Log-structured Protocols in Delos

Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn
Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, Jingming Liu, Filip Gruszczynski, Jun Li
Rounak Tibrewal, Ali Zaveri, Rajeev Nagar, Ahmed Yossef, Francois Richard, Yee Jiun Song
Facebook, Inc.

ABSTRACT

Developers have access to a wide range of storage APIs and functionality in large-scale systems, such as relational databases, key-value stores, and namespaces. However, this diversity comes at a cost: each API is implemented by a complex distributed system that is difficult to develop and operate. Delos amortizes this cost by enabling different APIs on a shared codebase and operational platform. The primary innovation in Delos is a log-structured protocol: a fine-grained replicated state machine executing above a shared log that can be layered into reusable protocol stacks under different databases. We built and deployed two production databases using Delos at Facebook, creating nine different log-structured protocols in the process. We show via experiments and production data that log-structured protocols impose low overhead, while allowing optimizations that can improve latency by up to 100X (e.g., via leasing) and throughput by up to 2X (e.g., via batching).

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; • **Software and its engineering** → **Layered systems**.

KEYWORDS

State Machine Replication, Consensus

1 INTRODUCTION

Replicated databases form the foundation of modern large-scale services. Facebook operates multiple such databases, each providing some specific API, such as a MySQL deployment; the ZippyDB [15] key-value store; a ZooKeeper [31] service; the LogDevice shared log service [1]; and a Redis-like [2] data structure store. Each such database can be divided into two halves: on top, a single-node state machine with some external API (e.g., a single MySQL server); and underneath, a consensus protocol that replicates this state machine.

Recent work has simplified such databases by layering a shared log API over the consensus protocol (i.e., the bottom half), allowing a single protocol to be used underneath different databases [9–11, 23, 55]. Unfortunately, the state machine for each database (i.e., the top half) remains an independent, complex, and large monolith that does not share code or operational tooling with its counterparts. Supporting each database in production requires a team of expert engineers who understand its specific internals and toolchain. No opportunity exists for amortizing this overhead across databases: for example, the ZooKeeper and MySQL services at Facebook are maintained by entirely separate teams, each with its own codebase, tooling, and operational best practices.

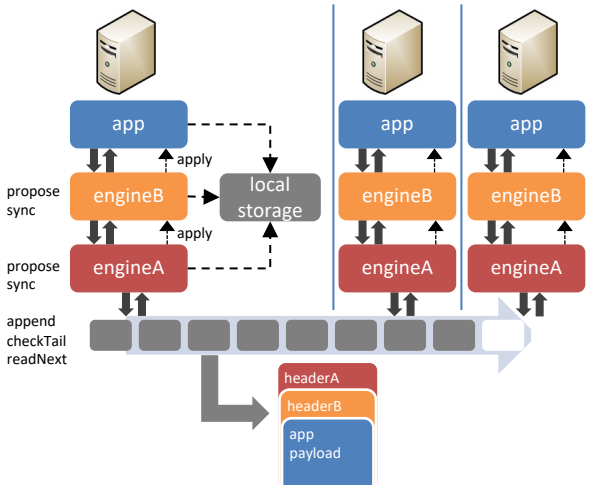

Figure 1: *Log-structured protocols are replicated state machines that can be layered like protocol stacks.*

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483544

In this paper, we observe that a significant fraction of the state machine for each replicated database comprises generic logic that does not vary between databases. Each database implements basic functionality for efficient playback and coordinated trimming of the shared log; performance optimizations such as batching, group commit [22], piggybacking [17], and lease-based local reads [27]; semantics such as exactly-once execution [39] and session-ordering [34]; tooling for repairing state and monitoring performance; and features such as non-voting followers [31] and Point-in-Time restore [8]. Given this similarity, we ask the question: can we implement such functionality once and re-use it across different databases?

Accordingly, we propose simplifying replicated databases via the novel abstraction of a *log-structured protocol*: a protocol layered over a shared log (see Figure 1). A log-structured protocol consists of an *engine* executing on each end-host, interacting with its counterparts on other hosts via the shared log. An engine can be viewed as a fine-grained replicated state machine [49] (or equivalently, a log-structured object [11, 55]), synchronizing its local replica of shared state via appends and reads on a shared log. Applications can be composed of multiple such protocols, each of which has independent state and performs some specific function. In this context, a log-structured protocol is simply a way to decompose the state and functionality of a replicated database into smaller pieces.

However, log-structured protocols are not just replicated state machines; they are also *protocols* that can be layered as a stack over the shared log. Such a protocol stack adds functionality over a shared log in much the same manner as a network protocol stack over a conventional point-to-point network [58]. New entries flow down the stack towards the shared log, while existing entries from the log flow up the stack towards the database. As with a protocol layer in a conventional network stack, an engine can piggyback headers on entries generated by higher layers; filter or reorder entries before they reach higher layers; and batch, encrypt, compress, or otherwise mutate entries en route to lower layers.

We leveraged log-structured protocols in the design, development, and deployment of Delos, a platform for building replicated databases that runs in production at Facebook. In prior work [9], we described how Delos achieves one of its novel capabilities – online upgrades of its consensus protocol – by virtualizing the shared log API. In this paper, we explore a second goal for Delos: allowing different replicated databases – with diverse application-facing APIs – to share a common codebase and deployment platform. An end-application can choose a database with an API and feature set most suited to its requirements, while obtaining identical durability and availability guarantees.

Log-structured protocols played a vital role in realizing this vision. We developed and deployed the first Delos database – DelosTable – as a stack of engines with the minimum required functionality. Subsequently, we ***incrementally upgraded*** the production DelosTable service without downtime; for example, we deployed a LogBackupEngine that coordinated the upload of the underlying log to a backup store. We deployed a second Delos service – a ZooKeeper clone called Zelos – by ***reusing code*** across databases, repurposing the production-ready stack of engines used by DelosTable. We ***tuned consistency*** for each database via specific engines; e.g., we provided the session-ordering guarantees required by the ZooKeeper API by inserting an extra SessionOrderEngine in Zelos. We also ***improved performance*** across both databases by developing common engines (e.g., a BatchingEngine that raised throughput by 2X; and a LeaseEngine that reduced read latencies by two orders of magnitude). Finally, we ***customized roles*** within each database, enabling support for passive read-only followers that executed a stripped-down subset of the protocol stack.

Many systems have abstracted out consensus as a reusable library, either via a shared log API [10, 23] or a State Machine Replication (SMR) API [43, 44]. Some systems [11, 55] have gone further by composing application state from smaller replicated state machines (e.g., composing a map from a set of linked lists). In addition, layering in replicated systems has been explored within the consensus mechanism [26, 42, 54]. Delos itself layers consensus protocols (underneath the shared log API), composing a virtual shared log from underlying log implementations (as described in prior work [9]). In contrast, this paper describes how Delos breaks new ground by layering replicated state machines (above the shared log API) as reusable, stackable protocols. To the best of our knowledge, we are also the first to report on the developer and operator experience for any form of SMR composition in a production system (prior systems with similar goals [11, 45, 55] were research prototypes).

Contributions:

- We propose the novel abstraction of a log-structured protocol: a replicated state machine over a shared log that can be layered in a protocol stack.
- We describe the design, development, and deployment of two production databases constructed using log-structured protocols. DelosTable currently serves 3.3B ops/day in production, while Zelos serves 36.5B ops/day.

- We describe nine log-structured protocols, of which seven are used in production.
- In our evaluation, we present production data showing that log-structured protocols have low overhead; and experiments showing that they can have an outsized impact on performance (e.g., a 100X latency improvement via the LeaseEngine; and a 2X throughput improvement via the BatchingEngine).

2 MOTIVATION

When the Delos project was started in late 2017, Facebook already operated a large number of distinct databases in production. This diversity of storage options placed a heavy burden on developers and operators, who were required to understand, maintain, and improve multiple complex distributed systems. In practice, large teams evolved over time around each database. Developers and operators were not fungible across teams: for instance, an engineer experienced in operating ZooKeeper would find it difficult to be productive within the MySQL deployment, and vice versa. Diversity was not uniformly beneficial for end-users either, who appreciated the rich variety of APIs available, but found it difficult to reason about the subtle differences in fault-tolerance between the options (e.g., ZooKeeper's quorum-driven consensus was more reliable than the primary-backup protocols of replicated MySQL, which also had a dependency on ZooKeeper for membership).

The Delos project was initially conceived with the goal of adding a quorum-replicated Table store to this menagerie of distributed systems, filling a gap for applications that required the fault-tolerance of ZooKeeper with the relational API of MySQL. However, a secondary goal soon emerged: could we implement the ZooKeeper API on the same codebase as this new Table store, eliminating the need to maintain and operate a separate ZooKeeper service? At the time, ZooKeeper was the primary store for the Facebook control plane, supporting thousands of use cases. To replace it, we would have to faithfully reproduce its features and idiosyncrasies. Changing all the call-sites to use a different API was untenable due to the difficulty and risk of such a migration.

2.1 A Choice of Abstractions

To implement ZooKeeper and a Table store on a single codebase, we needed a low-level distributed substrate responsible for durability, failure atomicity, and concurrency control. For the API of this substrate, we first considered a key-value store or address space, either with first-class support for transactions [16, 19, 24] or some form of conditional put that can be used by higher layers to implement locks and undo/redo logs [7, 47]. Internally, the key-value store is typically replicated using SMR [6, 19]. In this design, each application (i.e., ZooKeeper or the Table store) is responsible for translating from its own abstractions (e.g., a table or a file, respectively) to the low-level API.

Unfortunately, layering over a low-level API was not sufficient for our immediate goal: we found it difficult to implement a ZooKeeper clone over a key-value store. In particular, ZooKeeper's session-ordering guarantee (stronger than linearizability [30] in its ordering of concurrent commands, which in turn can operate on complex constructs such as watches and ephemerals) is hard to achieve over a linearizable key-value store. Second, ZooKeeper provides support for streaming, allowing non-voting followers to play its underlying totally ordered update stream. Both features emerge naturally from ZooKeeper's implementation of its API directly over SMR, where the fundamental abstraction is a total order of updates; but are difficult to extract over an intermediate key-value abstraction that hides the total order or may not contain one.

Building applications directly over SMR has other benefits. We can replicate any existing single-node software as long as it is deterministic (e.g., a SQLite [4] database), retaining the features and reliability of battle-hardened software; rather than implement new systems from scratch over a low-level API (e.g., a SQL database over a transactional key-value store [6, 16, 50], or a Table layer over an object store [7]). In addition, SMR-based applications can optionally store their data within in-memory or persistent NVM data structures [13, 18, 25, 57] that exactly match their own internal data structures; mapping these to a remote key-value store or some similarly narrow interface can be awkward, inefficient, and limiting, as others have observed [5, 20, 36, 37, 56].

Accordingly, we chose to use SMR directly as our low-level abstraction. Our initial design for Delos involved a reusable platform layer exposing an SMR API, allowing any arbitrary application black box to be replicated above it. The platform itself is also a replicated state machine, containing functionality generic to applications. We hoped to create an hourglass architecture where many applications (built by different teams with domain expertise) could be replicated on the same reusable platform, which in turn could run on any consensus protocol (via Virtual Consensus [9]). In this manner, we hoped to get the benefits of SMR while enabling reusability.

Unfortunately, structuring the platform as a monolithic state machine limited its reusability. When the ZooKeeper team at Facebook began building Zelos on the Delos platform, they needed to modify the platform

layer to obtain additional properties such as session-ordering guarantees, batching / group commit, and non-voting modes. Having two independent teams modify a monolithic state machine for entirely different business goals was not sustainable; the common code turned into a snarl of custom configurations and runtime flags for toggling between the desired behaviors of the two applications.

How could we retain the benefits of SMR, while still allowing teams to incrementally modify and selectively deploy parts of the common platform? The answer involved layering state machines, as we now describe.

3 THE LOG-STRUCTURED PROTOCOL

A log-structured protocol is an implementation of State Machine Replication over a shared log. An application can use the protocol to consistently replicate state across different servers. On each server, the application interacts with the log-structured protocol engine via the SMR API shown in Figure 2. Using the IEngine API, the application can `propose` new entries to the shared log; `register` an upcall for receiving new entries from the shared log (by implementing the IApplicator API); and `sync` to make sure that all entries in the shared log have been provided to it via the upcall. The application stores its local state in a persistent storage system (e.g., such as RocksDB), which we call the LocalStore. The LocalStore is a deterministic function of the underlying shared log; it is updated exclusively within the `apply` upcall as the engine feeds the application new log entries.

A log-structured protocol is itself a stackable replicated state machine. Engines can be layered in a stack (see Figure 3), with each engine acting as an application over another engine (i.e., the upper engine calls `propose`/`sync` on the lower engine; whereas the lower engine calls `apply` on the upper engine). Equivalently, an engine can implement the IEngine API in Figure 2 over another engine with the same API. Further, each engine has access to local state in the LocalStore. When a new entry is proposed to an engine, it can add its own header to the entry before invoking `propose` on the engine below it in the stack. Similarly, when a new entry is applied on it by the underlying engine, it can inspect its own header and modify its local persistent state in the LocalStore before invoking `apply` in turn on the layer above it.

The top-most layer of the stack is the application, which exposes some upstream API to its end-users (e.g., a Table API, or the ZooKeeper API). The bottom-most engine in the stack (BaseEngine) is specialized to operate directly over the shared log API. We first describe

```
template <class ReturnType, class EntryType>
class IEngine {
  Future<ReturnType> propose(EntryType e);
  Future<ROTx> sync();
  void registerUpcall(IApplicator<ReturnType,
      EntryType> app);
  void setTrimPrefix(logpos_t pos);
}

template <class ReturnType, class EntryType>
class IApplicator {
  ReturnType apply(RWTx txn, EntryType e,
      logpos_t pos);
  void postApply(EntryType e, logpos_t pos);
}
```

Figure 2: *Log-structured Protocol Engine APIs.*

a minimal stack consisting of an application and the BaseEngine; and then describe how other engines can be slotted into the stack.

3.1 The Application

Consider a single-node service that exposes some external API (e.g., `foo` in Figure 3); and implements this API over some local state. To replicate such a service via a stack of log-structured protocols, we split it into two parts: a Wrapper (exposing `foo`) and an Applicator (exposing the IApplicator API in Figure 2 and internally implementing `foo`). Rather than directly execute `foo`, the Wrapper serializes an incoming request (without executing it) and calls `propose` on the top-most engine.

Subsequently, the Applicator receives this request from the top engine via the `apply` upcall, executes `foo`, and returns the response to the top engine. The engine stack then relays this response back to the waiting `propose` call, completing the service invocation. Figure 3 shows the path of the request through the system. The Applicator on each server also receives and executes proposals by other servers in the system. In effect, the engine stack provides it with all proposals in the system – originating at different servers, and issued by multiple threads on each server – in the total order imposed by an underlying shared log. For proposals that did not originate at the local server, the response of the Applicator is discarded by the engine stack, since the local Wrapper is not waiting for the response.

The apply upcall on the Applicator is called by a single thread (which we call the *apply thread*). All accesses by the Applicator to the LocalStore occur within a transaction handle provided as a parameter to the apply upcall. This transaction handle provides failure atomicity: if the Applicator crashes or throws an exception, all activity within the apply upcall is rolled back.

The workflow described above obtains linearizable semantics for arbitrary read-write requests (issued concurrently within multiple threads on different servers) by funneling all activity through the Applicator's single-threaded apply upcall on each server. However, the application can optimize read-only requests by calling sync on the top-most engine rather than propose. The sync call directs the engine stack to check the current tail of the shared log and apply any new updates to the Applicator. When the sync returns, the LocalStore is guaranteed to reflect all completed writes in the system (i.e., sync returns a read-only transaction on the LocalStore representing a linearizable snapshot). The Wrapper can then directly invoke accessor functions in the LocalView from any thread using the returned transaction handle, rather than route the request via the engine stack to the Applicator's single thread. As a result, at any point in time there can only be a single writer to the LocalStore (i.e., the apply thread via the RWTx) but many readers (via the ROTx returned by each sync).

For performance, the application may want to maintain soft state within the Applicator (e.g., pre-computed aggregates). This is problematic for failure atomicity, since modifications to such state will occur outside the LocalStore transaction and not be automatically rolled back on exceptions. To allow the Applicator to update soft state, the engine stack issues a postApply upcall if the previous apply committed successfully; the Applicator can safely update soft state within this second upcall. The Wrapper can read (but not update) soft state maintained by the Applicator; this is safe as long as the Applicator acquires read-write locks in the apply upcall and releases them in the postApply.

3.2 The Bottom Engine

The BaseEngine resides at the bottom of the stack and implements the IEngine API in Figure 2 over a shared log. The primary role of the BaseEngine is to play the log forward and apply each entry to the application above it. To do so, it maintains a cursor in the LocalStore. As the BaseEngine plays each entry, it creates a LocalStore transaction context; updates its cursor in the store within the transaction; and then calls apply on the application via the single apply thread (which it spawns

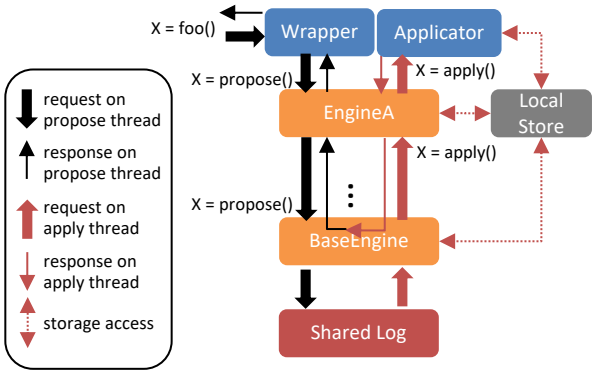

Figure 3: *The lifecycle of a proposal.*

on creation and owns). When the application returns – having processed the entry and issued writes to the transaction – the BaseEngine commits the transaction on the LocalStore. A committed transaction is visible but not immediately durable on the LocalStore, which we flush periodically in a background thread (we can replay the entry from the log if the server reboots).

The BaseEngine is a passive component; it does not initiate log-playing on its own. Rather, it plays the log forward in response to either a sync or a propose. On a sync, the BaseEngine checks the tail of the shared log to find the last committed entry; and then plays the log forward, applying each entry to the application, until it reaches that tail position. Once this point is reached, the BaseEngine creates a local read-only transaction on the LocalStore (which now reflects all log entries until the tail) and returns it to the application. For high throughput, the BaseEngine queues multiple sync calls behind a single outstanding tail check on the log, borrowing a trick found in other SMR systems [9, 11, 47].

On a propose, the BaseEngine appends the entry to the shared log; and then plays the log forward until the newly appended entry, passing each entry up to the apply upcall of the application via the apply thread. When the application apply responds to the newly appended entry, the BaseEngine relays the response to the waiting propose call. In effect, the propose call implements a form of replicated RPC that is *durable*, since it returns only once the entry is stored durably on the shared log; *failure-atomic*, since it is executed on each server within a LocalStore transaction; and *linearizable*, since it is ordered via the shared log before executing on the local server.

The BaseEngine is also responsible for the mechanism of GC: it periodically trims the shared log in a

background thread. However, it does not determine the policy of GC; the application or other engines use the setTrimPrefix call to tell the BaseEngine what prefix of the log can be trimmed safely.

3.3 The Middle Engines

Engines can be stacked on top of each other, as shown in Figure 3; i.e., an engine can implement the IEngine API over another engine with the same API. When the application calls propose, the engine piggybacks its own header on the proposed entry and passes the entry along to the engine below. The piggybacked header is then stored durably and atomically with the entry within the shared log. The engine can also choose to delay calling propose on the engine below (e.g., for batching); or manipulate the contents of the entry in some way (e.g., compress, encrypt, etc.).

An engine can also implement the IApplicator API and register apply/postApply upcalls with the engine below it. The engine has access to the LocalStore and can modify it within the apply upcall using the supplied transaction handle. When the underlying engine provides a new log entry to it via the apply upcall, the engine extracts and processes its own header; modifies its local state; and passes on the log entry to the application (or engine) above it by invoking apply in turn. The engine can also drop the entry to filter it from the engines above it. For example, Figure 4 shows the code for an engine that can be used to block entries from reaching the stack above it.

As with other layered systems, cross-layer inspection between engines is discouraged (e.g., an engine typically cannot interpret headers belonging to other engines). In principle, engines also have isolated state: they are not typically allowed to access, interpret, or modify local state in the LocalStore belonging to other engines. However, this isolation can be overridden in certain cases: we found at least one reasonable use case that requires an engine to be able to modify any state (i.e., the BrainDoctorEngine, which we describe in the next section).

In addition to piggybacking headers on outgoing entries, engines can also generate new log entries on their own; either in response to some external-facing API that can be invoked either by the application or some command-line tool (e.g., the toggleBlock call in Figure 4); or based on some internal trigger like a timer. In this case, the engine marks the entry's header before proposing it (in Figure 4, by setting a message type); the apply upcall returns immediately without sending it further up the engine stack.

```
template <class ReturnType, class EntryType>
class BlockingEngine : IEngine, IApplicator {
  Future<ReturnType> propose(EntryType e){
    e.BHdr = BEngineHdr{.msgtype=APP};
    return downstream->propose(e);
  }
  Future<ROTx> sync(){
    return downstream->sync();
  }
  ReturnType apply(RWTx txn, EntryType e, logpos_t
      pos){
    if(e.BHdr.msgType==TOGGLE){
      if(txn->get("blocked")=="True")
        txn->put("blocked", "False");
      else
        txn->put("blocked", "True");
      return Unit;
    }
    blocked = (txn->get("blocked")=="True");
    if(blocked)
      throw BlockedException{};
    else
      return upstream->apply(txn, e, pos);
  }
  void postApply(EntryType e, logpos_t logpos){
    if(e.BHdr.msgType==APP && !blocked)
      upstream->postApply(e, logpos);
  }
  void toggleBlock(){
    EntryType e;
    e.BHdr = BEngineHdr{.msgType=TOGGLE};
    downstream->propose(e).get();
  }
  void setTrimPrefix(logpos_t pos){
    downstream->setTrimPrefix(pos);
  }
}
```

Figure 4: *An example engine.*

Most engines implement setTrimPrefix as a pass-through call; i.e., they do not have an opinion on garbage collection. However, as we describe later, some engines are specifically responsible for controlling GC (e.g., an engine that is responsible for backing up the shared log may want to delay its trimming). Each such engine tracks its own opinion on the trimmable prefix, as well as the last prefix that was relayed to it by the engine above it. It invokes setTrimPrefix on the engine below it with the minimum of these two values, ensuring that trimming respects both constraints. As a result, the BaseEngine

only trims a prefix of the log if every engine in the stack allows it.

3.4 Discussion

Dynamic Updates: Inserting a new engine into the stack can be fraught: if an engine begins to operate before all servers are updated, it can cause inconsistent state across servers. In practice, we use a two-phase upgrade protocol to insert engines. In the first phase, we perform a rolling upgrade to add the engine to the stack: it can immediately piggyback its header on outgoing proposals, and generate its own proposals, but is not allowed to change the LocalStore within its `apply` upcall. In other words, it can append to the shared log but not reflect those appends on the local store.

Once all servers have the new engine, we enable it by sending a command via the log itself. This ensures that the effects of the engine are visible on the local store beyond a consistent log position, retaining the property that the LocalStore is a deterministic function of the shared log. The protocol to remove an engine is the exact reverse: we first disable the engine via the log, and then perform a rolling upgrade to remove it from the stack.

This update protocol requires all activity within an engine's `apply` upcall to be guarded by a flag which can only be toggled via the log. Further, adding an engine requires us to first upgrade all servers in the deployment with the new binary. We assume that we can upgrade, kill, or fence servers via container infrastructure [51].

Static Typing: Each engine is templated on an EntryType (which is an application-specific serializable object containing a map of headers and a specific application payload) and a ReturnType, which is returned both by the `propose` call of the engine as well as the `apply` upcall of the engine or application above it. The ReturnType is typically a variant consisting of all the return types of the individual functions used by the application (for example, if the application is a Queue service, ReturnType would consist of {PushResponse, PeekResponse, PopResponse}).

In our first implementation, each entry was a literal stack of buffers (similar to a network packet); each engine would push/pop its own header. However, we found that such a layout was brittle against stack upgrades: engines could push/pop the wrong header if they were playing entries generated by a previous iteration of the engine stack. Since the entry is instead a map of headers, each engine can simply check within the `apply` upcall if its own header is within the entry and accordingly process or pass it through to the engine above.

Exception Handling: Each engine's `apply` upcall operates inside a nested sub-transaction within the supplied transaction. If the upcall throws an exception, its own updates to the LocalStore are discarded and the entry does not reach the engines above it; but the updates of the engine below it are preserved as long as it catches the exception. As a result, the application can throw benign, deterministic exceptions within its `apply` upcall (e.g., a row_not_found error in a Table store) without interfering with the operation of the protocol engines below it. The BaseEngine relays the exception to the waiting propose thread and re-throws it, preserving RPC-like semantics for the `propose` call.

Note that if the exception is non-deterministic (i.e., thrown on one server but not on another), state can become inconsistent across servers. Oddly enough, the prescribed course of action for the system is completely opposite based on the determinism of the exception: if it is deterministic, the right thing to do is to continue processing the next log entry (maintaining availability since consistency is guaranteed); whereas for a non-deterministic exception, crashing immediately is the only safe option (giving up availability to ensure consistency). In practice, we maintain a list of exceptions known to be non-deterministic (e.g., an out-of-space error in the LocalStore) in order to crash immediately when we encounter them; and rely on checksums to quickly catch any inconsistencies.

What constitutes an application? The LocalStore exposes a key-value API (which can be implemented by an embedded store such as RocksDB [3]). As a result, the application can be any single-node database that stores its local state on this API. As described earlier, the Wrapper/Applicator components can transparently wrap such an application so that all writes are funneled to it via the single apply thread (as long as the application is deterministic for single-threaded writes). The application itself is oblivious to the existence of replicas; any coordination across replicas occurs via log-structured protocols.

Importantly, the application is not constrained to access storage via the key-value store API; it can instead implement its own persistence layer, as long as it provides a LocalStore API for the engines to use. For example, a target application is SQLite [4]; in this case, the Applicator relays SQL commands to SQLite, while engines store data via the LocalStore API implemented over SQLite. In this manner, we can leverage SMR to replicate any stateful black box (potentially one that uses NVM data structures) rather than force applications to be built over a particular storage API.

4 A TALE OF TWO DATABASES (AND NINE ENGINES)

Delos implements the log-structured protocol abstraction in modern C++. We use RocksDB as the LocalStore (with a second, experimental SQLite implementation) and Thrift as a serialization format for entries. The BaseEngine runs over the VirtualLog [9], which implements fault-tolerant consensus. Each Delos database is typically replicated on 5 or 7 machines (similar to ZooKeeper [31] or Chubby [14]); it does not natively provide sharding or cross-shard transactions, which can be obtained via a separate layer above it.

Table 5 lists the nine log-structured protocols we implemented. We mark whether each engine maintains state (i.e., it is a replicated state machine) and/or filters, modifies, observes, or delays entries en route to other engines (i.e., it is a protocol). We described the operation of the BaseEngine in the previous section; we now describe the engines we layered above it.

4.1 Rapid deployment of DelosTable

Our initial stack for DelosTable consisted of a single engine – the ViewTrackingEngine – interposed between the application (DelosTable) and the bottom BaseEngine.

A. ViewTrackingEngine (2018): This engine coordinates the trimming of the log. It tracks the playback position of each server; when all servers have played the log past some point X, the log can be trimmed until X. The ViewTrackingEngine relays this safe trim position by invoking `setTrimPrefix` on the engine below it. When a server appears to have crashed (and is no longer playing the log), the ViewTrackingEngine stops including it in its calculation of the safe trim position. As its name suggests, the ViewTrackingEngine effectively tracks a membership 'view' of servers playing the log.

The ViewTrackingEngine on a server adds a header on each outgoing `propose` with its local playback position. To determine this position, it queries the LocalStore for the last log position that has been applied and flushed durably (recall that we only flush the LocalStore periodically, not immediately after each `apply`). When its `apply` upcall is invoked with a new log entry, the ViewTrackingEngine extracts this log position and updates its local state. The local state consists of a map from server ID to playback position. In this manner, each server in the system constructs a deterministic, consistent view of the playback positions across all servers.

To add and remove servers to this view, the ViewTrackingEngine uses the log itself as a discovery and failure-detection mechanism, respectively. A server is added to the view the first time it appends an entry in the log. Conversely, if a server has not appended an entry into the log for more than some time period, another server can eject it from the known view via a command in the log. Importantly, the view is a deterministic function of the log; and as a result, so is the decision to trim a prefix of the log.

B. ObserverEngine (2018): In our first deployment, we also layered an ObserverEngine between DelosTable and the ViewTrackingEngine. The ObserverEngine is a lightweight layer that measures and externally logs end-to-end latencies on each `propose`/`sync` call; our standard practice is to layer one above each engine. The ObserverEngine has two benefits. First, it allows us to track down latency slowdowns to a specific engine in the stack. Second, it separates monitoring from core logic; as a result, we can measure the performance of each engine – and different versions of the same engine – in a generic way. In contrast, if we had allowed engines to measure their own performance, we would need safeguards against inadvertent changes to the monitoring code and variance in this code across engines.

4.2 Adding features to DelosTable

C. BrainDoctorEngine (2019): We added new features to DelosTable by deploying new log-structured protocols in the stack. One such protocol is the BrainDoctorEngine, which acts as a simple pass-through engine, with one addition: an external call that accepts a list of raw LocalStore writes and proposes it into the log. When the engine processes this control command in its `apply` upcall, it directly applies the writes to the LocalStore.

As the name suggests, the BrainDoctorEngine is used in emergencies to perform "brain surgery" on the key-value store, directly changing the state of a running Delos database without going through application logic. We added this feature after we encountered a bug in DelosTable that caused secondary indices to be written incorrectly; to fix the state of the running database, we had to route around the Table logic to directly change the secondary indices in the LocalStore.

D. LogBackupEngine (2019): We added another engine in response to a customer request for Point-in-Time restore. To enable this, we needed to copy the shared log to a backup store before trimming it; we could then reconstruct any intermediate state of the database by starting from a prior snapshot backup and playing the log backup forward. Backing up the shared log efficiently – without overloading any single node, or experiencing delays due to a node failure – required all the nodes to coordinate.

Year	Engine	Prod	State/Prot	Use Case	LOC
2018	Base	Both	Yes/No	*State Machine Replication over the log.*	1081
2018	ViewTracking	Both	Yes/No	*Track durable copies of DB for trimming the log.*	844
2018	Observer	Both	No/Yes	*Monitor underlying stack.*	208
2019	BrainDoctor	Both	Yes/No	*Edit LocalStore directly, bypassing DB.*	274
2019	LogBackup	Both	Yes/No	*Coordinate learners to back up the log.*	688
2020	SessionOrder	Zelos	Yes/Yes	*Enforce session-ordering guarantee.*	521
2020	Batching	Zelos	No/Yes	*Improve throughput via batching + group commit.*	512
2021	Time	None	Yes/No	*Implement distributed time-outs.*	904
2021	Lease	None	Yes/Yes	*Enable 0-RTT strongly consistent reads.*	371

Figure 5: *Different Log-structured Protocol Engines.*

The LogBackupEngine uses the log itself as a coordination mechanism. The state of the protocol is a map of "bids" made by each node for uploading disjoint segments of the log. Each node proposes control commands to bid for segments; these commands are applied to the map of bids. If a node successfully bids for a segment, it then begins uploading that segment to the backup store. Once it completes the backup, it marks the bid as completed via another proposal.

The LogBackupEngine delays trimming the log until it has been backed up. It does so by calling `setTrimPrefix` on the layer below it whenever the backed-up prefix advances. As described earlier, `setTrimPrefix` in each engine is implemented so each layer relays the constraints given to it: accordingly, a prefix is only trimmed by the BaseEngine if the ViewTrackingEngine and LogBackupEngine both agree. Other signals can further delay trimming; for example, a snapshot backup manager attached to the LocalStore calls `setTrimPrefix` on the stack once it backs up a snapshot corresponding to some prefix.

4.3 Stronger guarantees for Zelos

E. SessionOrderEngine (2020): In 2020, we implemented and deployed the second Delos database: a ZooKeeper clone called Zelos. The first version of Zelos reached production within months by re-using the DelosTable stack verbatim, replacing only the application layer at the top. However, we ran into a snag at this point: to be fully compatible with ZooKeeper, Zelos needed to emulate its unique ordering guarantees. Specifically, ZooKeeper provides a session-ordering guarantee: within a session, if a client first issues a write and then a concurrent read (without waiting for the write to complete), the read must reflect the write. This property is stronger than linearizability, which allows concurrent writes and reads to be ordered arbitrarily; and encompasses exactly-once semantics [39].

The SessionOrderEngine stamps outgoing proposals with a sequence number. When the proposals are applied, the SessionOrderEngine `apply` only allows entries upstream if they arrive (i.e., appear in the log) in sequence order, filtering them otherwise (e.g., #11 arrives before #10). The `propose` then re-proposes all entries since the disorder event (e.g., #10 and #11). Out-of-order arrivals are rare; the Delos stack itself does not reorder proposals as they flow down to the log or up through the `apply` upcalls, and most log implementations also retain order (e.g., via TCP connections to some leader). However, disorder can occur due to leader changes within the log implementation, or due to code changes in the Delos stack.

Unlike other engines, the SessionOrderEngine does not have a one-to-one mapping from each `propose` call to a `propose` on the underlying sub-stack; instead, it may need to do multiple proposals on the sub-stack for retries. As a result, it implements its own RPC-like bookkeeping and signaling between the apply and propose threads; each `propose` is notified by `postApply` directly, rather than waiting for the return value of the sub-stack `propose`. This style of *short-circuiting* has the side-effect that the engine can return from its `propose` earlier than the engines below it.

F. TimeEngine (2021): One of the required features for Zelos involved distribution of the update stream. In ZooKeeper, non-voting followers can play the command stream directly from the ZAB [34] protocol, which in turn allows them to create secondary indices and caches over data stored in a ZooKeeper cluster. To duplicate this feature, we needed two pieces of functionality. First, we created a passive stack with a subset of engines, as shown in Figure 6 (e.g., omitting the ViewTrackingEngine since we did not want these servers counted as durable first-class replicas). Second, we needed to delay the trimming of the log by some period of time to give non-voting followers a chance to play entries from it.

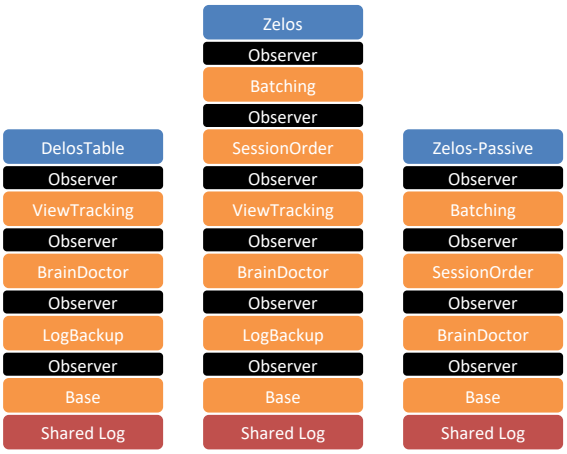

Figure 6: *Production Delos stacks.*

To support time-based trimming in a way that is robust to clock skew and drift, we implemented a TimeEngine. The TimeEngine allows the creation of a timer object which fires once a fixed amount of time has elapsed on a constant number of servers within the cluster. The timer is created via an external API of the engine, which inserts a command in the log; each server inserts a corresponding time-out command once enough time has passed on its local clock since it encountered the creation command. Time-based trimming is implemented via an external trimming module that creates a timer event at some log position; and once it expires, calls `setTrimPrefix` on the top of the engine stack. The TimeEngine reached production just after the data for this paper was collected.

4.4 Improving performance

G. BatchingEngine (2020): Since Zelos was intended to replace ZooKeeper at Facebook, performance parity was a requirement. Zelos incorporated a BatchingEngine into its stack to improve Zelos throughput. The BatchingEngine validated our approach in one significant way: it was immediately usable across both databases with zero customization, allowing us to accelerate both databases via a single engine.

The BatchingEngine illustrates an interesting point about the placement of functionality in Delos. If batching were implemented in the database above the engine stack, it would have to be re-implemented separately by DelosTable and Zelos. Conversely, if batching were implemented below the engine stack / within the shared log (i.e., at the level of the consensus protocol), the BaseEngine would have to create a separate LocalStore transaction for each sub-entry in the batch (since it would be layered above the batching logic and oblivious to it). Situating batching in the engine stack enables a group commit optimization where the entire batch is applied in a single LocalStore transaction.

H. LeaseEngine (2021): The BaseEngine has a leaderless design above the shared log: any server can `propose` a command, while each server can `sync` with the shared log to ensure strong consistency. This design has the advantage that the loss of a single server does not disrupt availability (as long as the shared log is still available). However, the `sync` before a strongly consistent read incurs a round-trip to the shared log, which in a typical implementation involves accessing either a MultiPaxos [52] leader or a majority quorum of acceptors.

In contrast, designs with a strong leader (e.g., Raft [44] and MultiPaxos [52]) can provide 0-RTT strongly consistent reads at the leader. This property is particularly useful in geo-distributed systems where each data item has a home region where most accesses originate; the leader can be situated in this region to enable strongly consistent reads without cross-region interaction.

We can enable such a capability in the Delos stack simply by inserting a LeaseEngine into the stack. The LeaseEngine elects a server as a designated proposer above the shared log. Reads at this server can be satisfied with strong consistency without accessing the shared log (i.e., the LeaseEngine's `sync` returns immediately as long as the server has a valid lease). The initial acquisition and renewal of the lease occur via commands in the log; as does the takeover of the lease by other servers if the lease-holder fails. In a sense, 0-RTT reads are not a property of a leader-based consensus protocol, but rather enabled by having a single designated proposer; which explains why leases can be implemented by inserting an engine into the stack above the shared log.

5 EVALUATION

We use a combination of production data and benchmarking experiments to evaluate the utility and overhead of log-structured protocols. As of May 2021, DelosTable is in production on 107 clusters and handles more than 3B transactions per day in aggregate, each of which can be a complex relational query. Zelos is in production on 155 clusters and handles 6.5B writes and 30B reads per day on the ZooKeeper API.

5.1 The Overhead of Layering

Log-structured protocols are lightweight. We first report on the deployed footprint of each log-structured protocol in our production systems. Figure 7 summarizes a single

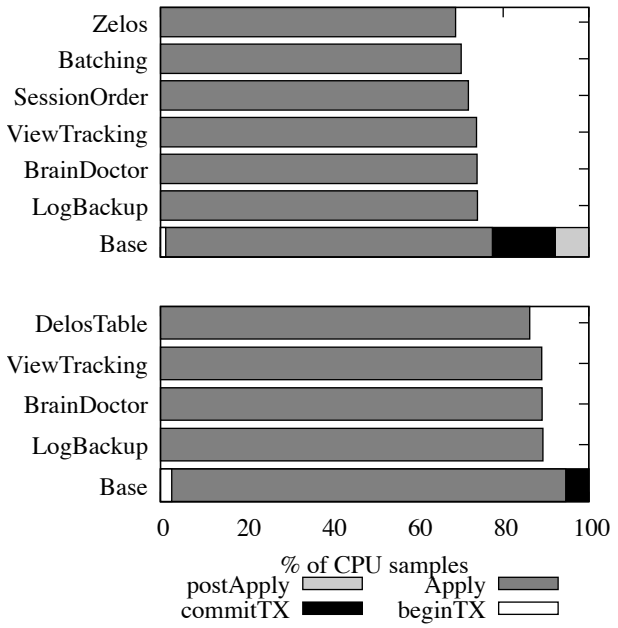

Figure 7: *Fleet-wide sampling of the apply thread in production clusters shows layering adds low overhead.*

week of stack samples across all DelosTable and Zelos production clusters. This data is obtained by a tool that runs on each Facebook server and periodically samples the current stack for all running threads in the process; we filter out the samples belonging to DelosTable/Zelos processes, and further extract out the stack for the apply thread.

Each graph in Figure 7 has a single bar for each engine deployed in the production deployment (exactly matching the stacks described in Figure 6). The bar for an engine describes the percentage of apply thread stack samples that include that engine's `apply` method. From the graphs, we can see that the apply thread spends most of its time in the application's `apply` upcall. Each individual engine adds very little overhead.

For both databases, a substantial amount of time is spent by the BaseEngine constructing the LocalStore transaction (in the `beginTX` call) and subsequently committing it (in the `commitTX` call). In addition, the Zelos `postApply` does a significant amount of work; these cycles are spent by the application to perform ZooKeeper-specific functionality such as triggering watches. In DelosTable, `postApply` does not show up since it is used mostly for lightweight activity such as updating in-memory caches.

The low overhead of log-structured protocols is further supported by our experience replacing ZooKeeper with

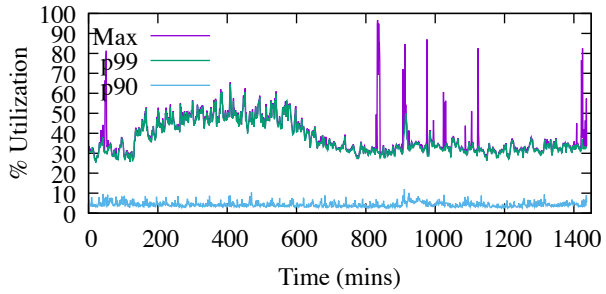

Figure 8: *Apply thread utilization across the fleet for a single day, measured over 1-minute periods: for each minute, we show the three clusters with the max / p99 / p90 utilizations. For any given minute, 90% of the clusters are below 10% apply utilization.*

Zelos. Zelos achieves performance parity with ZooKeeper on most of our production and benchmarking workloads. For example, on a mixed workload with 50% 100-byte writes (SetData) and 50% 100-byte reads (GetData), Zelos offers 56K/s operations compared to 36K/s from ZooKeeper on identical hardware. This speed-up cannot be attributed entirely to log-structured protocols (for example, Zelos is written in C++ rather than Java); however, it does provide evidence that a layered design does not hurt performance.

The apply thread is not the bottleneck. A single-threaded protocol stack is easy to operate and reason about; but does it hurt performance? To answer this question, we show the utilization of the apply thread across all DelosTable and Zelos production clusters. Figure 8 shows the clusters with the max / p99 / p90 utilizations for each minute over a single day. We see that max utilization rarely spikes higher than 60%. For any given minute, 90% of the clusters are below 10% apply utilization. There are two reasons for this. First, cluster workloads are typically dominated by read-only operations (which are not sequenced through the apply thread) rather than read-write transactions. Second, clusters that run at high write rates are bottlenecked by SSD bandwidth for the consensus protocol (which requires synchronous writes) rather than the apply thread. In our experiments, we typically only saturate the apply thread if we relax the durability of the consensus protocol (i.e., not flush to SSD); or if our writes are computationally expensive (e.g., compute some expensive predicate).

5.2 The Benefits of Layering

Log-structured protocols can optimize performance significantly. In Figure 9, we run a set of 5 clients against

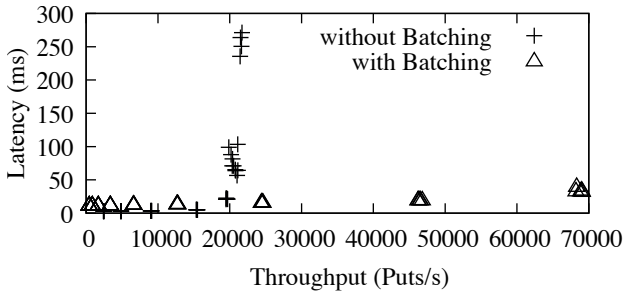

Figure 9: *The BatchingEngine provides a 2X increase in maximum throughput under 20ms p99 latency.*

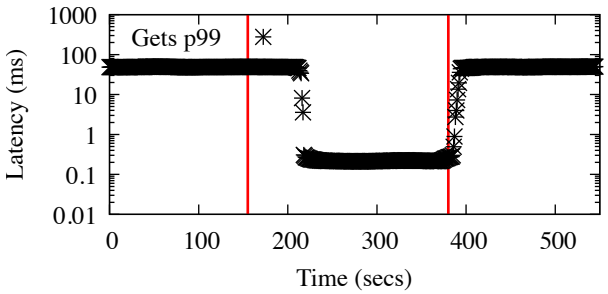

Figure 10: *When enabled, the LeaseEngine allows zero-coordination strongly consistent reads at the server holding a lease, lowering read latency by 100X for a deployment distributed across the continental USA.*

a 5-node cluster (within a single data center) at different write rates (for 100-byte writes), obtaining a set of throughput-latency points for DelosTable with and without the BatchingEngine. For a maximum acceptable p99 latency of 20ms, we get a 2X speed-up. We get similar results for 100-byte writes on Zelos on the same HW (not shown).

To show the LeaseEngine in action, we use a geo-distributed DelosTable cluster with 5 servers distributed across the continental USA (this is a common deployment mode for us when disaster tolerance is required). We collocate a client on one of the servers and measure the p99 read latency on a 1-minute sliding window. In the first part of Figure 10, the LeaseEngine is disabled; a strongly consistent read requires the server to reach out to a quorum (with a p99 latency of roughly 48ms). At T=155s, we enable the LeaseEngine by sending an admin command to the server, which turns on the LeaseEngine via a command in the shared log. Once the local server has a lease, strongly consistent reads do not require cross-region coordination and complete within 220 μs (which is reflected by the graph once the 1-minute sliding window

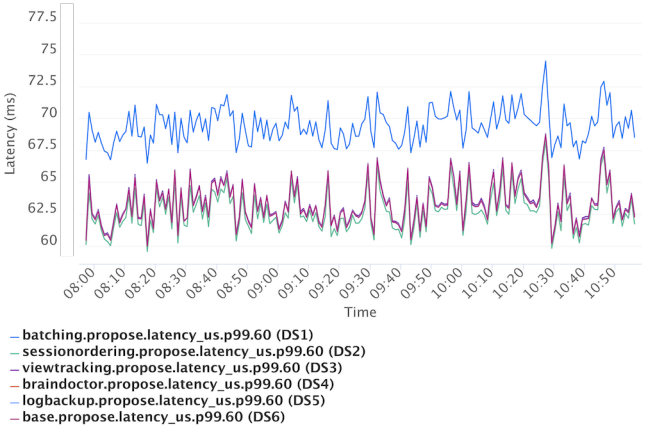

Figure 11: *Screenshot of ObserverEngine-enabled production dashboard for monitoring engines on a cluster.*

advances). At T=385s, we disable the LeaseEngine again; latency shoots back up to 48ms.

As this experiment shows, engines (and the optimizations they enable) can be dynamically turned on and off, ensuring correct behavior by synchronizing via the log; in the case of the LeaseEngine, the absence of such synchronization would result in consistency violations (e.g., if one server had the engine enabled and thought it had a lease; while another server did not have it enabled and ignored the lease).

Log-structured protocols enhance observability. An important benefit of engine layering is the ability to monitor – via the ObserverEngine – the performance of each layer in a generic way. Figure 11 is a screenshot from an actual production dashboard for a geo-distributed Zelos cluster, showing the 1-minute p99 latency of `propose` calls at each layer. As the dashboard shows, the BatchingEngine adds latency while accumulating a batch, while the other engines incur low latency. The lines in the dashboard for each engine are in stack order, with one notable exception: the lowest line in the graph is the SessionOrderEngine rather than the BaseEngine, due to the short-circuiting described in Section 4.3.

6 DISCUSSION

Why a stack? In most cases, engines are independent state machines that do not affect each other, and hence can be executed in parallel (for example, the `apply` logic for the ViewTrackingEngine and TimeEngine can execute simultaneously). Some engines do need to be ordered with respect to others (e.g., the SessionOrderedEngine filters entries from other engines). We considered organizing engines in a DAG; however, we found that a linear stack provides simplicity for developers (e.g., around

exception handling) and operators (e.g., each cluster is configured as a simple list of engines). Further, as we showed, we are not bottlenecked on the apply thread in our production clusters.

If we bottleneck on the apply thread in the future for different workloads and HW, we can execute engine `apply` logic in parallel for a single entry. We also plan to explore applying multiple entries from the shared log in parallel, if we know *a priori* that the entries have disjoint read/write sets on the LocalStore. Applying entries out-of-order requires ensuring prefix recoverability [40] (i.e., the LocalStore must always correspond to a prefix of the shared log). Another approach is to execute transactions speculatively within the propose thread before appending intentions to the log, and validate (rather than execute) within the apply thread [11, 12, 46].

Developer / Operator experience: New engineers on the project often found it difficult to reason about whether a module should be written as an engine. We came up with simple rules of thumb: A) the module must either be stateful (i.e., a replicated state machine), or B) control or observe what other engines and the application see (i.e., a protocol). Engineers also erred in the other direction, implementing reusable logic with tight application-coupling, but usually pivoted quickly after a round of code review. For example, the BatchingEngine was first implemented as a part of Zelos; but then cleanly layered out during code review so that integrating it with DelosTable was seamless.

A significant source of complexity for engineers was the lack of a clean protocol for dynamic updates. We initially did not provide such a protocol, expecting each engine to handle its own roll-out in custom ways. Ad-hoc stack updates resulted in inconsistency events across servers in production, which caused us to formalize the two-phase update protocol in Section 3.4.

Surprisingly, engine roll-out has been the only source of inconsistency in production so far, despite initial worries that developers would find it difficult to write deterministic code. We built extensive protection against this failure mode via incremental checksums of the LocalStore, but so far have encountered only events linked to ad-hoc stack updates. In the future, we may consider deterministic execution techniques [20, 21, 41] for safety.

The Delos Hourglass: Delos enables a three-tier architecture: multiple databases with different APIs running over a narrow waist of log-structured protocols (as described in this paper); and a virtualized shared log that supports multiple consensus protocols underneath (as described previously [9]). In recent months, we have implemented new databases rapidly in this architecture: for example, an undergraduate intern recently developed a new queue service called DelosQ over the summer; while one of our engineers built a new locking service in roughly two months. Importantly, these new services are nearly production-ready out of the box and do not impose much custom load on operators. All Delos databases have unified tooling for creating a new cluster, upgrading binaries, manually reconfiguring membership within a cluster, offline consistency / integrity checking, and recovering from backups.

7 RELATED WORK

A number of systems have proposed composing applications from collections of fine-grained replicated state machines [11, 45, 55]. Delos advances this state of the art by stacking replicated state machines, such that each one can filter or alter the entries applied upstream or proposed downstream. The log-structured protocol is a specific high-level programming abstraction above a shared log; prior systems have proposed other such abstractions, including fine-grained objects [11], composable data structures [55], and stateful serverless functions [33].

Log-structured protocols are inspired by classical research on layering in network stacks [17, 32, 48] and storage [28, 29, 35]. Stacking has been proposed before for replicated systems at a lower level of abstraction. For example, Delos itself layers log implementations via the VirtualLog abstraction [9]. Using Lamport's terminology [38], the log-structured protocols in Delos comprise a stack for learners (i.e., above the log abstraction) whereas the VirtualLog in Delos stacks acceptors (i.e., under the log abstraction). The difference is subtle but significant: any functionality that requires local materialized state (e.g., ViewTrackingEngine) or optimizes access to it (e.g., BatchingEngine via group commit) can only be implemented as a replicated state machine over the shared log, rather than within the shared log itself.

Log-structured protocols are directly inspired by replication systems from the 90s such as Horus [54], Ensemble [42], and BAST [26], which created modular, layered stacks for group communication. One difference from Delos relates to the target abstraction (SMR over a shared log vs. group communication). By substituting the shared log for the process group as the central abstraction, Delos extends ideas from the group communication literature (such as protocol layering) to a broader class of replicated systems. The underlying shared log can be implemented by any consensus protocol (including Raft [44] and MultiPaxos [52]). In addition, the interface between layers in Horus is rich: 16 types of downcalls and 14 types of upcalls [53]; as well as 16 protocol properties such as

causal delivery, best effort delivery, etc. that each layer either requires of the one below it or inherits / provides on its own. In contrast, Delos has an intentionally narrow API (`propose`/`sync` and `apply`), designed explicitly to lower the implementation burden for new engines, as well as simple semantics (i.e., the engine API is linearizable and durable, with no options for weaker semantics). In hindsight, these choices proved important for the success of Delos in a production setting, lowering complexity for developers and operators.

8 CONCLUSION

Log-structured protocols provide the benefits of State Machine Replication while enabling code reuse and consolidated operations across multiple databases. At Facebook, we composed nine log-structured protocols into two production databases with significantly different APIs, allowing engineers to interchangeably develop or operate either database. Other areas such as networks and filesystems have benefited hugely from layered design; the log-structured protocol stack gives us similar advantages by enabling simple, flexible, and reliable replicated databases. New log-structured protocols – and improvements or fixes to existing protocols – can immediately benefit multiple databases in production without any additional porting effort.

ACKNOWLEDGMENTS

We would like to thank the SOSP 2021 reviewers and our shepherd Fernando Pedone. We would like to thank all the engineers and interns on the ZooKeeper, DelosTable, and Delos teams at Facebook. In particular, Angela Chen, Aaryaman Sagar, Rhed Shi, Hazem Hassan, Nitin Muppalaneni, Kapil Agarwal, and David Devecsery provided valuable input for the ideas in this paper. Jose Faleiro, Xiao Shi, and Vijayan Prabhakaran provided feedback on drafts. Mack McCauley and Xianan Zhang managed the Delos team during the period when log-structured protocols were first introduced into the system. Maxim Khutornenko, Artemiy Kolesnikov, Denis Samoylov, and Ben Reed were instrumental in incubating the Delos project.

REFERENCES

[1] LogDevice. https://logdevice.io/.
[2] redis. https://redis.io/.
[3] Rocksdb. https://rocksdb.org/.
[4] Sqlite. https://www.sqlite.org/.
[5] Adya, A., Grandl, R., Myers, D., and Qin, H. Fast key-value stores: An idea whose time has come and gone. In *HotOS 2019*.
[6] Aguilera, M. K., Leners, J. B., and Walfish, M. Yesquel: Scalable sql storage for web applications. In *ACM SOSP 2015*.
[7] Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., van Hovell, H., Ionescu, A., Łuszczak, A., et al. Delta lake: high-performance acid table storage over cloud object stores. In *VLDB 2020*.
[8] Azagury, A., Factor, M. E., Satran, J., and Micka, W. Point-in-time copy: Yesterday, today and tomorrow. In *IEEE MSST 2002*.
[9] Balakrishnan, M., Flinn, J., Shen, C., Dharamshi, M., Jafri, A., Shi, X., Ghosh, S., Hassan, H., Sagar, A., Shi, R., et al. Virtual Consensus in Delos. In *USENIX OSDI 2020*.
[10] 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*.
[11] 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 *ACM SOSP 2013*.
[12] 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*.
[13] Bittman, D., Alvaro, P., Mehra, P., Long, D. D., and Miller, E. L. Twizzler: a Data-Centric OS for Non-Volatile Memory. In *USENIX ATC 2020*.
[14] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *USENIX OSDI 2006*.
[15] Cao, Z., Dong, S., Vemuri, S., and Du, D. H. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *USENIX FAST 2020*.
[16] Chrysafis, C., Collins, B., Dugas, S., Dunkelberger, J., Ehsan, M., Gray, S., Grieser, A., Herrnstadt, O., Lev-Ari, K., Lin, T., McMahon, M., Schiefer, N., and Shraer, A. FoundationDB record layer: A Multi-Tenant Structured Datastore. In *ACM SIGMOD 2019*.
[17] Clark, D. D. The structuring of systems using upcalls. In *ACM SOSP 1985*.
[18] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. In *ACM ASPLOS 2011*.
[19] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013), 1–22.
[20] Cui, H., Gu, R., Liu, C., Chen, T., and Yang, J. PAXOS Made Transparent. In *ACM SOSP 2015*.
[21] Cui, H., Simsa, J., Lin, Y.-H., Li, H., Blum, B., Xu, X., Yang, J., Gibson, G. A., and Bryant, R. E. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), pp. 388–405.
[22] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D. A. Implementation techniques for main memory database systems. In *ACM SIGMOD 1984*.
[23] 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 *USENIX NSDI 2020*.

[24] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. FaRM: Fast Remote Memory. In *USENIX NSDI 2014*.

[25] Friedman, M., Herlihy, M., Marathe, V., and Petrank, E. A Persistent Lock-Free Queue for Non-Volatile Memory. *ACM SIGPLAN Notices 53*, 1 (2018), 28–40.

[26] Garbinato, B., and Guerraoui, R. Flexible protocol composition in BAST. In *ICDCS 1998*.

[27] Gray, C., and Cheriton, D. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *ACM SOSP 1989*.

[28] Guy, R. G., Heidemann, J. S., Mak, W., Page Jr, T. W., Popek, G. J., and Rothmeier, D. Implementation of the Ficus Replicated File System. In *USENIX Summer 1990*.

[29] Heidemann, J. S., and Popek, G. J. File-System Development with Stackable Layers. *ACM Transactions on Computer Systems (TOCS) 12*, 1 (1994), 58–89.

[30] 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.

[31] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC 2010*.

[32] Hutchinson, N. C., and Peterson, L. L. The x-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software engineering 17*, 1 (1991), 64.

[33] Jia, Z., and Witchel, E. Boki: Stateful Serverless Computing with Shared Logs. In *ACM SOSP 2021*.

[34] Junqueira, F. P., Reed, B. C., and Serafini, M. Zab: High-performance broadcast for primary-backup systems. In *IEEE DSN 2011*.

[35] Khalidi, Y. A., and Nelson, M. N. Extensible File Systems in Spring. In *ACM SOSP 1993*.

[36] Kogias, M., and Bugnion, E. HovercRaft: Achieving Scalability and Fault-tolerance for microsecond-scale Datacenter Services. In *ACM EuroSys 2020*.

[37] Kulkarni, C., Moore, S., Naqvi, M., Zhang, T., Ricci, R., and Stutsman, R. Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage. In *USENIX OSDI 2018*.

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

[39] Lee, C., Park, S. J., Kejriwal, A., Matsushita, S., and Ousterhout, J. Implementing Linearizability at Large Scale and Low Latency. In *ACM SOSP 2015*.

[40] Li, T., Chandramouli, B., Faleiro, J. M., Madden, S., and Kossmann, D. Asynchronous Prefix Recoverability for Fast Distributed Stores. In *ACM SIGMOD 2021*.

[41] Liu, T., Curtsinger, C., and Berger, E. D. Dthreads: efficient deterministic multithreading. In *ACM SOSP 2011*.

[42] Liu, X., Kreitz, C., van Renesse, R., Hickey, J., Hayden, M., Birman, K., and Constable, R. Building Reliable, High-Performance Communication Systems from Components. In *ACM SOSP 1999*.

[43] 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 *ACM EuroSys 2006*.

[44] Ongaro, D., and Ousterhout, J. K. In Search of an Understandable Consensus Algorithm. In *USENIX ATC 2014*.

[45] Ostrowski, K., Birman, K., Dolev, D., and Ahnn, J. H. Programming with Live Distributed Objects. In *ECOOP 2008*.

[46] Pedone, F., Guerraoui, R., and Schiper, A. The Database State Machine Approach. *Distributed and Parallel Databases 14*, 1 (2003), 71–98.

[47] Peng, D., and Dabek, F. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *USENIX OSDI 2010*.

[48] Ritchie, D. M. The UNIX System: A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal 63*, 8 (1984), 1897–1910.

[49] Schneider, F. B. Implementing Fault-tolerant Services using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR) 22*, 4 (1990), 299–319.

[50] Shute, J., Vingralek, R., Samwel, B., Handy, B., Whipkey, C., Rollins, E., Littlefield, M. O. K., Menestrina, D., Cieslewicz, S. E. J., Rae, I., Stancescu, T., and Apte, H. F1: A Distributed SQL Database That Scales. In *VLDB 2013*.

[51] 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 *USENIX OSDI 2020*.

[52] Van Renesse, R., and Altinbuken, D. Paxos Made Moderately Complex. *ACM Computing Surveys (CSUR) 47*, 3 (2015), 1–36.

[53] van Renesse, R., Birman, K. P., Friedman, R., Hayden, M., and Karr, D. A. A Framework for Protocol Composition in Horus. In *ACM PODC 1995*.

[54] Van Renesse, R., Birman, K. P., and Maffeis, S. Horus: A Flexible Group Communication System. *Communications of the ACM 39*, 4 (1996), 76–83.

[55] 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*.

[56] You, J., Wu, J., Jin, X., and Chowdhury, M. Ship Compute or Ship Data? Why Not Both? In *USENIX NSDI 2021*.

[57] Zhang, W., Shenker, S., and Zhang, I. Persistent state machines for recoverable in-memory storage systems with nvram. In *USENIX OSDI 2020*.

[58] Zimmermann, H. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications 28*, 4 (1980), 425–432.