Virtual Consensus in Delos

Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song, *Facebook, Inc.*

https://www.usenix.org/conference/osdi20/presentation/balakrishnan

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

Virtual Consensus in Delos

Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi
Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski
Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, Yee Jiun Song
Facebook, Inc.

Abstract

Consensus-based replicated systems are complex, monolithic, and difficult to upgrade once deployed. As a result, deployed systems do not benefit from innovative research, and new consensus protocols rarely reach production. We propose virtualizing consensus by virtualizing the shared log API, allowing services to change consensus protocols without downtime. Virtualization splits the logic of consensus into the VirtualLog, a generic and reusable reconfiguration layer; and pluggable ordering protocols called Loglets. Loglets are simple, since they do not need to support reconfiguration or leader election; diverse, consisting of different protocols, codebases, and even deployment modes; and composable, via RAID-like stacking and striping. We describe a production database called Delos[1] which leverages virtual consensus for rapid, incremental development and deployment. Delos reached production within 8 months, and 4 months later upgraded its consensus protocol without downtime for a 10X latency improvement. Delos can dynamically change its performance properties by changing consensus protocols: we can scale throughput by up to 10X by switching to a disaggregated Loglet, and double the failure threshold of an instance without sacrificing throughput via a striped Loglet.

1 Introduction

The last decade has seen significant research advances in faster and more flexible consensus protocols. Unfortunately, systems that use consensus to replicate state are monolithic, complex, and difficult to evolve. As a result, deployed systems rarely benefit from new research ideas (e.g., ZooKeeper [20] still runs a decade-old protocol [22]); in turn, such ideas only have real-world impact when entire new production systems and applications are built from scratch around them (e.g., VMware's CorfuDB [1] uses sharded acceptors [7, 16]; Facebook's LogDevice [3] implements flexible quorums [19]; etcd [2] runs on Raft [39]). Contrast this state of affairs with other areas such as OSes and networks, where modular design and clean layering allow plug-and-play adoption of new mechanisms and incremental improvement of existing ones: for example, a new type of SSD, a new filesystem layout, or a new key-value store like RocksDB can each be deployed with no modification to the layers above or below it.

Recently, the shared log has gained traction as an API for consensus in research [7–9, 16, 37] and industry [1, 3, 23, 47]. Applications can replicate state via this API by appending updates to the shared log, checking its tail, and reading back updates from it. The consensus protocol is hidden behind the shared log API, allowing applications to bind to any implementation at deployment time.

Unfortunately, an API on its own is not sufficient to enable incremental evolution. First, new implementations of the shared log are *difficult to deploy and operate*: no support exists for upgrading and migrating applications to different implementations without downtime, which is untenable for highly available services. Second, new implementations are *difficult to develop*: the consensus protocol implementing the shared log is itself a complex distributed system containing a data plane (for ordering and storing commands durably) and a control plane (for reconfiguring leadership, roles, parameters, and membership). Existing protocols such as Raft aggressively combine both planes into a single protocol; in doing so, they give up the ability to incrementally change the data plane (i.e., the ordering mechanism) without reimplementing the entire control plane. As a result of these two limitations, systems have to be written and deployed from scratch around new consensus protocols (e.g., ZooKeeper cannot be upgraded to run over Raft [39] or CORFU [7]); and protocols have to be rewritten around new ordering mechanisms (e.g., Raft cannot be changed easily to support sharded acceptors).

In this paper, we virtualize consensus by virtualizing the shared log API. We propose the novel abstraction of a virtualized shared log (or *VirtualLog*). The VirtualLog exposes a conventional shared log API; applications above it are oblivious to its virtualized nature. Under the hood, the VirtualLog

[1] Delos is an island in the Cyclades, a few hundred miles from Paxos and Corfu.

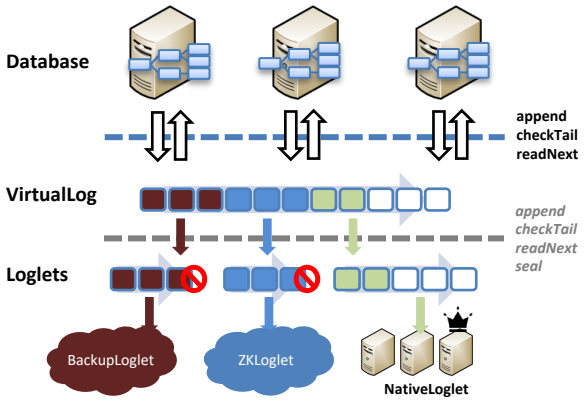

Figure 1: *Virtual consensus: servers replicate state via a VirtualLog, which is mapped to underlying Loglets.*

chains multiple shared log instances (called *Loglets*) into a single shared log. Different Loglets in a VirtualLog can be instances of the same ordering protocol with different parameters, leadership, or membership (e.g., different instances of MultiPaxos [45]); they can be entirely distinct log implementations (e.g., Raft [39], LogDevice [3], Scalog [16], or CORFU [7]); or they can be simple log shims over external storage systems (e.g., ZooKeeper [20] or HDFS [43]). Virtualization enables *heterogeneous reconfiguration*: a single VirtualLog can span different types of Loglets and dynamically switch between them.

We implemented virtual consensus in Delos, a database that stores control plane state for Facebook. Delos has been in production for over 18 months and currently processes over 1.8 billion transactions per day across all our deployments. One of its use cases is Twine's Resource Broker [44], which stores metadata for the fleet of servers in Facebook; each Delos deployment runs on 5 to 9 machines and manages server reservations for a fraction of the fleet. Internally, Delos is a shared log database [7, 8, 33]; it replicates state across servers by appending and playing back commands on the VirtualLog. Delos supports multiple application-facing APIs; we have a Table API in production, while a second ZooKeeper API is under development.

Virtual consensus in Delos *simplified the deployment and operation* of consensus implementations. Virtualization slashed time to deployment since we had the ability to deploy the system rapidly with an initial Loglet implementation, and later upgrade it without downtime. We reached production within eight months with a simple Loglet implemented as a shim over ZooKeeper (ZKLoglet); later, we obtained a 10X improvement in end-to-end latency in production by migrating online to a new, custom-built Loglet implementation (NativeLoglet). We also enabled seemingly infinite capacity for the VirtualLog by migrating older segments to a Loglet layered on cold storage (BackupLoglet); in turn, this allowed Delos to provide operators with a point-in-time restore capability. Loglets can be converged (i.e., collocated on the same machines as the database) or disaggregated (i.e., running on an entirely different set of machines), allowing operators to switch the Delos deployment mode on the fly to obtain different performance and fault-tolerance properties.

Virtual consensus also *simplifies the development* of new consensus implementations. Virtualization splits the complex functionality of consensus into separate layers: a control plane (the VirtualLog) that provides a generic reconfiguration capability; and a data plane (the Loglet) that provides critical-path ordering. While the VirtualLog's reconfiguration mechanism can be used solely for migrating between entirely different Loglet implementations, it can also switch between different instances of the same Loglet protocol with changes to leadership, roles, parameters, and membership. As a result, the Loglet itself can be a statically configured protocol, without any internal support for reconfiguration. In fact, the Loglet does not even have to implement fault-tolerant consensus (i.e., be highly available for appends via leader election), as long as it provides a fault-tolerant `seal` command, which is theoretically weaker and practically simpler to implement. When a Loglet fails for appends, the VirtualLog seals it and switches to a different Loglet, providing leader election and reconfiguration as a separate, reusable layer that can work with any underlying Loglet.

Accordingly, new Loglets are simple to design and implement since they are not required to implement fault-tolerant consensus or reconfiguration. To demonstrate this point, we describe the Delos NativeLoglet, which uses a primary-driven protocol that is unavailable for appends if the primary fails, but can support seals as long as a quorum is alive. New Loglets are also easy to construct via RAID-like composition; for example, we describe StripedLoglet, a thin shim layer that stitches together multiple Loglets to enable behavior equivalent to rotating sequencers [35] and sharded acceptors [7, 16].

Virtual consensus has some limitations. The reusability of VirtualLog-driven reconfiguration comes with a latency hit for certain types of reconfigurations such as planned leader changes. Loglets can optimize for specific cases by relying on their own in-band reconfiguration instead of the VirtualLog. A second limitation relates to generality: since we virtualize a specific API for consensus that captures a total order of commands, we do not currently support protocols that construct partial orders based on operation commutativity [26, 36]. In future work, we plan to extend virtual consensus to partially ordered shared logs [33].

We are the first to propose a virtualized shared log composed from heterogeneous log implementations. Prior work composes a logical shared log directly from storage servers [7, 9, 16, 47]; or virtualizes in the opposite direction, multiplexing homogenous streams over a single shared log [8, 48]. Delos is the first replicated database that can switch its consensus implementation on the fly to different

protocols, deployment modes, or codebases. While the theory of consensus has always allowed learners and acceptors to be disaggregated, Delos is also the first production system that can switch between converged and disaggregated acceptors.

In this paper, we make the following contributions:

- We propose virtualizing consensus via the novel VirtualLog and Loglet abstractions; and describe Delos, a storage system that implements these abstractions.
- Using production data, we show Delos upgrading to NativeLoglet without downtime for a 10X latency improvement.
- Using experiments, we show that Delos can: A) switch to a disaggregated Loglet for a 10X improvement in throughput under a 15ms p99 SLA; B) double its failure threshold without lowering throughput via a StripedLoglet that rotates sequencers. Further, we show that StripedLoglet can support over a million 1KB appends/s on a log-only workload by sharding acceptors.

2 The Path to Virtual Consensus

Virtualization for faster deployment: In 2017, Facebook needed a table store for its core control plane services with strong guarantees on durability, consistency, and availability. Two practical imperatives drove the design and development of this system: fast deployment (it had to reach production within 6-9 months) and incremental evolution (it had to support better performance over time).

At the time, Facebook already operated four different storage systems: a ZooKeeper service; a shared log service based on LogDevice [3]; a key-value service called ZippyDB [5]; and a replicated MySQL service [13]. None of these systems fit the exact use case, either due to a mismatch in API (e.g., ZooKeeper does not provide a table API) or fault-tolerance guarantees (e.g., the MySQL service provided inadequate availability).

Further, these systems could not be easily modified to provide the required API or guarantees. Each of them was a monolith: the database API could not be easily changed, nor could the underlying consensus protocol be re-used. In some systems, no abstraction boundary existed between the database and the consensus protocol. In other systems, an abstraction boundary did exist in the form of a shared log API, allowing the underlying consensus protocol to be reused; however, no support existed to migrate from one implementation of the abstraction to another.

Building yet another monolithic system from scratch – including a new consensus implementation – was not feasible since we had to hit deployment within 6-9 months. Layering the system over an existing shared log such as LogDevice would allow us to reach production quickly, but also tie us for perpetuity to the fault-tolerance and performance properties of that consensus implementation.

Our solution was to virtualize consensus. In the remainder of this paper, we describe how virtual consensus allowed us to reach production quickly with an existing implementation of consensus, and then migrate without downtime to new implementations.

Virtualization for faster development: Beyond fast initial deployment and online migration, virtualization also enabled faster development of new consensus implementations. On its own, the shared log abstraction simplifies consensus-based systems, separating applications from the logic of consensus via a data-centric API. Virtualizing the shared log further splits the consensus protocol into two layers: a control plane, which includes the logic for reconfiguration, and a data plane, which orders commands on the critical path.

In practice, such separation allowed us to incrementally improve the system by re-implementing just the data plane of the consensus protocol via new Loglets, while reusing the VirtualLog control plane for features such as leader election and membership changes. In the process, we completely changed the operational characteristics and performance of the system, as we describe later.

Importantly, such a separation also enables diversity in the data plane. The last few years have seen a number of consensus protocols with novel ordering mechanisms [3, 7, 14–16, 22, 24, 31, 35, 37, 39, 42], providing vastly different trade-offs between performance and fault-tolerance. By making it easier to implement such protocols and deploy them within running systems, virtualization lowers the barrier to innovation.

3 Abstractions for Virtual Consensus

In this paper, we propose virtualizing consensus by virtualizing the shared log abstraction. We have three design goals for virtualization. First, virtualization should be *transparent to applications*, which should be unmodified and oblivious to the virtualized nature of the log. Second, virtualization should allow underlying logs to be *simple and diverse*, lowering the barrier to new log implementations. Third, virtualization should allow for *migration* between implementations without downtime. We obtain these properties via two core abstractions.

In virtual consensus, the application operates above a VirtualLog, which stitches together multiple independent Loglets. The VirtualLog and Loglets expose a conventional shared log API (see Figure 2). Applications can `append` an entry, receiving back a log position; call `checkTail` to obtain the first unwritten position; call `readNext` to read the first entry in the passed-in range; and call `prefixTrim` to indicate that a prefix of the log can be trimmed. Virtualization requires two additions to this basic API: a `seal` command, which ensures that any new appends fail; and an augmented `checkTail` response

```
class ILoglet {
        logpos_t append(Entry payload);
        pair<logpos_t,bool> checkTail();
        Entry readNext(logpos_t min, logpos_t
            max);
        logpos_t prefixTrim(logpos_t trimpos);
        void seal();
}
class IVirtualLog : public ILoglet {
        bool reconfigExtend(LogCfg newcfg);
        bool reconfigTruncate();
        bool reconfigModify(LogCfg newcfg);
}
```

Figure 2: *Loglet and VirtualLog APIs.*

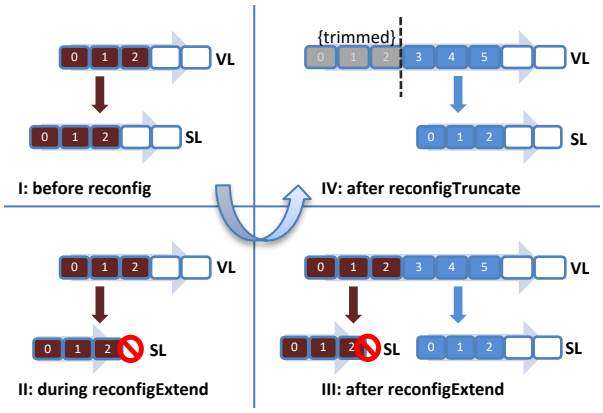

Figure 3: *The VirtualLog reconfigures from a single Loglet (I) by first sealing the Loglet (II) and installing a new Loglet (III). Later, the old Loglet is removed (IV).*

that indicates via a boolean whether the log is sealed. In addition, the VirtualLog also implements extra reconfiguration APIs to add and remove Loglets.

The VirtualLog is the only required source of fault-tolerant consensus in the system, providing a catch-all mechanism for any type of reconfiguration. The Loglet does not have to support fault-tolerant consensus, instead acting as a pluggable data plane for failure-free ordering. Existing systems typically struggle to implement monolithic consensus protocols that are simple, fast, and fault-tolerant. In virtual consensus, we divide and conquer: consensus in the VirtualLog is simple and fault-tolerant (but not necessarily fast, since it is invoked only on reconfigurations), while consensus in the Loglet is simple and fast (but not necessarily fault-tolerant). We now describe these abstractions and their interaction in detail.

3.1 The VirtualLog abstraction

The VirtualLog implements a logical shared log by chaining a collection of underlying Loglets. In this section, we use the term 'client' to refer to an application process that accesses the VirtualLog. Clients accessing the VirtualLog see a shared, append-only virtual address space that is strongly consistent (i.e., linearizable [18]), failure-atomic, and highly available. Internally, this address space is mapped to the individual address spaces of different Loglets in a chain. Operations to the VirtualLog are translated to operations on underlying Loglets based on this chain-structured mapping.

The simplest possible VirtualLog is a trivial singleton chain: $[0,\infty)$ of the VirtualLog is mapped to $[0,\infty)$ of a single Loglet. In this case, commands to the VirtualLog are passed through unmodified to the underlying Loglet. A more typical chain consists of multiple Loglets, mapping different segments of the virtual address space to each log: for example, $[0,100)$ is mapped to $[0,100)$ of Loglet A; $[100,150)$ is mapped to $[0,50)$ of Loglet B; $[150,\infty)$ to $[0,\infty)$ of C. We use the following notation for such a chain: $[0 \xrightarrow{A} 100 \xrightarrow{B} 150 \xrightarrow{C} \infty]$.

Any `append` and `checkTail` commands on a VirtualLog are directed to the last Loglet in the chain, while `readNext` commands on a range are routed to the Loglet storing that range. In the process, log positions are translated from the virtual address space to each Loglet's individual address space (for example, in the chain described above, if an `append` that is routed to Loglet C returns position 10, we map it to position 160 on the VirtualLog). Log positions can be contiguous (i.e., every position has an entry) or sparse (i.e., positions can be left unoccupied), depending on the underlying Loglet. Importantly, only the last log in the chain is appendable (we call this the active segment); the other logs are sealed and return errors on appends (we call these sealed segments).

The VirtualLog can be reconfigured to a new chain via its API (see Figure 2). The `reconfigExtend` call changes the active segment of the VirtualLog so that new appends are directed to an entirely different Loglet. Reconfigurations can also modify sealed segments via the `reconfigModify` call (e.g., to replace failed servers within a sealed Loglet). The `reconfigTruncate` call is used to remove the first sealed segment from the chain (e.g., when the VirtualLog's address space is trimmed).

3.2 VirtualLog design

The VirtualLog is composed of two distinct components: a client-side layer exposing the shared log API, which routes operations to underlying Loglets based on the chain-structured mapping; and a logically centralized metadata component (MetaStore) that durably stores the chain. Each client maintains a local (potentially stale) cached copy of the chain.

The MetaStore component has a simple API: it is a single versioned register supporting a conditional write. Reading the MetaStore returns a value with an attached version. Writing

to it requires supplying a new value and an expected existing version.

The primary technical challenge for the VirtualLog is providing clients with a shared, strongly consistent, and highly available virtual address space. In steady-state, when the chain remains unchanged, this task is trivial: the client-side layer can use its locally cached copy of the chain to route operations. However, the chain can be changed via the VirtualLog reconfiguration APIs shown in Figure 2.

Any client can initiate a reconfiguration, or complete a reconfiguration started by some other client. Reconfiguration involves three steps: *sealing* the old chain, *installing* the new chain on the MetaStore, and *fetching* the new chain from the MetaStore. The following reconfiguration protocol is expressed entirely in the form of data-centric operations against the Loglet and MetaStore; it assumes nothing about the internal implementation of either component.

Step 1: Sealing the current chain by sealing its last Loglet: The first step of a reconfiguration involves sealing the current chain (C_i) to stop new appends from succeeding within it. To seal the current chain, the reconfiguring client simply calls `seal` on the active segment, since this is the only segment that receives appends, and all other segments are already sealed. Seals on a Loglet are idempotent; accordingly, multiple clients can concurrently seal the current chain. Once the current chain is sealed by the reconfiguring client, any subsequent `append` on the current chain by a client returns an error. After sealing the active segment, the client calls `checkTail` to retrieve its tail; this determines the start of the new active segment.

Step 2: Installing the new chain in the MetaStore: Once the old chain C_i is sealed, the reconfiguring client writes a new chain C_{i+1} to the MetaStore. The MetaStore is simply a versioned register supporting a conditional write; accordingly, it only accepts the new chain C_{i+1} if the existing chain is C_i. In effect, multiple reconfiguring clients – after running step 1 idempotently – can race to install the new chain in the MetaStore, with at most one guaranteed to win. The chain is stored as a list of segments with start/stop positions, each with an opaque Loglet-specific configuration.

Step 3: Fetching the new chain from the MetaStore: In the final step, the reconfiguring client fetches the latest chain from the MetaStore. In the common case, this step can be omitted if the reconfiguring client succeeded in installing its candidate chain in Step 2. Alternatively, if the write in Step 2 failed, some other client may have won the race and installed a different chain, which we have to fetch.

Concurrency: After a client seals a chain in Step 1, it is possible that other clients continue operating within it. An `append` to the VirtualLog using the sealed chain will be routed to its last Loglet, which is now sealed in the new chain. As a result, a client issuing appends in the old chain will obtain an error code indicating that the Loglet is sealed; it will then fetch the latest chain from the MetaStore and retry.

A `checkTail` to the VirtualLog using the sealed chain also gets routed to the last Loglet in the chain. In response, the Loglet returns not just its tail position, but also a bit indicating whether or not it is sealed. If the Loglet has been sealed, then the client knows that it is operating on a stale chain, which means in turn that the computed tail is likely to not be the true tail of the VirtualLog. In this case, it fetches the latest chain from the MetaStore and retries.

Failure Atomicity: When a client encounters a sealed chain, it is possible that it does not find a newer chain in the MetaStore. This can happen if the reconfiguring client (which sealed the chain) either failed or got delayed after the seal step but before installing the new chain. In this case, after a time-out period, the client 'rolls forward' the reconfiguration by installing its own new chain. Note that the client completing the reconfiguration does not know the original intention of the failed client (e.g., if it was reconfiguring to a different Loglet type); hence, it creates a default new chain by cloning the configuration of the previous active segment.

Reconfiguration Policy: The protocol above provides a generic mechanism for reconfiguration. However, it has to be invoked based on some policy. There are three primary drivers of reconfiguration. First, planned reconfigurations (e.g., upgrading to a faster Loglet) are driven via a command line tool by operators. Second, the VirtualLog calls `reconfigTruncate` on itself when it trims the entirety of its first sealed Loglet while servicing a `prefixTrim`. For example, if the application calls `prefixTrim(100)` on chain $[0 \xrightarrow{A} 100 \xrightarrow{B} \infty]$; the VirtualLog trims all of A and then reconfigures to chain $[100 \xrightarrow{B} \infty]$. Third, individual Loglets that do not implement their own leader election or reconfiguration are responsible for detecting failures and requesting a `reconfigExtend` on the VirtualLog, as we describe later.

A subtle point is that an old chain only has to be sealed if it conflicts with a newer chain: i.e., the new chain remaps some unwritten virtual address to a different Loglet. Reconfigurations for sealed segments (e.g., to rebuild failed servers in a sealed Loglet, or to copy and remap a sealed segment to a different Loglet) do not change the locations of unwritten virtual addresses. As a result, they do not necessarily require the old chain to be sealed first before the new chain is installed; different clients can concurrently operate in the old and new chains. Similarly, truncation (i.e., removing the first segment) does not require the old chain to be sealed; if a client with the old chain tries to access an address on a truncated Loglet, it may succeed or receive an error saying the entry has been deleted. In practice, this means rebuild and GC activity on sealed segments does not interfere with appends at the tail of the VirtualLog.

3.3 The VirtualLog MetaStore

As described above, the VirtualLog stores a mapping – its chain of constituent Loglets – in a MetaStore: a single ver-

sioned register supporting a conditional write predicated on the current version. We now discuss the requirements and design space for this component.

The VirtualLog MetaStore is a necessary and sufficient source of fault-tolerant consensus in our architecture; as we describe later, Loglets are not required to implement consensus. The MetaStore has to be highly available for writes; accordingly, it requires a fault-tolerant consensus protocol like Paxos. Since the VirtualLog (and its MetaStore) is a separate, reusable layer, we need to implement this fault-tolerant consensus protocol only once. Further, the MetaStore is not required to be particularly fast, since it is accessed by the VirtualLog only during reconfigurations.

Why does the VirtualLog require its own MetaStore? Existing reconfigurable systems often store similar information (e.g., the set of servers in the next configuration) inline with the same total order as other commands (within the last configuration [34] or a combination of the old and new configurations [39]). In this case, the steps of sealing the old chain and writing the membership of the new chain can be done in a single combined operation. In the VirtualLog, this would be equivalent to storing the identity of the next Loglet within the current active Loglet while sealing it. However, such a design requires the Loglet itself to be highly available for writes (i.e., implement fault-tolerant consensus), since reconfiguring to a new Loglet would require a new entry to be written to the current Loglet. With a separate MetaStore, we eliminate the requirement of fault-tolerant consensus for each Loglet. Since one of our design goals is to make Loglets simple and diverse, we choose to use a separate MetaStore.

Using a separate MetaStore means the common-case latency of a reconfiguration consists of a `seal`, a `checkTail`, and a write to the MetaStore. In our current setting (control plane applications running within a single data center), reconfiguration latencies of 10s of ms are tenable. If reconfiguration is driven by failure, the latency of failure detection is typically multiple seconds in any case, to avoid false positives [30]. In the future, when we run across regions, it may be important to optimize for planned reconfiguration (e.g., replacing servers); since the Loglet is still available in this case, we can potentially reconfigure by storing inline commands within the Loglet itself, borrowing existing techniques such as α-windows [25, 34].

3.4 The Loglet abstraction

The Loglet is the data plane abstraction in virtual consensus: a shared log designed to operate as a segment of the VirtualLog. The requirements for a Loglet are minimal: it provides totally ordered, durable storage via the shared log API. Significantly, the Loglet can operate within a static configuration; it does not have to provide support for role or membership changes. It does not have to support leader election, either; i.e., it is not required to provide high availability for `append` calls. Instead, the Loglet provides a highly available `seal` command that prevents new appends from being successfully acknowledged. The VirtualLog uses such a sealing capability to support highly available `append` calls via reconfiguration, as described earlier in this section.

A `seal` bit does not require fault-tolerant consensus. Compared to similar data types that are equivalent to consensus, it differs from a write-once register [42], since only one 'value' can be proposed (i.e., the bit can be set); and a sticky bit [40], since it has only two states (undefined and set) rather than three. It can be implemented via a fault-tolerant atomic register that stores arbitrary values [6, 11], which in turn is weaker than consensus and not subject to the FLP impossibility result [17]. As we describe later, a `seal` is also much simpler to implement than a highly available `append`.

In addition to supporting `seal`, the Loglet provides an augmented `checkTail` to return its sealed status along with the current tail (i.e., `checkTail` returns a $(tailpos, sealbit)$ tuple rather than a single tail position). To lower the burden of implementing this extra call on each Loglet, it is designed to have weak semantics: the tail position and seal status do not need to be checked atomically. Instead, the `checkTail` call is equivalent to a conventional checkTail and a checkSeal executed in parallel, combined in a single call for efficiency.

In similar vein, a successful `seal` call ensures that any new `append` call will not be successfully acknowledged to the appending client; however, these failed appends can become durable and be reflected by future `checkTail` calls on the Loglet due to specific failure scenarios. As a result, calling `checkTail` on a sealed log can return increasing values for the tail position even after the log is successfully sealed. These 'zombie' appends on a sealed log do not appear on the VirtualLog's address space, which maps to fixed-size segments of the Loglet's address space (i.e., if the VirtualLog chain is $[0 \xrightarrow{A} 100 \xrightarrow{B} \infty]$, appends on log A can become durable beyond 100 without any impact on the VirtualLog). These semantics are sufficient to implement a linearizable VirtualLog: all we need is that any `append` on a sealed log throws an exception, and that any `checkTail` returns the seal status correctly.

The Loglet API provides a common denominator interface for different log implementations. Such implementations may provide availability for appends via internal leader election protocols; they may even support their own reconfiguration protocols for adding and removing storage servers. In such cases, the VirtualLog can be used to reconfigure across different Loglet types, while each Loglet can perform its own leader election and membership changes. To draw an analogy with storage stacks, Loglets can be functionally simple (e.g., like hard disks) or rich (e.g., like SSDs).

The log positions returned by a Loglet can be contiguous or sparse, depending on its internal implementation. Loglets that implement their own leader election or reconfiguration protocols typically expose sparse address spaces, since the log

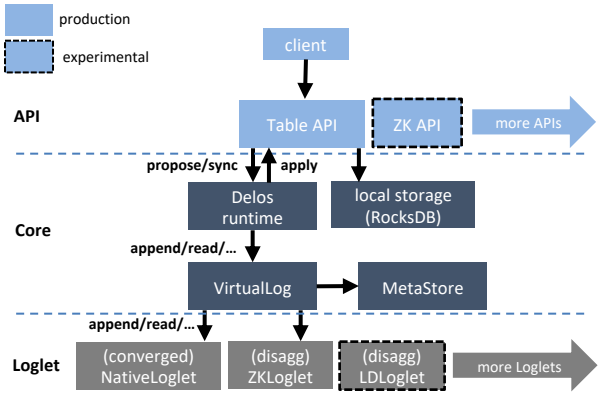

Figure 4: *Delos design: multiple APIs can run over a Core platform, which in turn can switch dynamically between multiple Loglet implementations.*

position often embeds some notion of a membership epoch.

In the case where the Loglet operates within a static configuration and relies on the VirtualLog for any form of reconfiguration, it reacts to failures by invoking `reconfigExtend` on the VirtualLog, which seals it and switches to a different Loglet. In this case, each Loglet is responsible for its own failure detection, and for supplying a new Loglet configuration minus the failed servers. In other words, while the VirtualLog provides the mechanism for reconfiguration, Loglets are partially responsible for the policy of reconfiguration.

4 Delos: Design and Implementation

Delos is a replicated storage system for control plane services. The design of Delos is driven by a number of requirements unique to control plane services: low dependencies on other services; strong guarantees for durability, availability, and consistency; and rich and flexible storage APIs. Delos is a fully replicated ACID database providing strict serializability. It does not implement transparent sharding or multi-shard transactions, but can be layered under other systems that provide these capabilities. Delos occupies a similar role in the Facebook stack to Google's Chubby [12], or the open-source etcd [2] and ZooKeeper [20].

The Delos database is similar to Tango [8]. Each Delos server maintains a local copy of state in RocksDB and keeps this state synchronized via state machine replication (SMR) over the VirtualLog. When a server receives a read-write transaction, it serializes and appends it to the VirtualLog without executing it. The server then synchronizes with the VirtualLog; when it encounters a transaction in the log (whether appended by itself or other servers), it executes the operation within a single thread as a failure-atomic transaction on its local RocksDB. The transaction returns when the appending server encounters it in the VirtualLog and applies it to the local RocksDB store. To perform a read-only transaction, the server first checks the current tail of the VirtualLog (obtaining a linearization position); it then synchronizes its local state with the VirtualLog until that position. The read-only transaction is then executed on the local RocksDB snapshot. For efficiency, Delos borrows a technique from Tango, queuing multiple read-only transactions behind a single outstanding synchronization with the VirtualLog.

As Figure 4 shows, the logic described above is separated into three layers on each Delos server: an API-specific wrapper at the top; a common Core consisting of a runtime that exposes an SMR interface, which in turn interacts with the VirtualLog; and individual Loglets under the VirtualLog. This layered design provides extensibility in two dimensions. First, Delos can support multiple Loglets under the VirtualLog, which is the focus of this paper. Second, Delos can support multiple application-facing APIs on a single platform. Each API wrapper is a thin layer of code that interacts with the Delos runtime and provides serialization logic against RocksDB. We support a Table API in production, with support for transactions, range queries, and secondary indices; we are currently deploying a second API identical to ZooKeeper. The ability to support multiple APIs on a common base is not novel: most state machine replication libraries treat the application as a black box. However, Delos provides a larger subset of application-agnostic functionality within the common Core, including local durable storage, backups, and state transfer when new servers join.

4.1 The Delos VirtualLog

In Delos, the VirtualLog is implemented via a combination of a client-side library and a MetaStore implementation. The library code implements the protocol described in Section 3.2, interacting with underlying Loglet implementations and the MetaStore. Initially, Delos went to production with the MetaStore residing on an external ZooKeeper service as a single key/value pair. Later, to remove this external dependency, we implemented an embedded MetaStore that runs on the same set of Delos servers as the database layer and VirtualLog library code.

To implement this embedded MetaStore, we used Lamport's construction of a replicated state machine from the original Paxos paper [25], which uses a sequence of independent Paxos instances. Each such instance is a simple, unoptimized implementation of canonical single-slot Paxos, incurring two round-trips to a quorum for both writes and reads. As in Lamport's description, each Paxos instance stores the membership of the next instance. We further simplify the protocol by disallowing pipelined writes at each proposer, and removing liveness optimizations such as leader election across multiple slots.

Such a protocol has inadequate latency and throughput to be used in the critical path of ordering commands, which is why

Loglet	Consensus	Deployment	Prod	Use
ZK	Yes	Disagg	Yes	Bootstrap
Native	No	Conv/Disagg	Yes	Primary
Backup	Yes	Disagg	Yes	Backup
LogDevice	Yes	Disagg	No	Perf
Striped	No	Conv/Disagg	No	Perf

Figure 5: *Different Loglet implementations in Delos.*

so many (complex) variants of Multi-Paxos exist. However, it is more than sufficient for a control plane protocol that is invoked only for reconfigurations.

4.2 The Delos Loglet(s)

Loglets can be converged, running on the Delos database servers; or disaggregated, as shims on the Delos servers accessing an external service. Each deployment model has its benefits: a converged log allows Delos to operate without a critical path service dependency, and without incurring the extra latency of accessing an external service. Disaggregation enables higher throughput by placing the log on a storage tier that can be independently scaled and I/O-isolated from the database servers. Delos currently supports three disaggregated Loglets (see Figure 5): ZKLoglet stores log entries on a ZooKeeper namespace; LogDeviceLoglet is a pass-through wrapper for a LogDevice service; BackupLoglet layers over an HDFS-like filesystem service used for cold storage. All three backing systems – ZooKeeper, LogDevice, and the HDFS-like filesystem – internally implement fault-tolerant consensus, including leader election and reconfiguration; Delos uses the VirtualLog solely to switch to/from them.

4.2.1 Loglets sans consensus: NativeLoglet

We argued earlier that Loglets can be simple since they do not require fault-tolerant consensus (i.e., highly available appends) or any form of reconfiguration. We now describe the NativeLoglet, which illustrates this point. A NativeLoglet can be either converged or disaggregated; we describe its converged form, which is how we use it in production.

Each Delos server – in addition to running the materialization logic and the VirtualLog code – runs a NativeLoglet client and a NativeLoglet server (or LogServer). One of the Delos servers also runs a sequencer component. The NativeLoglet is available for `seal` and `checkTail` as long as a majority of LogServers are alive; and for `append` if the sequencer is also alive. Each LogServer stores a local on-disk log, along with a seal bit; once the seal bit is set, the LogServer rejects new appends to its local log. The local log stores commands in a strictly ascending order that can have gaps (i.e., it may not store every command).

We first define some terms before describing the protocol. A command is *locally committed* on a particular LogServer after it has been written and synced to its local log. The *local tail* for a particular LogServer is the first unwritten position in its local log. A command is *globally committed* once it is locally committed on a majority of LogServers and all previous commands have been globally committed. The *global tail* of the NativeLoglet is the first globally uncommitted log position. The NativeLoglet does not have gaps; i.e., every position from 0 up to the global tail stores a globally committed command. Each component (i.e., LogServers, clients, and the sequencer) maintains a *knownTail* variable: the global tail it currently knows about, which can trail the actual global tail. Components piggyback *knownTail* on the messages they exchange, updating their local value if they see a higher one.

append: To append commands to the NativeLoglet, Delos servers send requests to the sequencer. The sequencer assigns a position to each command and forwards the request to all LogServers. It considers the append globally committed (and acknowledges to the client) once it receives successful responses from a majority of unsealed LogServers. If a majority of LogServers report that they have been sealed, an error is returned indicating that the NativeLoglet is sealed. In all other cases where a majority of LogServers respond negatively or fail to respond before a timeout, the sequencer retries the append. Retries are idempotent (i.e., the same command is written to the same position), and the sequencer continues to retry until the append succeeds or the NativeLoglet is sealed.

Each LogServer locally commits a particular log position n only after the previous position $n-1$ has either (1) been locally committed on the same server, or (2) has been globally committed on a majority of servers (i.e., $knownTail > n-1$). The sequencer maintains an outgoing queue of appends for each LogServer, and omits sending a command to a particular LogServer if it knows the command has already been globally committed. As a result, slow LogServers do not block appends from completing on other LogServers and a trailing LogServer can catch up more easily because it is allowed to omit storing commands that are already globally committed.

seal: Any client can seal the NativeLoglet by contacting each LogServer to set its seal bit. If the seal completes successfully – i.e., a majority of LogServers respond – future appends are guaranteed to return an error code indicating the NativeLoglet is sealed. Note that a successful seal operation can leave different LogServers with different local tails.

checkTail: This call returns both the current global tail of the NativeLoglet, as well as its current seal status. Any client can issue a `checkTail` by sending a command to all the LogServers and waiting for a majority to respond. Once a majority responds, the `checkTail` call follows a simple 5-state state machine, as described in Figure 6. For ease of exposition, we assume that no more than a majority responds; if this is not true, the protocol below can work trivially by discarding the extra responses, though in practice we use the additional infor-

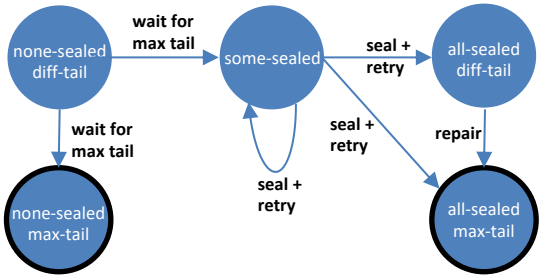

Figure 6: *NativeLoglet* `checkTail` *state machine.*

mation for better performance. Three outcomes are possible for the returned seal bits:

1. all-sealed: In the case where all responding LogServers are sealed and they all return the same local tail X, the return value is (X, true). However, it is possible that the LogServers can have different local tails (e.g., if a seal arrives while an append is ongoing). In this case, the client 'repairs' the responding LogServers to the maximum returned position X_{max}, copying over entries from the LogServers that have them (and bypassing the seal bit). It then returns X_{max} to the application. Note that repair is safe: the single sequencer ensures that there can never be different entries for the same position on different LogServers. This repair activity can result in the 'zombie' appends described in Section 3.4, where appends on a sealed Loglet are not acknowledged but can complete later due to repairs.

2. some-sealed: In the case where the responding LogServers have a mix of sealed and unsealed status bits, the client issues a `seal` first and then reissues the `checkTail`. In the absence of an adversarial failure pattern (e.g., where the seal continually lands on a different majority), the subsequent `checkTail` should return the *all-sealed* case above where all responding LogServers are sealed.

3. none-sealed: In the case where none of the responding LogServers are sealed, the client picks the maximum position X_{max} and then waits for its own *knownTail* to reach this position. While waiting, if the client discovers that some LogServer is sealed, the `checkTail` is in the *some-sealed* state described above, and proceeds accordingly. If the sequencer fails, the client's *knownTail* may never reach X_{max}; in this case, the Loglet will eventually be sealed, putting the client in the *some-sealed* or *all-sealed* state.

The latency of the `checkTail` in the *none-sealed* case depends on how quickly the client's *knownTail* is updated, along with its knowledge of the seal status of a majority of LogServers. Clients quickly and efficiently discover this information via an extra API exposed by each LogServer, which allows them to ask for notification when the local tail reaches a particular position or the LogServer is sealed.

readNext: Loglet semantics dictate that `readNext` behavior is defined only for log positions before the return value of a previous `checkTail` call from the same client. As a result, a `readNext` call translates to a read on a particular log position that is already known to exist. This simplifies the `readNext` implementation: the client first checks the locally collocated LogServer to find the entry. If it can't find the entry locally, it issues a read to some other LogServer. Note that a quorum is not required for reads, since we already know that the entry has been committed; we merely have to locate a copy.

To reiterate, the NativeLoglet does not implement fault-tolerant consensus: it becomes unavailable for appends if the sequencer fails. As a result, the `append` path has no complex leader election logic. Instead, the NativeLoglet implements a highly available `seal`, which is a trivial operation that sets a bit on a quorum of servers. The `checkTail` call follows a compact state machine for determining the seal status and global tail of the NativeLoglet. Practically, we found this protocol much easier to implement than fault-tolerant consensus: it took just under 4 months to implement and deploy a production-quality NativeLoglet.

When the sequencer or one of the LogServers fails, the NativeLoglet is responsible for detecting this failure and invoking reconfiguration on the VirtualLog (which in turn seals it and switches to a new NativeLoglet). In our implementation, we use a combination of in-band detection (e.g., the sequencer detecting that it has rebooted, or that other servers are persistently down) and out-of-band signals (via a gossip-based failure detector, as well as information from the container manager) to trigger reconfiguration. In other words, the VirtualLog provides the mechanism of reconfiguration / leader election, while the NativeLoglet handles the policy by selecting the LogServers and sequencer of the new NativeLoglet instance.

4.2.2 Loglets via composition: StripedLoglet

The StripedLoglet stripes a logical address space across multiple underlying Loglets. The mapping between address spaces is a simple and deterministic striping: logical position L_0 maps to position A_0 on stripe A; L_1 maps to position B_0; L_2 to C_0; L_3 to A_1; and so on (see Figure 7).

Incoming `append` calls to the StripedLoglet are routed to individual stripe Loglets in round-robin order. This routing is done independently at each StripedLoglet client (i.e., the Delos database servers). When the append on an individual Loglet returns with a stripe-specific position (e.g., A_1), we map it back to a logical position (e.g., L_3). However, we do not acknowledge the append to the StripedLoglet client immediately; instead, we wait until all prior logical positions have been filled, across all stripes. This ensures linearizability for the StripedLoglet: if an append starts after another append completes, it obtains a greater logical position. For example, in Figure 7, an `append` is routed to stripe B at position B_2 or L_7; but it is not acknowledged or reflected by `checkTail` until L_6 appears on stripe A at position A_2.

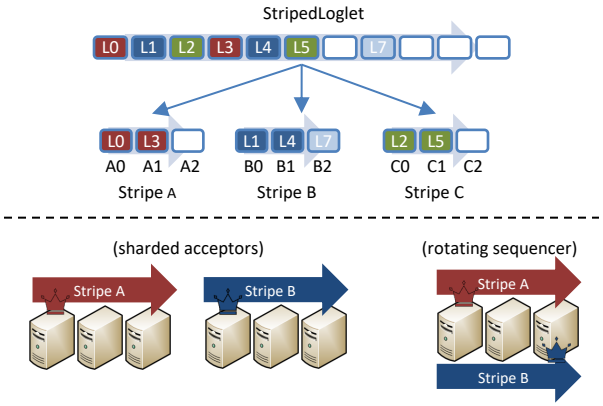

Figure 7: *The StripedLoglet stripes over underlying Loglets. Loglets can be disjoint (sharded acceptors) or have overlapping membership but different sequencers (rotating sequencer).*

A `checkTail` call fans out to all stripes and returns the first unfilled logical position, while `readNext` calls are directed to the corresponding stripe. For this protocol to work, the StripedLoglet requires the stripe Loglets to have contiguous log positions. While the NativeLoglet provides this property, LogDeviceLoglet does not: LogDevice embeds an epoch number (generated by its own internal reconfiguration mechanism) within the log position, so that positions are typically contiguous but can skip forward if an internal reconfiguration takes place.

We found composition to be a simple and effective way to create protocols with new properties. The StripedLoglet is a shim layer with only around 300 lines of code and consists entirely of invocations on the underlying Loglets; yet it provides a versatile building block for scaling throughput. We experimented with two uses of it (shown in Figure 7):

Rotating sequencer: A StripedLoglet can be composed from NativeLoglets with identical LogServers but different sequencers. For instance, a 9-node NativeLoglet will bottleneck on the single sequencer, which has to transmit each entry 8 times. Instead, a StripedLoglet can be layered over two NativeLoglet stripes, each of which uses the same LogServers but a different sequencer.

Sharded acceptors: A StripedLoglet can be layered over multiple disaggregated Loglets, achieving a linear scaling of throughput similar to CORFU [7] or Scalog [16], albeit via a design that doesn't require a centralized sequencer or separate ordering layer. StripedLoglet also differs by relying on virtualization: it implements a Loglet API over other Loglets. As a result, StripedLoglet can scale any existing Loglet while inheriting its fault-tolerance properties (i.e., the StripedLoglet fails if any of its stripes fail).

Note that the StripedLoglet code is identical for both these use cases: what changes is the composition of the individual Loglets. These Loglets can have different memberships (and even entirely different Loglet implementations) in the sharded acceptor case; or identical membership (and the same Loglet implementation) but different sequencers in the rotating sequencer case.

From the viewpoint of the VirtualLog, the StripedLoglet is like any other Loglet; it has to be sealed as a whole (i.e., every stripe has to be sealed) even if only one of its stripes needs to be changed via the VirtualLog reconfiguration mechanism. In the future, we plan to explore schemes for selectively sealing and replacing a single stripe.

5 Evaluation

We use two hardware setups for evaluating Delos. The first is the *production* hardware that most of our instances run on, which consists of 5 servers with shared boot SSDs. Since Delos has to run in a variety of data centers, we cannot assume specific or dedicated storage hardware. The second is *benchmark* hardware with dedicated NVMe SSDs. In both setups, we run within Twine [44] containers, and have production-grade debug logging and monitoring enabled.

We show numbers for two workloads. The first is real production traffic. For a representative deployment, the workload consists of 425 queries/sec and 150 puts/sec on the Delos Table API. Write size has a median of 500 bytes and a max of 150KB. Each deployment stores between 1GB and 10GB. In production, Delos takes local snapshots every 10 minutes and ships them to a backup service every 20 minutes.

The second workload is a synthetic one consisting of single-key puts and gets. The value consists of a single 1KB blob. The keys are chosen from an address space of 10M keys; we select keys randomly with a uniform distribution and generate random values, since this provides a lower bound for performance by reducing caching and compression opportunities, respectively. We pre-write the database before each run with 10GB; this matches our production data sizes.

Delos runs with two Loglets in production: ZKLoglet and NativeLoglet. In our experiments, we additionally use LogDeviceLoglet (or LDLoglet) and StripedLoglet. The external ZooKeeper service used by ZKLoglet lives on a set of 5 servers similar to our production hardware, running on shared boot SSDs and collocated with other jobs. LDLoglet uses a LogDevice service deployed on a set of 5 servers similar to our benchmark hardware, with dedicated NVMe SSDs.

In all the graphs, we report 99th percentile latency over 1-minute windows. We assume a p99 SLA of 15ms, which matches our production requirements.

5.1 The Benefit of Virtualization

Virtual consensus allowed Delos to upgrade its consensus protocol in production without downtime. In Figure 8, we show the actual switch-over happening from ZKLoglet to

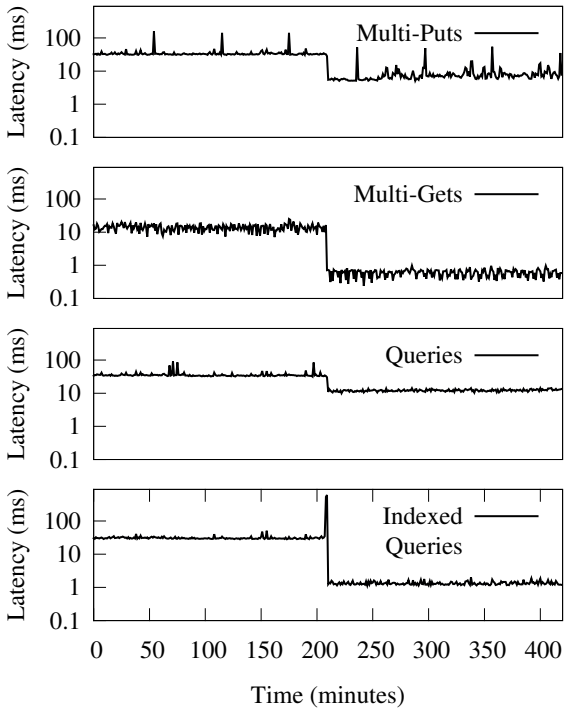

Figure 8: *Production upgrade from ZKLoglet to NativeLoglet (log-scale y-axis).*

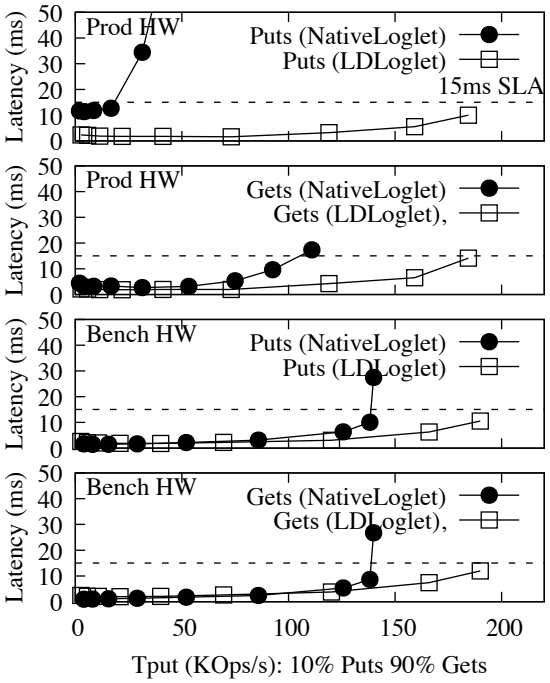

Figure 9: *Delos can scale throughput between 10X (top) and 33% (bottom) for different HW types with a disaggregated LDLoglet instead of a converged NativeLoglet.*

a converged NativeLoglet for the first time on a Delos production instance on April 2nd 2019. Switching to a different log implementation provides substantially lower latency for a production workload. The graph shows p99 latencies for four categories of Table operations: we see 10X improvements for multi-gets and indexed queries, and a 5X improvement for multi-puts. Additionally, the switch-over happens without downtime: p99 latency spikes for indexed queries during reconfiguration, but otherwise service availability is not disrupted. The latency improvement is largely due to the unoptimized nature of our ZKLoglet implementation, which simply writes a new sequential ZooKeeper key on each append.

Interestingly, the graph shows two reconfigurations: the first is a `reconfigExtend` that seals the ZKLoglet and switches the active segment to a NativeLoglet, causing the visible shift in performance; the second, which happens a few minutes later, is a `reconfigTruncate` that removes the old ZKLoglet segment from the VirtualLog, but does not require a seal (as described in Section 3.2) and hence does not cause any disruption. The hourly spikes in multi-puts are due to periodic large writes from the application.

Delos can scale throughput by running over a disaggregated Loglet. In Figure 9, we plot throughput on the x-axis and p99 latency on the y-axis, for the synthetic workload with 90% gets and 10% puts. We compare the converged NativeLoglet vs. the disaggregated LDLoglet. In the top two graphs, the Delos database runs on production HW with shared SSDs; latency with NativeLoglet starts rising at 15K ops/s for puts due to contention between the Loglet and the database. With a disaggregated LDLoglet running on 5 other machines, throughput scales 10X higher at 150K ops/s without breaching 15ms p99 latency for either puts or gets. This 10X improvement is partly due to more HW (twice the machines); better HW for the log (LDLoglet runs over dedicated NVMe SSDs); and less I/O contention (the database and log are on different machines).

In the bottom two graphs, Delos runs on benchmark HW with dedicated SSDs; the performance hit for the converged NativeLoglet is less stark due to more IOPS to share between the log and database, with latency rising for both puts and gets at around 139K ops/s. The disaggregated LDLoglet provides 33% higher throughput at 190k ops/s. For both types of HW, disaggregation allows the shared log to utilize a separate, dedicated set of resources. We get similar results running against a disaggregated NativeLoglet instead of LDLoglet, but wanted to highlight Delos' ability to run over different consensus implementations.

Delos can switch dynamically between converged and disaggregated modes without downtime. Figure 10 (Left) demonstrates the ability of Delos to change dynamically between converged and disaggregated modes, and the utility of doing so in order to handle workload shifts. In this experiment, we run the synthetic workload on the high-contention production HW. We want to maintain a 15ms p99 latency SLA.

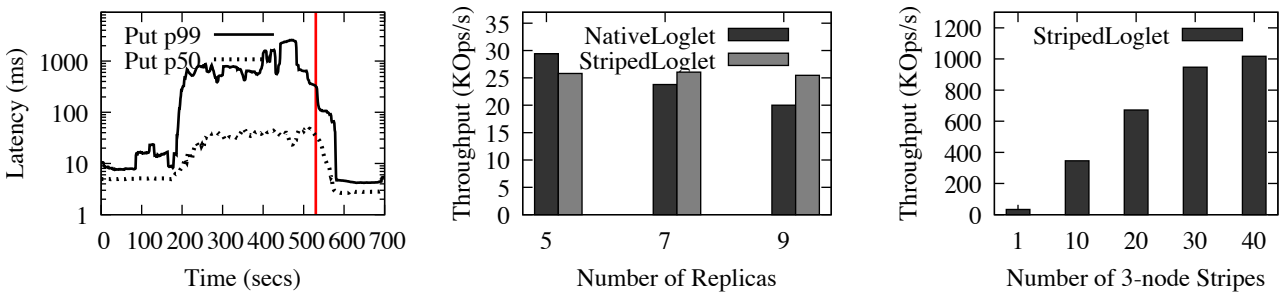

Figure 10: ***Left:*** *Delos can dynamically switch from converged (NativeLoglet) to disaggregated (LDLoglet) logging to handle workload shifts (log-scale y-axis).* ***Middle:*** *StripedLoglet (rotating sequencer) alleviates the sequencer bottleneck as Delos scales.* ***Right:*** *StripedLoglet (sharded acceptors) scales to 1M+ appends/s for a log-only workload.*

For the first 180 secs, we run a constant, low workload of 100 puts/sec; after that, we increase the workload to 2500 puts/sec. Delos initially runs over the NativeLoglet, which meets the 15ms SLA for the low workload. But when the workload switches, Delos+NativeLoglet is no longer able to keep up due to I/O contention for the SSDs, with p99 latency degrading to over a second. At around 530 secs, we reconfigure to use LDLoglet; this reduces I/O pressure on the local SSDs, allowing p99 latencies to drop back to under 15ms (after a 60-second lag due to the 1-minute sliding window).

If a disaggregated log provides better throughput and lower latency, why not always use one? First, disaggregation is inefficient from a resource standpoint for low workloads, using 10 machines compared to 5 with a converged log. Second, converged Delos does not depend on any external service in the critical path, which is important for some control plane applications.

New protocols with useful properties can be implemented via Loglet composition. In the NativeLoglet, all appends are routed via the sequencer node. For a 100% 1KB put workload on a 5-node cluster, Delos is bottlenecked by the IOPS of the NativeLoglet LogServers. However, when we run Delos over 9 LogServers for higher fault-tolerance, the bottleneck shifts to the NativeLoglet sequencer, which now has to send out each entry 8 times. If we instead use a StripedLoglet over 2 NativeLoglets (each with the same set of LogServers but different sequencers), we rotate the sequencing load across two servers. As Figure 10 (Middle) shows, Delos+StripedLoglet runs 25% faster than Delos+NativeLoglet with 9 nodes on the benchmark HW.

We also ran log-only experiments with StripedLoglet in Figure 10 (Right). We created a StripedLoglet over multiple 3-node NativeLoglets, and appended 1KB payloads from 20 VirtualLog clients. We see linear scaling of throughput as we go from 1 stripe (3 LogServers) to 30 stripes (i.e., 90 LogServers); beyond that, our clients became the bottleneck. The LogServers run on shared NVMe SSDs, which provide 30K IOPS with a p99 of 75ms; we report the maximum throughput for each configuration with a p99 latency of under 75ms. We obtained similar results with 4KB payloads (750K appends/s with 30 shards); this is the highest reported single-log throughput in a non-emulated experiment, exceeding CORFU (570K/s) and Scalog (255K/s). Delos cannot leverage such a high-throughput log, since it bottlenecks on log playback; we plan to explore selective playback [8], as well as compare against Scalog's higher emulated numbers.

5.2 The Cost of Virtualization

Virtualization is inexpensive in terms of critical path latency. In most cases, the VirtualLog acts as a simple pass-through layer. Figure 11 (Left) shows the p99 latency of VirtualLog and NativeLoglet operations; this data is measured over a one-hour time period on a production cluster running over the NativeLoglet. For `append` and `checkTail`, virtualization adds 100-150 μseconds to p99 latency; this is largely due to the overhead of an asynchronous Future-based API. In contrast, `readNext` is a synchronous pass-through call and adds only a few μseconds.

Reconfigurations occur within 10s of ms. In Figure 11 (Middle), we show a histogram of all reconfigurations in a 1-month period on our production clusters. Since `reconfigTruncate` does not call `seal`, it has lower latency than `reconfigExtend`. For our applications, reconfiguration latencies of 10s of ms are tenable. The vast majority of these reconfigurations are triggered by 1) continuous deployment of software upgrades; and 2) machine preemptions for hardware maintenance, kernel upgrades, etc. Actual failures constitute a small percentage of the reconfigurations. In practice, clusters see a few reconfigurations per day; for example, one of our production clusters was reconfigured 98 times in the 1-month period.

Virtualization does not affect peak throughput. We performed an apples-to-apples comparison of Delos to ZooKeeper on our benchmark HW. We translate puts into SetData commands and gets into GetData commands on the

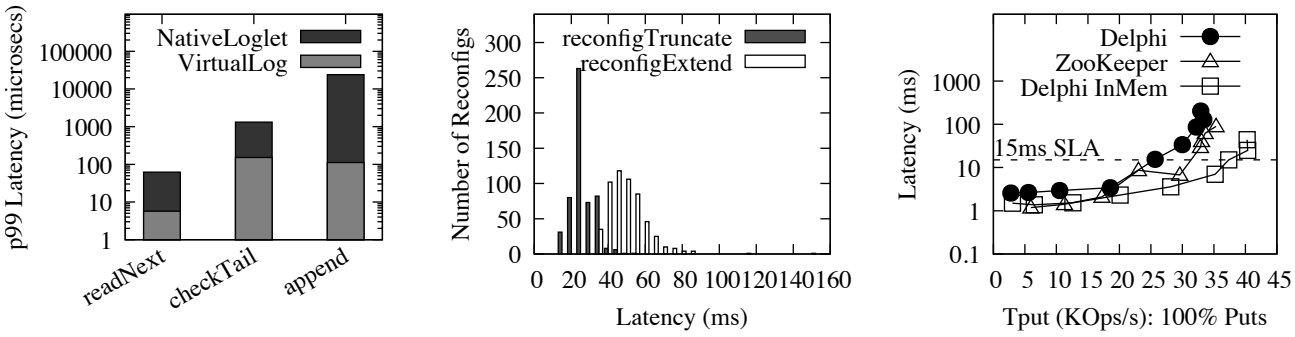

Figure 11: ***Left:*** *virtualization overhead is low in production: 6 µs for readNext and 100 to 150 µs for append / checkTail p99.* ***Middle:*** *reconfigurations take tens of ms in production.* ***Right:*** *Delos+NativeLoglet matches ZooKeeper performance.*

ZooKeeper API. Since ZooKeeper does not support more than a few GB of data, we ran with a 1GB database. Figure 11 (Right) shows that ZooKeeper can provide over 30K puts/sec before p99 latency degrades beyond 15ms. In contrast, Delos+NativeLoglet manages around 26K puts/sec. The primary reason for the performance difference is that ZooKeeper stores its materialized state in memory while Delos uses RocksDB. We also ran a version of Delos where the materialized state lives in memory; this prototype hit 40K puts/sec. While storing state in RocksDB causes a performance hit at small sizes, it enables us to scale; the Delos+NativeLoglet curve for a 100 GB database (not shown in the graph) is nearly identical to the 1GB case. These results show that Delos performance is comparable to unvirtualized systems.

6 Discussion

Virtual consensus provided a number of ancillary benefits for the operation of Delos.

Fate-sharing... but only when my fate is good: In production, Delos typically runs as a converged database with no external dependencies, where all logic and state (including the database and the log) resides on a set of 5 machines. However, the database / learner layer is simultaneously more fragile than the log / acceptor layer (since it is updated frequently to add features) and requires lower fault-tolerance (only one learner needs to survive, compared to a quorum of acceptors). If two converged replicas crash out of five, another failure can cause unavailability and data loss for the log. In this situation (which was not uncommon), we found it valuable to reconfigure the system to a disaggregated log, temporarily decoupling the fate of the database and the log. Once the database was restored to five replicas, we reconfigured back. This style of temporary decoupling also proved valuable when we discovered latent bugs in the NativeLoglet; we reconfigured to ZKLoglet, rolled out hotfixes, and then reconfigured back. Currently, switching between converged and disaggregated logs is a manual operation driven by operators; in the future, we may explore automated switching.

Consensus is forever... until it's not: Deletion of arbitrary entries is typically quite difficult in conventional consensus protocols. However, with virtual consensus, we can delete an entry simply by changing the metadata of the VirtualLog. Similarly, altering written entries is possible via remapping. We found this kind of surgical editing capability useful when faced with site-wide outages: on one occasion, a "poison" entry caused hangs on all learners processing the log.

ZooKeeper over Delos... over ZooKeeper: Virtualization often enables interesting and practically useful layerings; for example, it is routine in storage stacks to run a filesystem over a virtual block device that in turn is stored on a different instance of the same filesystem. Virtual consensus brings similar flexibility to replicated databases: in our current stack, we have the ability to run our experimental ZooKeeper API over the VirtualLog, which in turn can run over the ZooKeeper-based ZKLoglet.

7 Related Work

Virtual consensus builds upon a large body of work in fault-tolerant distributed systems. Most approaches to reconfigurable replication (including Viewstamped Replication [32, 38], ZAB [22], Raft [39], Vertical Paxos [27], and SMART [34]) use protocols for leader election and reconfiguration that are tightly intertwined with the ordering protocol. Virtual Synchrony [10, 46] is an exception: it uses a unified view change protocol for leader election and reconfiguration that sits above the ordering mechanism used within each view. This unified approach is also found in more recent systems such as Derecho [21] and CORFU [7]. Reconfiguration has been explored as a layer above a generic state machine by Stoppable Paxos [28, 29]; unlike Loglets, the underlying state machine has to implement fault-tolerant consensus.

Virtual consensus borrows many ideas from this literature, combines them, and applies them to a production system: for example, unified leader election and reconfiguration (Vir-

tual Synchrony); a segmented total order with potentially disjoint configurations (SMART); an external auxiliary (Vertical Paxos); and a generic ordering abstraction (Stoppable Paxos). However, virtual consensus also differs from all this prior work in several important aspects. First, we target and demonstrate diversity in the ordering layer (e.g., deploying new layers, switching to disaggregated mode, etc.). Second, the ordering layer is only required to provide a highly available seal, which is weaker than fault-tolerant consensus and easier to implement. Finally, virtual consensus applies and extends these ideas to shared logs, which pose unique opportunities (e.g., data-centric APIs that hide complexity) and challenges (e.g., application-driven trims and explicit reads).

To assess the generality of the Loglet abstraction, we did an informal survey of recent replication protocols. The majority of these protocols either directly expose a log API [3,7,16,37,42] or can be wrapped as a Loglet [14,15,22,24,31,35,39]. Virtualization gives us the ability to easily experiment with these protocols under Delos and deploy them to production. Other work – such as protocols that exploit speculation [41] or commutativity [4,26,36] – does not currently fit under the Loglet API.

Virtual consensus is based on the shared log approach for building replicated systems; we leverage the existence of the shared log API as a boundary between the database and consensus mechanism. Shared logs were first introduced by CORFU [7] and Hyder [9] as an API for consensus. Subsequently, CorfuDB [1] was the first production database to be deployed over a shared log API. Along similar lines, systems such as Kafka [23] and LogDevice [3] have become popular in industry, exposing large numbers of individual, independently ordered logs. In contrast, research has largely focused on scaling a single log, either via faster ordering protocols [16] or different forms of selective playback [8,48]. Rather than build a faster shared log or a more scalable database above it, virtual consensus seeks to make such systems simpler to build and deploy as they become commonplace in industry.

8 Conclusion

Virtual consensus enables faster deployment and development of replicated systems. Reconfiguration and leader election is encapsulated in a control plane (the VirtualLog) that can be reused across any data plane (Loglets). Delos is the first system that supports heterogeneous reconfiguration, allowing changes to not just the leader or the set of servers in the system, but also the protocol, codebase, and deployment model of the consensus subsystem. As a result, new systems can be developed and deployed rapidly (e.g., Delos hit production within 8 months by leveraging an external service for its Loglet); and upgraded without downtime to provide significantly different performance and fault-tolerance properties (e.g., we hot-swapped Loglets in production for a 10X latency reduction).

Acknowledgments

We would like to thank our shepherd, Jay Lorch, and the anonymous OSDI reviewers. Many people contributed to the Delos project, including Adrian Hamza, Mark Celani, Andy Newell, Artemiy Kolesnikov, Ali Zaveri, Ben Reed, Denis Samoylov, Grace Ko, Ivailo Nedelchev, Mingzhe Hao, Maxim Khutornenko, Peter Schuller, Suyog Mapara, Rajeev Nagar, Russ Arun, Soner Terek, Terence Feng, and Vidhyashankar Venkataraman. Marcos Aguilera, Jose Faleiro, Dahlia Malkhi, and Vijayan Prabhakaran provided valuable feedback on early iterations of this work.

References

[1] CorfuDB. https://github.com/corfudb.

[2] etcd. https://etcd.io/.

[3] LogDevice. https://logdevice.io/.

[4] Ailijiang, A., Charapko, A., Demirbas, M., and Kosar, T. WPaxos: Wide Area Network Flexible Consensus. *IEEE Transactions on Parallel and Distributed Systems 31*, 1 (2019), 211–223.

[5] Annamalai, M., Ravichandran, K., Srinivas, H., Zinkovsky, I., Pan, L., Savor, T., Nagle, D., and Stumm, M. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *Proceedings of USENIX OSDI 2018*.

[6] Attiya, H., Bar-Noy, A., and Dolev, D. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM (JACM) 42*, 1 (1995), 124–142.

[7] Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobber, T., Wei, M., and Davis, J. D. CORFU: A Shared Log Design for Flash Clusters. In *USENIX NSDI 2012*.

[8] Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J. D., Rao, S., Zou, T., and Zuck, A. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of ACM SOSP 2013*.

[9] Bernstein, P. A., Das, S., Ding, B., and Pilman, M. Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases. In *Proceedings of ACM SIGMOD 2015*.

[10] Birman, K. P., and Joseph, T. A. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems (TOCS) 5*, 1 (1987), 47–76.

[11] BURKE, M., CHENG, A., AND LLOYD, W. Gryff: Unifying Consensus and Shared Registers. In *Proceedings of USENIX NSDI 2020*.

[12] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of USENIX OSDI 2006*.

[13] CAO, Z., DONG, S., VEMURI, S., AND DU, D. H. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of USENIX FAST 2020*.

[14] CHARAPKO, A., AILIJIANG, A., AND DEMIRBAS, M. PigPaxos: Devouring the communication bottlenecks in distributed consensus. *arXiv preprint arXiv:2003.07760* (2020).

[15] DANG, H. T., CANINI, M., PEDONE, F., AND SOULÉ, R. Paxos Made Switch-y. *ACM SIGCOMM Computer Communication Review 46*, 2 (2016), 18–24.

[16] DING, C., CHU, D., ZHAO, E., LI, X., ALVISI, L., AND VAN RENESSE, R. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *Proceedings of USENIX NSDI 2020*.

[17] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM) 32*, 2 (1985), 374–382.

[18] HERLIHY, M. P., AND WING, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[19] HOWARD, H., MALKHI, D., AND SPIEGELMAN, A. Flexible Paxos: Quorum intersection revisited. In *Proceedings of OPODIS 2016*.

[20] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of USENIX ATC 2010*.

[21] JHA, S., BEHRENS, J., GKOUNTOUVAS, T., MILANO, M., SONG, W., TREMEL, E., RENESSE, R. V., ZINK, S., AND BIRMAN, K. P. Derecho: Fast State Machine Replication for Cloud Services. *ACM Transactions on Computer Systems (TOCS) 36*, 2 (2019), 1–49.

[22] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of IEEE DSN 2011*.

[23] KLEPPMANN, M., AND KREPS, J. Kafka, Samza and the Unix Philosophy of Distributed Data. *IEEE Data Engineering Bulletin, 38 (4)* (2015).

[24] KOGIAS, M., AND BUGNION, E. HovercRaft: Achieving Scalability and Fault-tolerance for microsecond-scale Datacenter Services. In *Proceedings of ACM EuroSys 2020*.

[25] LAMPORT, L. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS) 16*, 2 (1998), 133–169.

[26] LAMPORT, L. Generalized Consensus and Paxos. *Microsoft Research Technical Report MSR-TR-2005-33* (2005).

[27] LAMPORT, L., MALKHI, D., AND ZHOU, L. Vertical Paxos and Primary-Backup Replication. In *Proceedings of ACM PODC 2009*.

[28] LAMPORT, L., MALKHI, D., AND ZHOU, L. Stoppable Paxos. *Microsoft Research Technical Report (unpublished)* (2008).

[29] LAMPORT, L., MALKHI, D., AND ZHOU, L. Reconfiguring a State Machine. *SIGACT News 41*, 1 (2010), 63–73.

[30] LENERS, J. B., WU, H., HUNG, W.-L., AGUILERA, M. K., AND WALFISH, M. Detecting failures in distributed systems with the FALCON spy network. In *Proceedings of ACM SOSP 2011*.

[31] LI, J., MICHAEL, E., SHARMA, N. K., SZEKERES, A., AND PORTS, D. R. Just say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of USENIX OSDI 2016*.

[32] LISKOV, B., AND COWLING, J. Viewstamped Replication Revisited. In *MIT Technical Report* (2012).

[33] LOCKERMAN, J., FALEIRO, J. M., KIM, J., SANKARAN, S., ABADI, D. J., ASPNES, J., SEN, S., AND BALAKRISHNAN, M. The FuzzyLog: a Partially Ordered Shared Log. In *Proceedings of USENIX OSDI 2018*.

[34] LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., AND HOWELL, J. The SMART Way to Migrate Replicated Stateful Services. In *Proceedings of ACM EuroSys 2006*.

[35] MAO, Y., JUNQUEIRA, F. P., AND MARZULLO, K. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of USENIX OSDI 2008*.

[36] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There Is More Consensus in Egalitarian Parliaments. In *Proceedings of ACM SOSP 2013*.

[37] NAWAB, F., ARORA, V., AGRAWAL, D., AND EL ABBADI, A. Chariots: A Scalable Shared Log for Data Management in Multi-Datacenter Cloud Environments. In *Proceedings of EDBT 2015*.

[38] OKI, B. M., AND LISKOV, B. H. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of ACM PODC 1988*.

[39] ONGARO, D., AND OUSTERHOUT, J. K. In Search of an Understandable Consensus Algorithm. In *Proceedings of USENIX ATC 2014*.

[40] PLOTKIN, S. A. Sticky Bits and Universality of Consensus. In *Proceedings of ACM PODC 1989*.

[41] PORTS, D. R., LI, J., LIU, V., SHARMA, N. K., AND KRISHNAMURTHY, A. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of USENIX NSDI 2015*.

[42] SHIN, J.-Y., KIM, J., HONORÉ, W., VANZETTO, H., RADHAKRISHNAN, S., BALAKRISHNAN, M., AND SHAO, Z. WormSpace: A Modular Foundation for Simple, Verifiable Distributed Systems. In *Proceedings of ACM SoCC 2019*.

[43] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Proceedings of IEEE MSST 2010*.

[44] TANG, C., YU, K., VEERARAGHAVAN, K., KALDOR, J., MICHELSON, S., KOOBURAT, T., ANBUDURAI, A., CLARK, M., GOGIA, K., CHENG, L., CHRISTENSEN, B., GARTRELL, A., KHUTORNENKO, M., KULKARNI, S., PAWLOWSKI, M., PELKONEN, T., RODRIGUES, A., TIBREWAL, R., PAWLOWSKI, M., PELKONEN, T., RODRIGUES, A., TIBREWAL, R., VENKATESAN, V., AND ZHANG, P. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of USENIX OSDI 2020*.

[45] VAN RENESSE, R., AND ALTINBUKEN, D. Paxos Made Moderately Complex. *ACM Computing Surveys (CSUR) 47*, 3 (2015), 1–36.

[46] VAN RENESSE, R., BIRMAN, K. P., AND MAFFEIS, S. Horus: A Flexible Group Communication System. *Communications of the ACM 39*, 4 (1996), 76–83.

[47] VERBITSKI, A., GUPTA, A., SAHA, D., BRAHMADESAM, M., GUPTA, K., MITTAL, R., KRISHNAMURTHY, S., MAURICE, S., KHARATISHVILI, T., AND BAO, X. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *ACM SIGMOD 2017*.

[48] WEI, M., TAI, A., ROSSBACH, C. J., ABRAHAM, I., MUNSHED, M., DHAWAN, M., STABILE, J., WIEDER, U., FRITCHIE, S., SWANSON, S., ET AL. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *USENIX NSDI 2017*.