Going Beyond Paxos

Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran and Ted Wobber Microsoft Research, Silicon Valley

July 29, 2011

1 Introduction

Flash storage is making inroads into data centers, enabling a new class of applications that require persistence as well as extreme performance. Clusters of flash can satisfy millions of IOs per second at submillisecond latencies while consuming significantly less power per IO than disk clusters. Unfortunately, current designs for scalable storage systems are predicated on the properties of hard disks, and can be inefficient or unreliable when used with flash clusters. For example, replacing hard disks in high-end servers with expensive PCI-E flash drives (e.g., Fusion-io [10]) can result in a rigid architecture which is hard to scale incrementally and may be susceptible to network bottlenecks. Furthermore, scaling techniques such as data partitioning (e.g., FAWN [2]) can age drives at difference rates, resulting in uneven reliability and performance across the cluster. The upshot is that new abstractions are required to fully realize the potential of flash clusters.

Accordingly, Corfu¹ is a new storage system for flash clusters. 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, this shared log is implemented as a distributed log spread over the flash cluster. This design makes sense for two reasons:

• Corfu is a distributed SSD...

Corfu runs over raw flash chips directly attached to the network, obviating the need for storage servers, and for commercial SSD controllers in the storage cluster; this slashes infrastructure cost and power consumption by an order of magnitude. In effect, Corfu acts as a distributed SSD, implementing flash management and wear-leveling at cluster scale.

• ... with a shared log interface.

From a top-down perspective, the Corfu shared log is a powerful primitive for building applications that require strong consistency, such as databases, transactional key-value stores and metadata services.

Corfu is implemented primarily as a client-side library, with two components on the server side: A set of flash devices and a *token-server* (essentially, a network counter). To append data to the shared log, a client first obtains a token from the token-server indicating the next free *offset* in the shared log. The client then uses a *configuration* – basically, a membership view of the storage cluster – to deterministically map this

¹Corfu is an island just off the Paxi group of Greek islands which contains the famous island Paxos.

token to a replica set of physical flash pages in the cluster. The client writes its data directly to these flash pages across the network. To read the entry at a specific offset in the shared log, the client similarly uses the configuration to map the offset to a set of physical flash pages, and then reads the data directly from one of these pages. When a flash drive fails in the cluster, clients in the system use a reconfiguration protocol to transition to a new configuration. Figure 1 depicts the Corfu architecture at a high level.

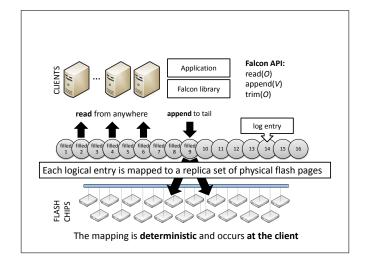


Figure 1: Corfu architecture

This design deviates from conventional implementations of clustered storage in several important ways:

- In traditional shared cluster designs, the entire cluster is responsible for every update, which caps throughput at roughly twice the IO capacity bound of individual servers. The only way around this limitation is through partitioning. However, whereas existing solutions partition the responsibility by objects, we partition 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 reason we can do it is that we use a separate sequencer. The advantage is that we obtain cluster-wide consistency guarantees and global load balancing. Though in itself, this idea is not new, we are aware of no previous system which employs this scheme.
- One of the main challenges with our sequencer-based control is that 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. The foundational approach to solving this would be to induce a configuration change. Rather than doing that, we introduce a simple novel hole-filling procedure which deals with holes efficiently.
- Having a single global log allows us to support higher semantics such as multi-object atomicity and transactions.
- Each time we fill up the space in a set of flash drives, we shift to a new configuration. However, we continue to maintain old configurations in order to access earlier data in the log. Thus, our reconfiguration mechanism actively handles a list of configurations which map to a sequence of contiguous segments of the log. Accordingly, the read load of the shared log can be distributed over different sets of flash drives.

• In order to allow us to operate directly over passive storage devices, we need to adapt all protocols to an appropriate 'data centric' model. The challenge in this model is that storage devices are not allowed to communicate with each other. Our solutions empower clients with most of the responsibility while employing thin servers.

The current Corfu implementation has been deployed over a cluster of 32 Intel X25M server-attached SSDs. This deployment currently supports 400K 4KB reads/sec and 200K 4KB appends/sec. Several applications have been prototyped over Corfu, include a transactional key-value store and a fully replicated database. While we are still evaluating these applications, the initial results are promising; for instance, our key-value store can support atomic multi-gets and multi-puts involving ten 4KB keys each at speeds of 40K/sec and 20K/sec, respectively.

2 SMR Primer and the Throughput Dilemma

The State Machine Replication (SMR) approach [17] 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 persists information about the commands and their order against possible failures. With TO, building a replicated service is a breeze: Each replica can be oblivious to being replicated and acts autonomously on its local state. Once a certain prefix of commands has been processed by at least F + 1 replicas (for *F*-tolerance), they may be evicted from the TO's store for garbage collection purposes.

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 renown Paxos framework [14] and many group communication works [12] 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 as a *TO-cluster* and another set of F + 1 machines as service replicas. In the context of the storage cluster problem, 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 and persist this information. It would output an ordered sequence of store-commands to F + 1 storage replicas. Figure 2(left) depicts this design. However, this naive realization is somewhat wasteful, as we proceed to explain below.

First, we deploy 2F + 1 machines in order to achieve an *F*-tolerant reliable store. Not only is this hardware extraneous to the storage cluster itself, it also needs to persist the history of commands (at least until a checkpoint). In our setting, commands to store data **are** the data. So data is stored twice, once as proposed commands in the TO-cluster, and again in the storage cluster itself. Moreover, the payload in store commands needs to travel more times than necessary across the network: From the client to the TO-cluster, and again from the TO-cluster to the back-end storage machines. Our goal of achieving high efficiency and low cost would not be served well with this design.

Part of the overhead is artificial, and is conventionally alleviated by co-locating the service-replicas with the machines of the TO-cluster. This style of replication has been explored extensively in the group-communication arena [12]. With this approach, we architect a storage cluster by sending store requests to

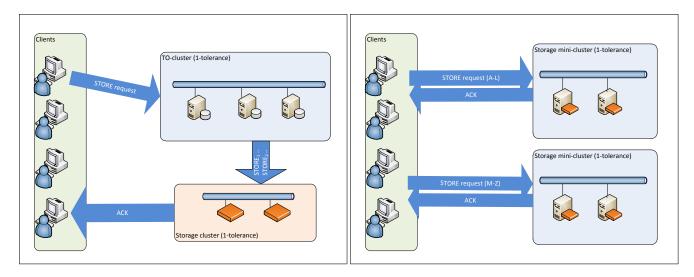


Figure 2: SMR : Vanilla Deployment (left) and Partitioned Deployment with Group Communication (right)

participating machines, who process them for ordering. As soon it learns an ordering decision for a store command, a participating machine already has the data stored locally, and it can immediately apply it to its local replica state.

The most common ordering protocol is primary-driven, e.g., Chain [18], which clearly caps throughput at the IO capacity of the primary. Other protocols are majorities-based, e.g., SRS [6], or use a revolving-token regime, e.g., Mencius [16]. These may achieve twice the throughput of any individual server, because the busiest replica out of any majority must suffer load of at least half of the global sequence of decisions. This is still far from realizing the aggregate throughput-potential of a sizable cluster.

The only way around the capacity bottleneck of individual servers is via partitioning, i.e., by setting up autonomous replica-sets which work concurrently. The traditional partitioning strategy divides the responsibilities on service data among distinct partitions (see Figure 2(right)). Several large-scale web services build off of such a partitioned infrastructure, e.g., Amazon's Dynamo [8], Facebook's Cassandra [13], and others. Unfortunately, with data partitioning, we lose cross-object consistency. Moreover, we also introduce load imbalance against dynamic and spiked loads.

3 The Corfu Solution

We take a radical paradigm shift from all of the above SMR frameworks in order to facilitate parallelizable IO at wire-speed without breaking the data into partitions. Our design contains new concepts throughout the protocol stack. Although our design is tailored to flash drives, from here on we refer to generic *storage devices* since the Corfu design may be carried onto other storage technologies.

Partitioning over Log-offsets

Our log is arranged over a set of storage devices by mapping each *log-offset* deterministically onto a *mini-set* of F + 1 devices and corresponding physical locations. Read and write operations on different log-offsets can work concurrently by going to different mini-sets. For example, say that we have 6 storage devices arranged in three mini-sets of two, each resilient to one failure (i.e., F = 1). We can map log offsets in

groups of 4 to pairs of devices, going around the system in a round-robin manner as follows: Offsets 1 thru 4 go to the first four pages of device 1 and mirrored on 2; offsets 5 thru 8 are mapped onto the first four pages of 3 and 4; offsets 9 - 12 are mapped onto the first four pages of 5 and 6; offsets 13 - 16 wrap around back onto devices 1 and 2, occupying pages 5 - 8 on each device; and so on. Figure 3 depicts the mapping of log offsets onto devices.

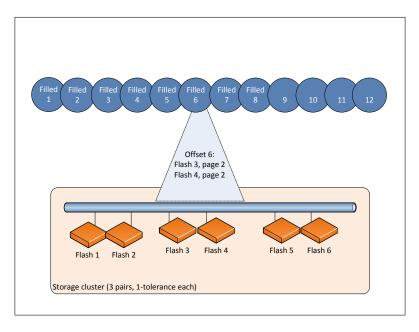


Figure 3: Corfu uses a fixed mapping of log offsets onto drives

Having done this, we are partitioning the storage cluster over **log offsets**, not over data, and this point deserves further attention. Consider an artificially skewed workload where clients update the same single data-object over and over. In a normal partitioning scheme, this would result in placing 100% load on the single partition which contains the object. In Corfu, we would be appending update entries to the log one after another, each one mapping onto a different mini-set of storage devices. Despite the extremely unbalanced load, we achieve perfect load balance on storage devices, even wear, and completely parallelizable reads and writes over partitions.

The crucial point is that we derive consistency from the sequenced structure of the log, and not by sequencing operations through some primary.

Decoupling Sequencing from IO

So far, we designated a fixed mini-set of storage devices for each individual log entry, but we did not escape the need to form agreement on its contents. Previously, agreement in a model where participants are passive storage entities has been addressed in Disk Paxos [11] and certain follow up works [7, 1]. We do not adopt the approach taken in any of these works because it lets 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.

In Corfu, a *log-append* request employs a centralized *token-server* which allocates increasing offsets in the log. Thus, the token-server removes contention by providing offset reservations on behalf of clients. In

normal scenarios, only one client will attempt to write data to any particular offset. Later, we discuss the case of a client failure, where we will need to revoke the client's exclusive use of an offset (see Section 3.1).

Log-append works as follows (see Figure 4):

- 1. A client contacts the token-server with a reservation request.
- 2. The token-server replies with the current token value and atomically increments it.
- 3. The client maps the reserved offset to physical locations and stores data directly onto storage devices. Note that only one client may hold a reservation for a particular offset, hence there is no possible contention or conflict on the physical locations it is mapped to.
- 4. The client waits for responses from all storage devices to complete the operation.

An append to an offset has committed when acknowledgments are received from all replicas in the mini-set.

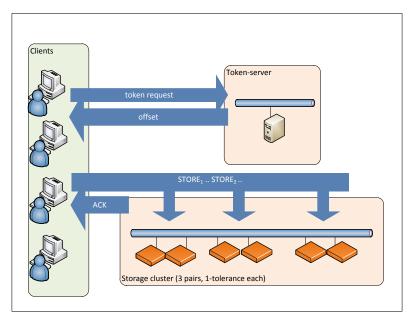


Figure 4: Corfu

A read of a committed offset simply maps the offset to physical location and reads data directly from any of the replicas in a mini-set.² Since different offsets map to different mini-sets of devices, reading is essentially scale-free and can stream from as many storage devices as desired. The IOs associated with appends are also completely parallelizable; however, they are bottlenecked at the token-server. In theory, we simply shifted the bottleneck from a participant acting as primary to a token-server.

In practice, we made a huge difference. Tokens can be served by a simple network counter, which may be realized on a standard PC at a few hundred Ks per second, and up to a million per second on dedicated hardware. With this simple, yet radical change, we have achieved in an experimental cluster a throughput

 $^{^{2}}$ Reads of non-committed offsets are not allowed. We provide API for probing the system to learn if an offset has committed, see Section 4 below.

of roughly half a million IOPS for reads or writes. More specifically, our cluster has 16 machines, each equipped with dual flash drives and a 1 Gbit/sec NIC. The total number of local 4-KByte IOPS driven by the dual drives is roughly 40K, but each machine is capped at serving over the network about 30K remote IOPS due to the NIC capacity. This gives an aggregate throughput of slightly over half a million IOPS. We further arranged data pages to be placed on two mirrored servers each; append throughput is therefore about 250K per second, which our current user-level token-server process roughly sustains.

In total, with a population of 40 application machines performing an equal mix of reads and appends, we are able to saturate the theoretical limit of half a million IOPS. This is achieved with hardware costs of 16 PCs reinforced with \$3,000 worth of flash drives (!).

We emphasize that the cluster provides a globally shared reliable and consistent log, i.e., a single SMR image at half a million IOPS throughput. The crucial point is that the consistent ordering is derived from the tokens, and not through sequential access at any primary. In this way, although IO is partitioned and highly parallelized, the cluster exposes a single totally-ordered log image.

3.1 Failure Handling

Corfu guarantees progress in face of various failure scenarios, including client failure, token-server failure, and up to F simultaneous storage-device crashes. There are different mechanisms for handling these, with increasing level of complexity. Figure 6 contains a succinct pseudo-code description of these protocols.

Client Failures and Holes. The token-server might appear simply as a Paxos leader in disguise, but this is not the case. The token-server is fundamentally different from a leader in Paxos. In particular, a Paxos leader needs to perform a startup protocol in order to determine which offsets might have been filled by previous leaders. A Paxos leader must reinforce partially completed proposals by previous leaders, and propose its own commands only in empty commands offsets. A token-server does none of that. It issues tokens in sequence and thereby relinquishes control on the corresponding offsets for good. Clients are responsible for filling the allocated offsets with data. Consequently, when a client fails (or delays) in filling an allocated offset with data, we are left with a *hole* in the log. This is undesirable, especially since higher level applications may need to play the log in sequence order to derive consistency. DeFago observes in [9] that because of the difficulty in handling this problem, employing a token-server for sequencing has rarely been tried in practical systems. Whereas we could treat client failures as a first-class failure and tackle them with reconfiguration, this would be costly. In our experience, clients hang or slow-down more frequently than servers, hence it is desirable to recover from client delays quickly and aggressively.

To this end, we enhanced the basic replication protocol we described above with mechanisms that borrow from Chain Replication (CR) [18], while adapting CR to our data-centric settings. Specifically, we extend each individual storage device to support *write-once* semantics, which means that once data is written to any physical location, it cannot be overwritten. We also modify the client append protocol to store data on the designated replicas of a mini-set one by one in a fixed order. We must wait for confirmation on each replica before we proceed with the next. If the write to the first replica fails, it means that some other content was already written there. We then mirror that content onto the remaining replicas, again, one by one. An append to an offset has committed when acknowledgement is received from the last replica in the mini-set.

Filling a hole in an offset is done by writing a special '*junk*' mark to that offset using the chained replication protocol above. The protocol guards against a race between the original client which holds a token to this offset and the client(s) with the *junk* mark: the fate of the entry will be set by whoever gets first to the first replica of the mini-set

The basic property we get is that we never allow different data values to be written on distinct replicas in the mini-set. This invariant is crucial so long as we want to support F failures with F + 1 replicas, since we could return one data value and then have all replicas holding that data value fail, in which case a subsequent read will return the different data value, regardless of the protocol used.

It is worth mentioning what happens in face of reconfiguration of the mini-set. There are two cases to consider. The first is that a value has been committed to all replicas in the mini-set before the reconfiguration. In this case, reconfiguration will not violate the commitment because any surviving subset of the mini-set will contain this value. The second case is that a value has been written only to some replicas. Then, this value may disappear due to reconfiguration, and the hole filling procedure may write 'junk' to the surviving mini-set. This does not violate the committed-write uniqueness, because the value has never been committed, and a fortiori, never been read. Although there are now potentially replicas storing the wrong, old value, our reconfiguration mechanism below prevents (through epoch-tagging) any client from reading the replicas which were removed from the mini-set.

It is also worth noting that with chained replication as described above, we could allow in principle any number of conflicting attempts to fill the same offset, and let the first replica determine a winner. However, we don't allow this in normal scenarios, and rather, we let the token-server remove contention in order to achieve high throughput. Nonetheless, when a hole blocks progress, we initiate the hole-filling procedure which fills the hole with a special *junk* mark. In this case, we indeed utilize the write-once semantics at the first replica to resolve the fate of the entry.

Token-server failure.

The token-server is a mechanism for performance enhancement, which removes contention between clients on the contents of log-offsets. Because its role has no bearing on consistency, we do not need to persist the token-server state. Moreover, in case of a token-server crash, the system may continue operating (at reduced speed) without one for a transient recovery period. A new token-server may be started with a conservative hint as to which index the failed server reached; any contention on slots which were assigned to multiple clients is resolved by our chained replication mechanism.

Reconfiguration.

Reconfiguration is triggered whenever a failure in any of the storage devices occurs, or when they fill up and we need to deploy new hardware. Here, too, we had to deviate from standard Paxos reconfiguration. Paxos requires a consensus decision on reconfiguration among the surviving members of the current configuration. Our choice of deploying only F + 1 replicas precludes that.

Our protocol follows recent guidelines on virtually synchronous reconfiguration [5]:

- We first *seal* the storage devices from accepting further writes in the current configuration. Each storage device in the system maintains an *epoch* counter; we seal a device by incrementing this counter and subsequently, the device will refuse requests (for read or for write) carrying a lower epoch value. We only need to seal the devices in the mini-set which undergoes a change, not the entire system. This necessitates maintaining a per mini-set *epoch* counter. However, for simplicity of the exposition, from here on we omit the details of mini-set epoch management, and refer to one global epoch in the entire system.
- 2. We then take a stable snapshot of the state of the log and determine which range of offsets are ever filled by the current configuration.

3. We employ an *auxiliary configuration manager* to approve and store a description of the offset range of the current configuration, and of any devices removed from it. The auxiliary also stores a unique next configuration, which becomes activated as soon as the transition is stored on the auxiliary.

The use of an auxiliary is justified both by our choice of deploying only F + 1 replicas (see [15]) and by our desire to keep storage-devices' functionality at minimum complexity. With our present design, the only configuration-related state maintained by a storage device is the *epoch* counter. The auxiliary is not in the critical path of high-throughput, hence it can easily sustain the load of multiple replication systems. Moreover, the auxiliary itself can be conventionally replicated for high availability using SMR, and acts as a reliable point of metadata maintenance in the system.

Unlike normal SMR, we do not abandon previous configurations, nor transfer their state: The data stored in any configuration is part of a logically infinite log, and we continue maintaining old configurations in order to provide read-access to parts of the log stored by them. (Garbage collection eventually occurs, but is not described here for brevity.) Hence, our auxiliary maintains a view of the log which maps contiguous parts onto a list of dynamic configurations. Each link in the list is subject to change as servers are taken off and recovery servers are deployed back. Only the last configuration is active for appends; the ones preceding it support only reads.

Briefly, the reconfiguration protocol works as follows. Denote the current epoch number by e.

- 1. Send all storage devices in the current configuration a seal(e) request. Each device increments its epoch to e + 1 and rejects future requests with any lower epoch value. It responds with an ACK(k) which carries the highest occupied offset k it stores.
- 2. Collect all ACK responses, and compute the highest written offset k_m .
- 3. Send the auxiliary a reconfiguration request $reconfigure(e, k_m, current, next)$, where current contains any changes to the membership of the current configuration, such as the removal of crashed devices. next contains the membership of the new configuration. The auxiliary accepts and acknowledges the first reconfiguration request on e and stores it; it rejects further requests with the same epoch e.
- 4. Wait for confirmation from the auxiliary.
- 5. Any client which learns about the configuration change from the auxiliary can initiate appends at the new configuration, starting at offset $k_m + 1$.
- 6. Meanwhile, any client which learns about the modified configuration responsible for offset-range preceding k_m may read from it, and it may trigger hole-filling in the modified configuration. Both activities must use epoch number e + 1.

4 Corfu Protocols

Each client is born with initial knowledge of the auxiliary configuration manager, and from it, it learns a mapping DSM() (for Distributed Storage Manager) which maps offsets to a set of storage-servers and corresponding physical locations on them. Internally, the auxiliary maintains a projection of contiguous offset ranges onto membership sets of storage servers, and within each membership set, it has information on storing offsets cyclically onto mini-sets. The mapping dynamically changes over time due to reconfigurations,

hence we tag each view with an *epoch* number. A client's knowledge might be outdated if it did not consult with the auxiliary about the latest epoch change. Should this happen, the client will get a rejection response from the system, which will trigger *reconfig* at the client to catch up. This guarantees that there is a match between where clients expect to find data and where they are permitted to access it (directly).

Below, our protocols are described succinctly in two frames. Figure 5 gives a pseudo-code description of the steady-state *logappend* and *logread* operations. In addition, it contains a *fillhole* utility, which fills junk into a log-entry if it blocks progress by remaining empty for too long.

Figure 6 elaborates the recovery protocols from token-server failure and flash-server failures. A failure in any of the storage servers is handled by reconfiguration. When any client in the system detects a token-server failure, it simply starts a new token-server. Two token-servers which respond with the same offset may cause contention, resulting in some inefficiency, but they do not induce any inconsistency. The start-up code executed by a new token-server, which strives to reduce this contention as much as possible, is provided below.

Local variables: curepoch: the client's current epoch; carried implicitly as $enum$ in all server-bound messages sealed[1curepoch]: indicates if flash-server is sealed, in a given epoch CF[1curepoch]: an array of (configuration, state) tuples $\ell\ell$: the next token value to be issued by the tokenserver function DSM (offset): return set of flash-servers and corresponding physical offsets on each		
$ \langle q \rangle$ request at token-server: resp, $\ell \ell \rangle$ back to client $ \langle o, u \rangle$ request at flash-server: um] aled \rangle rejection and return led itten, $content \rangle$ response and return ysical offset o map of filled offsets esp \rangle to client		
$ o \rangle$ request at flash-server: um] send sealed rejection esp , $content \rangle$ to client		

Г

Figure 5: Corfu implementation – steady state protocols

Operation $reconfig(newconf)$ at client: send $\langle sealreq \rangle$ request to all flash-servers wait for reply $\langle sealresp, highind_q \rangle$ from at least one flash-server q compute highindex from all highind _q send $\langle reconfreq, newconf, highindex \rangle$ to Auxiliary	Upon $\langle sealreq \rangle$ request at flash-server q: set $sealed[enum] = true$ denote $high_q$ the highest index of filled offsets send $\langle sealresp, high_q \rangle$ to client
wait for response $\langle \text{reconfresp}, C \rangle$ increment <i>curepoch</i> and set $CF[curepoch]$ to C return C	Upon $\langle \text{reconfreq}, conf, highind \rangle$ at Auxiliary: If first reconfreq for $enum + 1$ set $CF[enum + 1] = (conf, highind)$ send $\langle \text{reconfresp}, enum + 1, CF[enum + 1] \rangle$ to client
Upon start-up at token-server: send (indexquery) request to all flash-servers wait for replies (indexresp, i_q) from all flash-servers set $\ell\ell$ to maximal i_q begin servicing tokenreq requests	Upon $\langle indexquery \rangle$ request at flash-server: denote $high_q$ the highest index of filled offsets send $\langle indexresp, high_q \rangle$ to client

Figure 6: Corfu implementation – recovery protocols

5 The Log is Everything

We are currently engaged in building a variety of applications on top of Corfu. A comprehensive description of these applications is beyond the scope of this manuscript. Here, we provide brief insight into the special considerations in constructing applications that make use of a shared global log.

The immediate challenge in using the Corfu log is reading the state of the replicated service. Remember that we queue store requests in the log, but we never really 'execute' them: the log **is the store**.³ But in order to read from the service, we may potentially need to process all preceding entries, or, at the very least, we need to locate the **latest** offset in the log which stores any data object we wish to read.

Here, again, we deviate considerably from generic SMR. In SMR, the simplest way to read the state of a replicated service is to inject a read-command into the ordered sequence of commands. In turn, replicas execute reads in sequence order, like other commands. The reader needs to wait for responses from F + 1 replicas (as usual), and compute a result which reflects the most up-to-date state of the service. Often, implementing reads in this manner would be terribly wasteful: In our setting, it is a bad idea to store numerous read commands on storage-servers just for the sake of ordering, because it uses up flash space and erase cycles.

Before we proceed with our approach, it is important to first acknowledge existing read optimizations in SMR, and explain why we cannot employ them in Corfu. In two recent ones, for example, local reads are aided with leader directives: The Paxos implementation in SRS [6] has the leader inform readers about the sequence-number of latest committed command; data transfer may proceed from any replica, conditioned on it catching up with committed commands up to the indicated index. Megastore [3] has a different optimization, based on replicas opting-out of local reads when they lose connection with the leader. Unfortunately, none of these optimizations (and others) are applicable in our settings, because Corfu deviates from standard SMR in how it executes commands. More specifically, standard SMR works in two stages: One to decide on a command, and another to inform replicas that the command has committed. Replicas in an SMR deployment may execute a command after they learn that it has committed (and learn the sequence ordering). It should be noted that, with a stable leader, this can actually be done quite efficiently, by having the leader embed commit information in its next command proposal. Nevertheless, this requires replicas to buffer commands until a commit indication arrives, which is more than we want to require from our simple, high-throughput flash-drives. Moreover, in our leaderless implementation clients would incur an extraneous round of communication in order to send a commit indication, thus cutting our throughput to a half.

The Corfu design is fundamentally different: we envision using the log as the store itself, rather than as transient queue of updates. That is, whereas in SMR we think of the queue of updates as transient and the service state as permanent, in Corfu, we envision the log as persisting information, whereas everything else can be maintained as soft state.

We demonstrate this through two classes of infrastructure service-layers on top of Corfu whose role is to efficiently learn committed commands and to inform higher level applications as needed. The first is a key-value store with strict atomicity guarantees. Because the Corfu log is persistent, we can implement a linear key-value store in a log-structured manner as follows:

'Pessimistic' key-value store: A 'pessimistic' use of the log treats every log-entry as a committed update to a key. Each update overwrites previous ones to the same key. Hence, the latest update in the log to any particular key reflects its most up-to-date value. In order to support consistent reads of the key-value space, we need to find the latest offset pertaining to the key we read. Here, an infrastructure

³Notwithstanding, we may naturally allow applications to checkpoint state periodically in order to reclaim a prefix of the log and to allow efficient state reconstruction.

service may be used to avoid having to pull updates from the log on each read: A map-server can maintain a translation of keys to log-offsets, and thus concentrate the effort of keeping up with the log. The map can be trivially partitioned across the key space for higher throughput, and since it does not need to persist any information, it does not become an IO bottleneck.

Using the power of a shared log, we may easily extend the key-value service to support atomic multikey operations. All we need to do is to demarcate relevant blocks of updates as atomic transactions and let the map-server take effect of each transaction block only when it is fully appended to the log.

The second service class uses the log as an optimistic concurrency-control engine, as follows:

'Optimistic' transactional store: An 'optimistic' use of the log treats every log-entry as speculative. The decision to commit or abort an entry depends on entries that precede it in the log and the isolation policy in effect. Here, too, we need to demarcate blocks of speculative updates to form atomic transactions. The last log-entry in the block may contain commit restrictions, namely, a read-set: we abort the transaction if it has read stale data. The *Meld* algorithm [4] contains an advanced conflict resolution scheme which exemplifies this use.

One of the advantage of pessimistic execution is that we can play the log completely autonomously from anywhere. Therefore, we may support a fully replicated transactional service with strict atomicity guarantees, with the Corfu log used as a shared back-end log.

References

- I. Abraham, G. V. Chockler, I. Keidar, and D. Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. In *Proceedings of the twenty-third annual ACM symposium on Principles* of distributed computing, PODC '04, pages 226–235, New York, NY, USA, 2004. ACM.
- [2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: a fast array of wimpy nodes. In *Proceedings of the 22nd Symposium on Operating Systems Principles* (SOSP'09), pages 1–14, 2009.
- [3] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data Systems Research*, pages 223–234, 2011.
- [4] P. Bernstein, C. Reid, and S. Das. Hyder a transactional record manager for shared flash. In Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (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] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX conference* on Networked systems design and implementation, NSDI'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.
- [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. 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.
- [9] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv., 36:372–421, December 2004.
- [10] Fusion-io Corporation. Fusion-io and supermicro break the million iops barrier. http://www. fusionio.com/press, 2011.
- [11] 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.
- [12] R. V. G.V. Chockler, I. Keidar. Group communication specifications: a comprehensive study. ACM Computing Surveys, 33(4), Dec. 2001.
- [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. Brief announcement: Vertical Paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of Distributed Computing (PODC'09)*, pages 312–313, 2009.
- [16] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machine for wans. In 8th Usenix Symposium on Operating Systems Design and Implementation (OSDI), pages 369–384, 2008.
- [17] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.
- [18] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation* - *Volume 6*, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.