From Paxos to CORFU: A Flash-Speed Shared Log

Dahlia Malkhi
Microsoft Research
Silicon Valley
dalia@microsoft.com

Mahesh Balakrishnan
Microsoft Research
Silicon Valley
maheshba@microsoft.com

John D. Davis
Microsoft Research
Silicon Valley
john.d@microsoft.com

Vijayan Prabhakaran
Microsoft Research
Silicon Valley
vijayanp@microsoft.com

Ted Wobber
Microsoft Research
Silicon Valley
wobber@microsoft.com

Introduction

In a 2006 talk [11], Jim Gray predicted that "flash is the new disk"; five years later, indeed, flash is making headway into data centers, but it is usually disguised as standard block storage. We posit that new abstractions are required to exploit the potential of large-scale flash clusters. It has been observed that each individual flash drive is best utilized in a log-structured manner due to the intrinsic properties of flash [6]. Further to that, in order to harness the aggregate bandwidth of a cluster of flash units, we propose to organize the entire cluster as a single shared log. CORFU[1] is a new storage system for flash clusters which demonstrates the feasibility of this approach. The key idea in CORFU is to expose a cluster of network-attached flash devices as a single, shared log to clients running within the data center. Applications running on the clients can append data to this log or read entries from its middle.

Internally, the CORFU shared log is implemented as a distributed log spread over the flash cluster. Each log entry is projected onto a fixed set of flash pages (see Figure 1). The cluster as a whole is balanced for parallel I/O and even wear by projecting different entries onto distinct page sets, rotating across the cluster. In this design, CORFU completely adopts the vision behind log-structured storage systems like Zebra [12], which balance update load in a workload-oblivious manner. The difference is that the CORFU log is global and is shared by all clients. Flash makes this design plausible, because it facilitates random reads from the middle of the log with no 'seek' penalty.

CORFU is not a 'from scratch' invention; it builds on a vast knowledge in reliability methods. Yet we found that applying those directly would not yield the high-throughput store that we aimed for. We drew foundational lessons from essentially every step of the CORFU construction. Whereas the full design of CORFU is given elsewhere [3], the rest of this paper aims to highlight the principled innovation in CORFU over previous clustered storage systems.

[1]CORFU stands for Clusters of Raw Flash Units, and also for an island near Paxos in Greece.

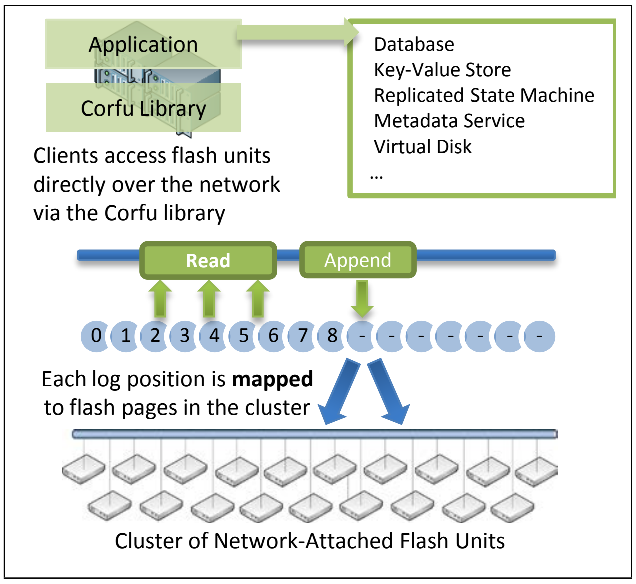
Application
Corfu Library
Database
Key-Value Store
Replicated State Machine
Metadata Service
Virtual Disk
...
Clients access flash units directly over the network via the Corfu library
Read
Append
0 1 2 3 4 5 6 7 8 - - - - - - -
Each log position is mapped to flash pages in the cluster
Cluster of Network-Attached Flash Units

Figure 1: CORFU architecture

Going beyond Paxos

In this section, we briefly overview several prevailing approaches and explain their limitations in our settings, thus motivating the need for a new scheme.

The State Machine Replication (SMR) approach [16] builds a reliable service out of failure-prone components. The approach works as follows. We build a single-machine service as a state machine whose transitions are deterministic. We instantiate multiple replicas of the service which all start with the same initial state, and deliver all state-manipulating commands to them in a unique sequence. Replicas process commands in the same order and since all replicas execute all commands we can lose all but one without compromising the service state.

The core of SMR is realized by implementing a total-ordering

(TO) engine which has two roles: One, it forms agreement on a sequence of commands to be delivered to replicas. Two, it stores information persistently about the commands and their order until they are applied to replicas with sufficient redundancy against possible failures. Once a certain prefix of commands has been processed by all replicas – and at least $F+1$ are needed for F-tolerance – they may be evicted from the TO's store for garbage collection purposes.

With TO, building a replicated service is a breeze: Each replica can be oblivious to being replicated and acts autonomously on its local state. Hence, much attention has been put into the consensus algorithm which processes command proposals by the clients in a stream and produces a sequence of agreement decisions as output. Indeed, the renowned Paxos framework [14] and many group communication works [8] address this challenge. These frameworks use $2F+1$ participants which are necessary to solve consensus with up to F failures.

The naive way to employ a consensus-based framework is to deploy $2F+1$ machines (*acceptors*) as a *TO-cluster*, and another set of $F+1$ machines (*learners*) as service replicas. In the context of a flash cluster, this would work as follows. We would have clients send *store* requests to the TO-cluster of $2F+1$ machines. The TO engine would process requests, form a total order on store commands and keep them in persistent storage, and output an ordered sequence of store-commands to $F+1$ replicated flash units.

Since in this setting, the store commands contain the stored data itself, we waste both network and storage resources in funneling commands through a separate TO cluster; can we use the $F+1$ flash units to also act as the acceptors in the TO-cluster? This requires additional effort, first, because we deploy only $F{+}1$ units for F-tolerance; second, because storage units should not need to initiate communication among themselves as Paxos acceptors would. Previously, agreement in a model where participants are passive storage entities has been addressed in Disk Paxos [10, 7]. Unfortunately, the approach taken in these works is to let clients contend for the contents of each entry. This unnecessarily consumes storage space which is reserved for clients to propose their inputs, as well as network bandwidth on lost attempts. These solutions also deploy $2F+1$ storage replicas, which is rather expensive.

Another limitation of previous approaches is load distribution. In order to allow concurrent client updates and reads to/from the cluster, and also to address load distribution, existing cluster stores partition data (according to some attribute). Several large-scale web services build off of such a partitioned infrastructure, e.g., Amazon's Dynamo [9], Facebook's Cassandra [13], and others. Unfortunately, with data partitioning, we lose cross-partition consistency. Moreover, we also introduce load imbalance against dynamic and spiked loads.

Summary of CORFU Design

The CORFU design differs from the above in profound ways. As noted in the introduction, it is beyond the scope of this paper to provide a full description of the CORFU design, which is described in detail elsewhere [3]. Here, we briefly outline the design in order to point out the salient departures from conventional methods.

The task of implementing a shared log over a cluster of flash units involves three distinct pieces of functionality:

1. Mapping: Each position in the shared log is mapped to a replica set of physical flash pages residing on different flash units. To read or write a log position, a client first maps it to its corresponding set of flash pages. In CORFU, this mapping – called a projection – is stored compactly at the clients to allow for fast lookups, as a deterministic function over a membership view of the flash cluster.

2. Tail-finding: To append data to the log, a client has to determine the current tail of the log. In CORFU, this is done via a dedicated sequencer node – essentially, a counter that can be accessed over the network – which allows clients to determine and reserve the next available position in the log. Importantly, the sequencer is just an optimization for finding the tail quickly and avoiding append contention with other clients through reservations.

3. Replication: When a client determines the set of flash pages mapped to a particular log position, it then uses a replication protocol to read or write data at those pages. By default, CORFU uses client-driven chain replication; the client writes to each of the flash pages in some deterministic order, waiting for each flash unit to acknowledge the write before proceeding to the next one. Reads are directed to the end of the chain, but can also be served by replicas in the middle of the chain if the write is known to be completed.

Accordingly, to read data at a particular log position, the client maps the position to the replica set of flash pages, and then uses the replication protocol to read from this set. To append data to the log, the client first finds and reserves the tail position of the log using the sequencer, maps it to the replica set of flash pages, and then writes data to them using the replication protocol.

The projection at each client maps log positions to flash pages by dividing the cluster of flash units into replica sets and then going round-robin over these sets. For example, if the cluster contains six flash units, F1 to F6, the projection organizes these into three replica sets of two units each (F1 and F2 forming a replica set, for instance). It then maps log positions in a circular manner onto pairs: position 1 to flash page F1:page-1 and F2:page-1; position 2 to F3:page-1 and F4:page-1, and so on, wrapping around by mapping 4 to F1:page-2 and F2:page-2. In this manner, CORFU implements a shared log design over a cluster of flash units, ensuring that reads to the log are completely parallelized while appends are limited only by the speed at which the sequencer can hand out tokens. Figure 2 depicts this mapping.

When a flash drive fails in the cluster, or when a flash unit fills up, clients in the system use an auxiliary to uniquely carve a new segment of the infinite sequence and project it onto a new set of drives. The auxiliary is only involved in these relatively low-frequency reconfigurations. For more common failures, we bypass the auxiliary: Once the sequencer allocates a particular offset to a client, there must be a way to claim back the 'hole' which a failed client would leave behind. We make use of our chain replication protocol to complete a partially filled offset efficiently, thus minimizing the disruption to the stream of log updates.

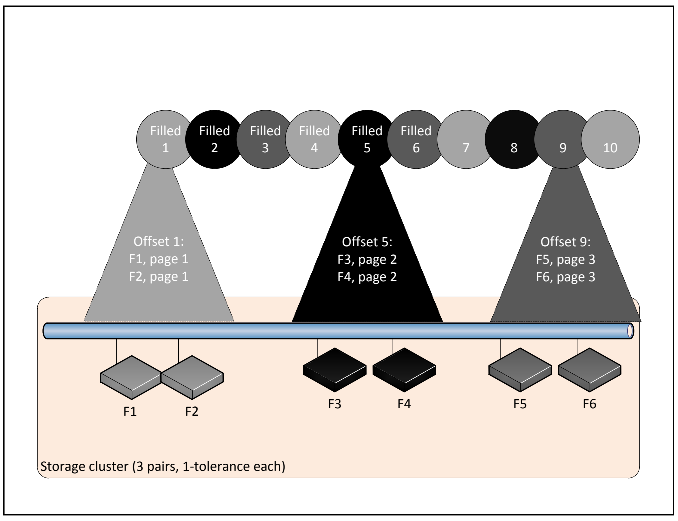
Filled 1
Filled 2
Filled 3
Filled 4
Filled 5
Filled 6
7
8
9
10
Offset 1: F1, page 1 F2, page 1
Offset 5: F3, page 2 F4, page 2
Offset 9: F5, page 3 F6, page 3
F1
F2
F3
F4
F5
F6
Storage cluster (3 pairs, 1-tolerance each)

Figure 2: CORFU projection

How far is CORFU from Paxos?

There are several foundational lessons drawn from building CORFU. Obviously, forming a globally-ordered sequence of updates constitutes a classical manifestation of state-machine-replication (SMR). But in order to build CORFU effectively, we needed to weave multiple methods and also to introduce novel mechanisms. The resulting design deviates from existing replicated systems with respect to each component highlighted in the previous section.

With respect to **mapping**, we employ an auxiliary in order to splice together individual log-segments into one global sequence. The use of an auxiliary to manage dynamic configurations of flash drives is explained in a recent tutorial [5] and has roots in the Vertical Paxos scheme [15], adapted to our passive-server settings. Internally, each entry in a segment is replicated for robustness on a set of pages using a **chained replication** protocol which borrows from Chain Replication [17]. The **sequencer** is quite a unique role. It is effectively a network counter and serves as a performance enhancer, arbitrating between clients which attempt to append to the tail of the log concurrently. However, it has no bearing on consistency, hence it does not need to keep any persistent information, and is not constrained by storage throughput.

This design differs from standard SMR in two crucial facets:

Time-slicing: Classical cluster replication based on SMR puts the responsibility over every update on the entire cluster, which caps throughput at roughly twice the I/O capacity of individual servers. The only way around this limitation is through partitioning. Traditionally, we partition a replicated storage system by some attribute of the data objects, essentially splitting the stream of updates into multiple independent streams. We lose atomicity and load balancing across partitions in this manner. In contrast, CORFU partitions the responsibility across log-offsets. That is, we map each offset in the global log onto a separate set of flash drives, thus allowing parallel IO across the cluster. The advantage is that we obtain cluster-wide consistency guarantees and, at the same time, global load balancing. Though in itself, this idea is not new, we are aware of no previous system which employs this scheme.

Decoupling sequencing from I/O: Most SMR schemes use one of the replicas as a leader/primary that injects a pipeline of proposals, with the unfortunate result that this bounds the pipeline throughput at the I/O capacity of the leader. In contrast, we separate the control role of the sequencer from that of the storage replicas, and as a result, SMR throughput is bounded only by the speed at which 64-bit tokens can be handed out to clients.

Other departures from conventional Paxos stem from CORFU's unique hardware constraints. CORFU is designed to operate over network-attached flash units: simple, inexpensive and low-powered controllers that provide networked access to raw flash. Accordingly, storage devices are not allowed to initiate communication, be aware of each other, or participate actively in fault-tolerance protocols. The result is a client-centric protocol stack, where most of the functionality for ensuring the consistency of the shared log abstraction resides at the clients.

Another key implementation difference results from the more powerful interface offered by a shared log. In conventional SMR, clients can only receive the next command in the sequence; learning about older commands is typically implemented via state transfer between clients. In contrast, the shared log allows clients to query the contents of older slots in the sequence for perpetuity (or until the application calls a garbage-collection primitive). To implement this, CORFU carves the log address space into disjoint ranges, each of which can be handled by a different cluster of storage devices. When the tail of the log moves past a particular range, the storage devices managing that range will stop receiving writes but continue to serve reads.

An additional desirable feature is that our replica sets consist of $F + 1$ flash units for F-tolerance. The reason we can form consensus decisions with fewer than $2F + 1$ replicas is that we use an auxiliary for handling failures. Our adaptation of the auxiliary-based reconfiguration approach to client-centric settings is novel, as is our common-failure hole-filling procedure.

CORFU as a Distributed SSD

Thus far, we have described CORFU as a shared log abstraction. Alternatively, CORFU can be viewed as a networked, distributed SSD. A conventional SSD implements a linear address space over a collection of flash chips. Such SSDs typically use a log-structured design under the hood to provide good performance and even wear-out in the face of arbitrary workloads.

In this context, CORFU plays a similar role over a distributed cluster of flash chips; it uses a log-structured design at distributed scale to provide good performance and controlled wear-out to a cluster of flash. In contrast to conventional SSDs, CORFU exposes the log directly to applications instead of a linear address space. However, such an address space can be layered over the shared log, allowing CORFU to act as a conventional block drive with a fixed-size address space.

Importantly, the CORFU logging design allows for distributed wear-leveling. Current designs for flash clusters partition data according to some key across storage devices. As a result, devices can wear out at different rates, depending on the write workload experienced by individual keys. A cluster with non-uniform wear-out can result in unpredictable performance and reliability; for example, in a database, certain tables may become slower or less reliable than others over time. In contrast, CORFU enables distributed wear-leveling by implementing a single log across devices, ensuring that all devices see even wear-out even for highly skewed write workloads.

CORFU's architecture borrows heavily from the FAWN system [1], which first argued for clusters of low-power flash units. While FAWN showed that such clusters could efficiently support parallelizable workloads, the shared log abstraction we implemented extends the benefit of such an architecture to strongly consistent applications. The other benefit is described above, namely, aggregate wear leveling and load balance across the entire cluster.

Applications

Another key deviation of CORFU from SMR is that the log is the store itself, rather than a transient queue of updates. That is, whereas in SMR we think of the total-ordering engine as transient and the service state as permanent, in CORFU, we envision the log as storing information persistently, whereas everything else can be maintained as soft state.

A flash-based shared log enables a new class of high throughput transactional applications. The key vision here is to persist everything onto the global log, and maintain metadata in-memory for fast access and manipulation. We describe in [3] a key-value store built on top of CORFU that supports a number of properties that are difficult to achieve on partitioned stores, including atomic multi-key puts and gets, distributed snapshots, geo-distribution and distributed rollback/replay. A map-service (which can be replicated) maintains a mapping from keys to shared log offsets; below, we further elaborate on that.

Another instance of this class, Hyder [4], is a recently proposed high-performance database designed around a flash-based shared log, where servers speculatively execute transactions by appending them to the shared log and then use the log to decide commit/abort status. The Hyder paper included a design outline of a flash-based shared log that was simulated but not implemented. In fact, the design requirements for CORFU emerged from discussions with the Hyder authors on adding fault-tolerance and scalability to their proposal [2], and the full Hyder system is currently being implemented over our code base.

A shared log can also be used as a consensus engine, providing functionality identical to consensus protocols such as Paxos (a fact hinted at by the name of our system). Used in this manner, CORFU provides a fast, fault-tolerant service for imposing and durably storing a total order on events in a distributed system. From this perspective, CORFU can be used as a Paxos implementation, leveraging flash storage to provide better performance that conventional disk-based implementations.

Building a Name-Space

Using a shared log as a the store creates a need to locate information in the log. That is, if one client updates a data object D by appending an update-entry to the log at position p, and later, another client queries the value of D, the second client should find log entry p. Required is a service which finds data in the log quickly. This service is itself a CORFU client. It reads the log in order to maintain a directory of locations of data in the log. For atomicity, the service needs to point to the most recent state of each data item.

A centralized name server which provides atomicity guarantees is relatively easy to build, because all requests are serialized through it. The order in which the centralized map-server handles updates/queries becomes the abstract linearization which conforms with the clients view of the execution. In particular, note that a client query returns the last updated entry in this linearization.

A centralized design does not present a single point of failure, because on failure, a new server is easily reconstructible from the log. Indeed, our entire vision is that a shared log enables this type of fast, in-memory applications which are free from the need to persist any information. However, a centralized design could be a performance bottleneck, and hence we discuss a distributed name space next.

A distributed name service allows clients to access distinct name-servers concurrently, with the goal of enhancing overall throughput. One potential design puts a name-server on each client machine, as a component of the CORFU client-side library; another design deploys a service-layer of dedicated name servers, which are spread over the network.

An obvious way to distribute the name service is via partitioning, but this has the same limitation as partitioning the log itself: no cross-partition consistency or load-sharing. To drive multiple name servers without such partitioning, each individual name server needs to guarantee freshness by playing the log forward. This is challenging for two reasons. The name server needs to somehow determine the current tail of the log. Second, it must then wait until it can read every offset until the end of the tail, and it might be delayed due to holes. We are currently investigating complementary ways to achieve freshness which alleviates some of this stress. Alternatives include a combination of techniques involving gossiping among clients about log updates; querying the sequencer for the tail of the log and doing a bulk-read; and many others.

Summary

CORFU organizes a cluster of flash devices as a single, shared log that can be accessed concurrently by multiple clients over the network. The CORFU shared log makes it easy to build distributed applications that require strong consistency at high speeds, such as databases, transactional key-value stores, replicated state machines, and metadata services. A key design point is to use time-slicing in order to distribute load and parallelize I/O across a cluster of flash units. This exploits flash storage to alter the trade-off between performance and consistency, supporting applications such as fully replicated databases at wire speed.

1. REFERENCES

[1] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: a fast array of wimpy nodes. In *SOSP 2009*.

[2] M. Balakrishnan, P. Bernstein, D. Malkhi, V. Prabhakaran, and C. Reid. Brief announcement: Flash-log – a high throughput log. In *24th International Symposium on Distributed Computing (DISC 2010)*, September 2010.

[3] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. Davis. Corfu: A shared log design for flash clusters. Technical Report MSR-TR-2011-119, Microsoft, September 2011.

[4] P. Bernstein, C. Reid, and S. Das. Hyder Ű a transactional record manager for shared flash. In *CIDR 2011*, pages 9–20, 2011.

[5] K. Birman, D. Malkhi, and R. V. Renesse. Virtually synchronous methodology for dynamic service replication. Technical Report MSR-TR-2010-151, Microsoft, November 2010.

[6] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *Operating Systems Review*, 41(2):88Ű93, 2007.

[7] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 78–87, New York, NY, USA, 2002. ACM.

[8] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33, December 2001.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP'07)*, 2007.

[10] E. Gafni and L. Lamport. Disk paxos. In *DISC '00: Proceedings of the 14th International Conference on Distributed Computing*, pages 330–344, London, UK, 2000. Springer-Verlag.

[11] J. Grey. Tape is dead, disk is tape, flash is disk, ram locality is king. Storage Guru Gong Show, Redmond, WA, 2006.

[12] J. Hartman and J. Ousterhout. The zebra striped network file system. *ACM TOCS*, 13(3):274–310, 1995.

[13] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[14] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[15] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *PODC 2009*, pages 312–313. ACM, 2009.

[16] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[17] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI 2004*.