Taming Consensus in the Wild (with the Shared Log Abstraction)

Mahesh Balakrishnan
Confluent, Inc.

Abstract

The shared log is an abstraction for building layered consensus systems that are simple to develop, deploy, evolve, and operate. Shared logs emerged from systems research and have seen significant traction in industry over the past decade. In this paper, we describe some design principles for consensus-based systems, based on our experience building and operating real-world shared log databases in the wild.

1 Introduction

Consensus-based systems form the foundation of today's computing platforms, enabling ubiquitous, always-on services that can store and process data with strong guarantees around durability and availability. Such systems are notoriously difficult to design, build, operate, and evolve in the face of complex failure patterns and arbitrary asynchrony. In practice, deployed systems such as ZooKeeper [10], Raft [16], and etcd [2] are complex, monolithic codebases consisting of multiple intertwined protocols – e.g., single-slot consensus, multi-slot ordering, membership, materialization, etc. – with each node playing different roles within each protocol. This complexity translates to reduced reliability and code velocity.

In other types of systems, abstraction has proved to be a powerful tool for reducing and corralling complexity; for example, OS abstractions such as processes, address spaces, and block devices hide the complexity and diversity of hardware; whereas networking abstractions such as protocol layers enable us to independently solve problems such as routing and reliable communication. Can similar abstractions help us simplify consensus-based systems?

The shared log abstraction provides one answer. A shared log is an append-only address space that is fault-tolerant and shared – i.e., multiple clients can append to it and read from it concurrently with strong semantics. The shared log approach to consensus appeared in its modern form in the Hyder [7] and Corfu [5] systems; subsequently, it generated a number of follow-up research systems [9, 11, 14, 15, 17, 20]; and has

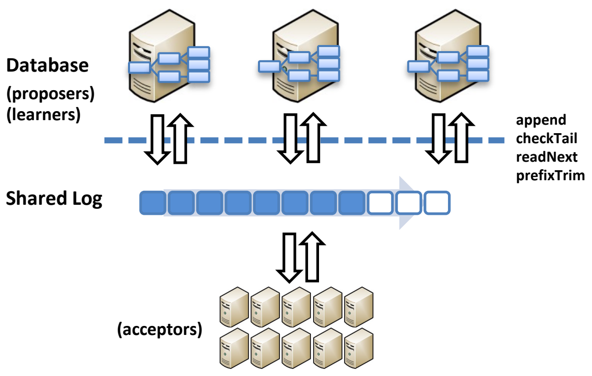

Figure 1: The Shared Log Approach

been directly deployed at massive scale within a number of different production systems [1, 3, 4], while influencing a number of related systems [12, 13, 19].

In this paper, we reflect on a decade of real-world experience with a specific lineage of systems: two research prototypes called Corfu [5] and Tango [6]; and a production system called Delos [4]. We describe how the shared log abstraction sheds light on the placement of functionality within a consensus-based system; and list a number of principles gleaned from developing and deploying these systems in production. Though these principles are most easily explained in the context of a shared log database, we expect them to generalize to any consensus-based system.

2 A Tale of Two Layers: the Database and the Log

The vast majority of consensus-based systems are designed over the State Machine Replication (SMR) abstraction [18]. In SMR, an arbitrary black box object (for example, say an implementation of a counter) is executed redundantly on multiple nodes, with each node maintaining a full copy of the object's state. Each mutating command (e.g., an increment on the counter) is first sequenced in a global, durable total order of commands; and then applied to the black box on each node. To serve an accessing command (e.g., a get on the counter) on a particular node, we first synchronize the local copy of the object with the global, durable total order; and then execute the command on the copy.

Most SMR frameworks provide a simple API that allows any black box object to be wrapped and transparently replicated. In the example above, let us say the interface of the object is an AbstractCounter; and its black box implementation is a LocalCounter. To use SMR, we can write a Wrapping-Counter that implements the AbstractCounter interface; any external application can then interact with a WrappingCounter and obtain linearizable semantics. When the increment call is invoked on the WrappingCounter, it first `proposes` the unexecuted command to the SMR framework; which stores the data on a global, durable total order; and then in turn `applies` the command to the application in that total order.

Note that the SMR abstraction fundamentally divides the system into two layers: the *database*, which is a set of nodes executing copies of the black box object, as well as the wrapping logic to propose commands and apply them back to those copies; and the *log*, which is a set of nodes storing the global total order. The database is effectively a materialization of the log. Mapping this to Lamport's terminology for the theory of consensus, we use the terms *proposer* or *learner* to refer to the database layer; and the term *acceptor* to refer to the log layer.

Prior to the shared log abstraction, this clean layering existed only in the theory of consensus. Existing practical systems did not explicitly separate these two layers; typically, a database (either a standalone instance or a shard in a partitioned system) would be replicated on a set of three or five nodes, each of which operated as both a learner and an acceptor. In effect, the SMR abstraction is implemented in such systems in a monolithic way, mixing the logic of forming and storing the total order (i.e., the log) with the code for materializing the total order (i.e., the database).

The shared log abstraction splits the SMR platform into two, separating proposers/learners from acceptors via a simple API. Clients can *append* entries, obtaining a log position (which is effectively a logical timestamp); invoke *checkTail* to obtain the first unwritten log position at the tail of the log; *read* the first entry in a supplied range of log positions; and *trim* a prefix of the log.

As an analogy, think of the SMR platform as a filesystem and the shared log as a block device. In much the same way that a block device makes it easier to build a filesystem without worrying about hardware internals (e.g., HDD vs. SSD), a shared log helps us write an SMR layer without reasoning about the internals of the consensus protocol. The SMR layer is then free to focus on the complexity of materialization, snapshot management, query scalability, single-node failure atomicity, etc., in much the same way that a filesystem can focus on file-grain multiplexing, directories, crash consistency, etc. When we roll out a new consensus protocol, we do not need to rewrite the SMR layer, in much the same way that we do not need rewrite a filesystem just to support a new type of block device. Finally, the SMR layer and the shared log can reside on entirely different sets of machines, in much the same way that a filesystem can operate over remote block storage.

Who stores what in the Shared Log? In classical SMR, any database replica (or proposer) can propose a command. The SMR abstraction is explicitly designed to support multi-master operation. There is no notion of leadership above the SMR abstraction. The shared log consists of unexecuted inputs – effectively, arbitrary pieces of code – that are executed deterministically by each database replica on playback. To use a concrete example, if the application is a simple counter, then the log entries can literally consist of "increment by 10" or "decrement by 5".

However, in some cases, such redundant execution can be overly expensive. For example, if the command is "scan the DB and sum all entries", it is wasteful to have each database replica execute the full scan. As a result, systems will often execute the command at a particular database replica and then propose an output to the SMR layer. Note that the SMR layer is oblivious to whether an input or an output has been proposed; in either case, it stores the command in the shared log and then applies it to each database replica. To use the example of a counter, log entries can be "set to 55".

When outputs are stored in the shared log, it is no longer safe to directly apply each entry from the shared log to local state. The proposing database replica generated the log entry based on some locally materialized snapshot of database state (corresponding to prefix [0, X) of the log); but by the time it appends the entry, the log might have grown, in which case the new entry lands at some position Y, such that there are multiple intervening entries generated by other proposers between X and Y. The generated output in entry Y can only be applied to the database if the intervening entries did not invalidate it (e.g., by modifying some key that was used to generate Y). In the counter example, two database replicas might read the current counter state as 54; append commands "set to 55" at the same time in consecutive log slots. In this case, we accept the first entry but reject the second entry.

3 Principles for Consensus-Based Systems

A natural consequence of building a layered system is that it becomes possible to characterize the scaling, performance, and reliability properties of each layer separately; as well as reason about the placement of functionality in each layer. As we developed and deployed these layered systems in production, we noted a number of principles that seemed generally true for any consensus-based system. We now describe these.

3.1 Scalability

A: Acceptors can be scaled out via sharding. The shared log hides the complexity of the acceptor role behind a simple, data-centric API. In conventional consensus systems where each node acts as a proposer / learner / acceptor (or equivalently, as both a database and a log replica), we do not have the ability to separately scale each role. However, in shared log systems we can disaggregate the acceptors from the proposers / learners, which in turn allows us to scale the acceptor layer separately (by scaling the shared log).

To scale the acceptor role, we observe that a shared log is simply an append-only address space. As with any other address space, we can scale log-reads linearly by striping the shared log over different sets of nodes. To scale appends, we have multiple options: we can have an off-path sequencer which does not see I/O and can consequently run at millions of appends / sec (e.g., see Corfu [5]); or we can rotate an I/O-bound sequencer role over a set of machines; or use deterministic merge to stitch together the order generated by multiple separate sequencers.

Scaling acceptors is particularly useful when acceptor storage is slow relative to other parts of the system. For example, if the database is an in-memory data structure and the shared log runs on slow drives (e.g., older SATA SSDs), we can decouple the two layers and run a small number of database nodes against a larger set of acceptors.

B: Learner playback limits (write) scale. Ultimately, a shared log is a broadcast medium: every database replica (or learner) has to see every log entry. As a result, even if we scale the acceptor layer to millions of log-reads (by sharding acceptors) and appends / sec (by using an off-path sequencer), the write throughput of the database is still limited by the ability of each database node to ingest entries from the shared log and apply them to its local state.

In simple designs, all playback happens on a single thread; if this is the case, then the entire database is typically bottlenecked on a single pegged "apply thread" on each database replica. For example, if we can process 50K transactions per second on that core, then that's the total write speed of the database, regardless of the number of learners or acceptors we use. Common techniques from the database and SMR literature can be used to optimize the apply thread. For example, a group commit optimization can drive up the throughput of the single "apply thread" at the cost of higher latency. Alternatively, we can parallelize playback for non-conflicting transactions on multiple cores and allow them to commit out of order, which adds complexity in the recovery path since the state at the database replica no longer matches an exact prefix of the shared log.

Even if we parallelize playback to the point where we are utilizing all cores on the database replica, we are still limited by the ingress bandwidth on each database replica. Having flexibility in whether we store inputs vs. outputs in the shared log can help balance ingress bandwidth against CPU overhead on the database replicas. For example, consider a command that scans the entire database to delete all keys with odd values; this can be expressed as a very compact input – literally a lambda – in the shared log that triggers significant computation (and local I/O) on each learner; or as a potentially large output on the shared log (a list of keys that match the predicate) triggering very little computation on each learner.

Once we are pegged either on learner CPU or ingress bandwidth, the only avenue to scale the system for writes is to shard learners, as proposed by Tango [6]: i.e., the state of the database is partitioned into shards (e.g., blue, red, and green); and a blue replica only stores the blue shard and materializes only the blue entries from the shared log. In practice, the key blocker for such a design proved to be blast radius: e.g., if we lose access to a single slot in the shared log after it is written (but before it is played), every learner across all shards will have to block processing at that entry until we can resolve the problem.

C: Strongly consistent reads can be scaled linearly by adding learners. We can scale read throughput in a shared log database simply by adding more learners. As described in Section 2, each learner serves reads (which can be arbitrary accessors, including read-only queries on a SQL database or get operations on a key-value store) by first synchronizing with the shared log, which involves a `checkTail` invocation. This protocol ensures strong consistency even in a multi-master setup where any database replica can propose new commands to the shared log. Unfortunately, invoking checkTail on the underlying shared log is an expensive operation, requiring us to either go to a sequencer or some quorum of acceptors (or a quorum on each acceptor shard if we shard acceptors).

To mitigate this problem, a number of shared log databases (including Delos) use a "bus-stand" optimization where multiple reads can queue up behind the next `checkTail`. In the unoptimized case, each read triggers an individual `checkTail`; this is similar to a passenger catching a cab from a source (the learner) to a destination (the shared log) and back again. With the optimization, we allow a single outstanding `checkTail` invocation to the shared log; this is similar to a bus shuttling back and forth between the source and destination. Incoming reads have to wait for the bus to return to the station before they can board it. In practice, this means each read waits for 2 full checkTail invocations – e.g., 2 RTTs to some sequencer – in the worst case (i.e., if it arrived just after the bus – the checkTail invocation – left the station).

3.2 Latency

D. Log-based Leases can support zero-coordination reads.

Some systems require low-latency strongly consistent reads with zero coordination (i.e., served by a single node). We can achieve this in a shared log database by electing a particular database replica as a designated proposer; all writes and reads are directed to this replica. The election of this designated proposer can happen above the log itself, by appending a special election command with the semantics that players must reject all subsequent entries not generated by the specified proposer (until the next election command). If we also assign a timeout to the election command's validity, we can enable low-latency strongly consistent reads at the designated proposer. Note that the timeout can be in real-time or in logical time defined by log positions; in the latter case, rather than wait out time, we can burn log positions via dummy entries to evict the old designated proposer.

In other consensus-based systems [8], a single, strong master or leader role is viewed as key to enabling zero-coordination reads. However, a shared log design makes it obvious that leadership *above the log* – which is required for zero-coordination reads – is distinct from leadership *below the log*: we can enable zero-coordination reads even in systems that do not necessarily have a conventional leader role below the log (e.g., out-of-path sequencers).

Note that zero-coordination reads are possible at any database replica if we are okay sacrificing consistency; in a shared log database, the application can simply access the local materialized state without syncing with the shared log. This is a one-line change in shared log databases and routinely used to support stale snapshot reads.

E: Acceptors and Learners can be isolated from each other to prevent I/O interference. In cases where the database resides on durable storage (e.g., local materialized state is stored in RocksDB or SQLite), decoupling learners from acceptors (either on different drives on the same machines; or entirely different tiers) can isolate I/O between the two. Acceptor latency is critical for shared log databases, since a missing/slow log entry can stall the entire database. Learners can interfere with acceptors in at least two ways. First, the potentially random access patterns of the learner can interfere with the sequential writes on the acceptors. Second, learner activity can trigger filesystem interactions requiring locks to be held (e.g., when we delete snapshots in RocksDB); which can combine with I/O stalls (often lasting tens of seconds) on older SSDs to delay log writes. Separating learners from acceptors alleviates both these problems; in addition, we can also use cheaper drives for the learners compared to the acceptors, which require a relatively small amount of fast storage (since the shared log is continuously being trimmed or moved to backup storage).

F: No single acceptor needs to be in the critical path of a write: In protocols such as Raft [16], the "leader" is a designated proposer; but also a required acceptor in the write quorum. This requirement reduces the effectiveness of quorums in lowering tail latency, particularly since the leader is likely to be a heavily loaded node.

In a shared log design, the acceptor quorum is hidden entirely behind the shared log API: any notion of leadership within the log is entirely separate from the concept of a designated proposer above the log (though the two can be collocated for performance). As a result, we do not require a particular acceptor to be in the critical path of writes.

3.3 Reliability

G. Proposers / Learners and Acceptors have different reliability requirements. For durability, we need just one database replica to survive; this is sufficient to ensure the durability of all commands played by that replica. Similarly, availability is ensured as long as one database replica is still alive and can access the shared log. In contrast, the shared log layer has more onerous requirements for reliability: we need a quorum of log replicas for durability; and a quorum of log replicas as well as the sequencing mechanism for availability.

Practically, separating the database and the log layer allows engineers to respond with different degrees of urgency to failures in each layer. A single crashed database replica is not particularly problematic in a 3-node deployment; however, a single crashed log replica is quite dangerous in a 3-node deployment, since a single additional failure can result in the loss of an acknowledged write.

To restate this in consensus terminology: learners have less stringent requirements for fault-tolerance than acceptors. As a result, disaggregating them into separate layers can result in systems that are easier to operate. Further, we can bootstrap the database by using some existing storage system like a key-value store as a slow, inefficient shared log; allowing us to quickly roll out a functional first version of the database while developing the acceptor logic over a longer period of time, as we did for Delos.

H. Proposers / Learners and Acceptors have different reliability characteristics due to code velocity. In production environments, a key factor impacting the reliability of a service is code velocity: failures are often linked to rollouts of new code. Anecdotally, a common operational scenario for Delos in converged mode (i.e., where the database and the log are collocated) involved multiple nodes crash-looping in the middle of a rollout.

In a shared log system, the database above the log is modified quite frequently to add new features and query optimizations; whereas the log itself is much less frequently updated, since its functionality does not evolve much over time. In consensus terms: learners are upgraded more frequently than acceptors; and hence more likely to fail. Even worse, when the learner and acceptor roles are collocated, the relatively common and benign failure of a learner is converted into a serious acceptor failure.

By decoupling the database and the log into separate services, we ensure that the service that's updated more frequently (and hence subject to more risk) is the one that has a less strict reliability requirement; and isolated from the service that has a more strict reliability requirement.

I. The Scheduler decides who lives and dies. Consensus-based systems often include sophisticated failure-detection primitives (e.g., based on gossip). In cloud environments, such systems are deployed via schedulers such as Kubernetes and Twine. As a result, it can be dangerous for systems to rely on their own internal mechanisms. In one particular Delos incident, Twine lost contact with all Delos nodes in a cluster and marked them as offline; but Delos itself reported good health based on an internal gossip mechanism. In such a state when the scheduler's view of node health does not match reality (as observed by the system's own mechanisms), the danger is that the scheduler has the ability to take unilateral actions (e.g., revoking a quorum of nodes) based on its assessment. In Delos, we found that failure detection for membership was best handled by the scheduler (via an interface that let any new node know which other node it was replacing), while we used various internal protocols for different types of leadership above and below the log.

4 Conclusion

The shared log is an abstraction for consensus; as such, its primary value lies in its ability to simplify the development and deployment of consensus-based systems by hiding the complexity of asynchrony, failures, and code velocity. However, a side-effect of abstraction is a greater understanding of how different types of functionality should be distributed in a system and interact with each other. In this paper, we presented a number of insights and principles obtained over multiple years of building and operating real-world shared log databases. We believe that these principles generalize to any consensus-based system and can simplify other types of replicated databases.

Acknowledgments

The observations in this paper are based on years of collaboration with many researchers and engineers. Dahlia Malkhi and I started the Corfu project at Microsoft Research; the project was particularly influenced by Vijayan Prabhakaran, Ted Wobber, and Michael Wei. Delos was started by a small team that included Chen Shen and Mihir Dharamshi; Jason Flinn joined soon after; and the project later expanded to a large team of engineers at Meta. These ideas have been influenced and refined by conversations with Lucas Bradstreet and Gardner Vickers at Confluent. Finally, the shared log abstraction is influenced heavily by ideas from Ken Birman and Robbert van Renesse.

References

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

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

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

[4] 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*.

[5] 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*.

[6] 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*.

[7] 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*.

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

[9] 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*.

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

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

[12] Junqueira, F. P., Kelly, I., and Reed, B. Durability with bookkeeper. *ACM SIGOPS OSR 47*, 1 (2013), 9–15.

[13] Kleppmann, M., and Kreps, J. Kafka, Samza and the Unix Philosophy of Distributed Data. *IEEE Data Engineering Bulletin, 38 (4)* (2015).

[14] 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*.

[15] 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*.

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

[17] QI, S., LIU, X., AND JIN, X. Halfmoon: Log-optimal fault-tolerant stateful serverless computing. In *ACM SOSP 2023*.

[18] SCHNEIDER, F. B. The state machine approach: A tutorial. In *Fault-tolerant distributed computing* (1990), Springer, pp. 18–41.

[19] 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*.

[20] 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*.