

[11] N. A. Lynch and A. 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.

[12] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Sys. (TODS)*, 4(2):180–209, June 1979.

[13] John Wilkes. Datamesh research project, phase 1. In *Proc. USENIX Workshop on File Systems*, pages 63–69, Ann Arbor, MI, May 1992.

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