Designing a Datacenter-wide Distributed Shared Log

Micah Murray
micahmurray@berkeley.edu
UC Berkeley
Berkeley, CA, USA

Wen Zhang
zhangwen@cs.berkeley.edu
UC Berkeley
Berkeley, CA, USA

Aisha Mushtaq
aisha@berkeley.edu
UC Berkeley
Berkeley, CA, USA

Natacha Crooks
ncrooks@berkeley.edu
UC Berkeley
Berkeley, CA, USA

Aurojit Panda
apanda@cs.nyu.edu
New York University
New York, NY, USA

Scott Shenker
shenker@icsi.berkeley.edu
UC Berkeley & ICSI
Berkeley, CA, USA

Abstract

Distributed shared logs simplify the implementation and interoperation of data stores. This paper addresses a simple question: Is it feasible to build a single, datacenter-wide distributed shared log that can support all the data stores running in a datacenter? We answer in the affirmative by presenting RingWorld, a scalable log based on a ring of programmable switches that can sustain tens of billions of appends per second while maintaining low latency. We hope the design of RingWorld will propel the adoption of shared logs as a core part of datacenter infrastructure.

CCS Concepts

• **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Networks** → **In-network processing**; **Data center networks**.

Keywords

Distributed shared logs, programmable switches, datacenter

ACM Reference Format:
Micah Murray, Wen Zhang, Aisha Mushtaq, Natacha Crooks, Aurojit Panda, and Scott Shenker. 2025. Designing a Datacenter-wide Distributed Shared Log. In *Workshop in Hot Topics in Operating Systems (HotOS 25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3713082.3730394

HotOS 25, Banff, AB, Canada

ACM ISBN 979-8-4007-1475-7/25/05
https://doi.org/10.1145/3713082.3730394

1 Introduction

Today's large-scale applications rely on distributed data stores [12, 14, 24] to store and serve data with high performance and consistency, availability, and durability guarantees. Unfortunately, these data stores are notoriously complex to implement and operate. This complexity comes from two main sources:

Replication For fault tolerance and scalability, a data store must spread each piece of data on multiple nodes (replicas), and in doing so, must provide some degree of consistency across its replicas. Ensuring consistency in the face of asynchrony and failures usually requires implementing and operating a distributed consensus protocol, both nontrivial tasks.

Interoperation Applications are commonly backed by *multiple* such replicated data stores that frequently interoperate. An application may *shard* its large-scale data across many data stores; it may even store different types of data in *heterogeneous* data stores, each providing different API and functionality. In both cases, business logic may rely on transactions across data stores, which may in turn require implementing an atomic commit protocol, adding to the complexity.

To tame this complexity, the systems community has proposed *distributed shared logs* [3] as a general primitive that takes care of these complexities for data stores built on top (§2.1). They provide the abstraction of an always-on, totally ordered, durable log that can be concurrently accessed by multiple servers; data-store replicas can then be implemented simply as in-memory clients of the log. Furthermore, multiple data stores implemented using the log abstraction can be operated at run time on *one single shared log*, greatly simplifying the implementation of cross-store operations.

When a shared log is deployed as a common substrate for many data stores, it operates in a high-throughput regime and so our primary concern is scalability: The log must be able to sustain the throughput of all data operations imposed by the data stores on top. Unlike for a *metadata service* (like

Chubby [10]), which lies on the control path, our shared log is on the data stores' *data path* with much higher load. As a secondary concern, log operations must have reasonably low latency. But given the high throughput requirement, we do not aim for—and cannot feasibly achieve—ultra-low latency at the microsecond level [26].

In this paper we ask the question: Can we implement a distributed shared log that scales to an *entire datacenter*? Such a log must field requests generated by as many data stores as can fit in a single data center; this can amount to an (ambitiously) estimated load of *80 billion appends per second* (§2.2). Existing shared-log systems cannot sustain such high load (§2.3): They either are not designed to scale to high throughput (e.g., Corfu [4] and LazyLog [26]), or suffer enormous latency at this scale (e.g., Scalog [17]).

And yet we believe that datacenter-scale shared logs are achievable. To demonstrate this, we propose a design called RingWorld that can scale to an entire datacenter while maintaining single-digit millisecond latency. RingWorld assumes that the top-of-rack (ToR) switches in the datacenter are *programmable switches* and leverages them to scale up sequencing, which is the throughput bottleneck in a classic shared-log design [4]. Specifically, in RingWorld the ToR switches are connected in a ring and programmed to collectively implement sequencing, distributing load among the switches using a moving-sequencer scheme [15, § 4.2]. While other components of a log implementation—such as broadcast and acknowledgment counting—are not throughput-critical, they can also be offloaded from servers to the ToR switches to (1) improve fault tolerance, given that switches are more reliable than servers [18], and (2) reduce the CPU load on servers and increase per-server throughput [35].

Given the feasibility of this design, our long-term vision is for a datacenter to provide a shared log *as part of its core infrastructure*—just like how datacenters today provide efficient, reliable data transmission via high-speed networks and transport protocols with carefully designed congestion control. Developers would build data stores using the log API without worrying about consistency, durability, or fault tolerance (this is already happening [6]); operators would deploy the stores on the shared-log infrastructure, allowing them to interoperate with good performance. The scale we are targeting may appear absurdly high by today's metrics, but today's datacenter scales would have been unthinkable two decades ago, so we think it likely that the scalable log, once built, will be put to good use.

2 Log and Scalability

2.1 A Primer on Distributed Shared Logs

The distributed shared log is both a versatile API that enables simple data-store implementations, and a unified platform that enables interoperation among multiple stores.

2.1.1 Log Abstraction. The log provides this basic API:

append Takes a value as argument, appends it to the end of the log, and returns the value's index in the log.

read Takes an index and returns the value stored at that index (or an error if the index is out of bounds).

getTail Takes no argument and returns the largest log index at which a value has been appended.

A distributed log provides linearizability [19] for these operations with respect to the obvious sequential specification.[1]

A full-fledged log provides other important features: a **trim** operation for garbage collection, and the ability to form *streams* so that different portions of the log can be consumed concurrently [5, § 5]. We will not be discussing these features as they are less relevant to scaling.

This simple API already allows implementing a data store. A *data-store server* serves requests from clients but itself acts as a client of the log, maintaining a (partial) in-memory view of the log's content. Typically, it performs a data-store update by appending to the log an entry that captures the update (e.g., for a transaction, read- and write-sets and data modifications) and, if necessary, playing the log forward until that entry to determine if the update succeeded (e.g., by checking for transaction conflicts) [5, § 3.2]. It serves a data-store read from its in-memory view after playing the log until the tail (or not, if the data store allows stale reads).

The log API is versatile: On top, people have designed data stores with different interfaces (key-value store [4, § 4], table store [6], serverless logbooks [21, 28]) and guarantees (snapshot and serializable isolation [8], session ordering [6, § 4.3]). It also hides the many complexities that plague distributed systems: Data-store servers need not implement consensus or atomic commit, maintain persistent state, or even directly communicate with one another (although they are still responsible for concurrency control, which is data-store-dependent).[2]

2.1.2 Log Platform. Once implemented using the log API, multiple data stores—no matter their data models or client-facing APIs—can be operated at run time *on a single log*. This log platform ensures data is persisted and sufficiently replicated, and allows heterogeneous data stores to share

[1] An exception: Some implementations allow "junk" to spuriously appear in the log under (suspected) failures [4]; junk entries are harmless.

[2] As an optimization, data-store servers may directly disseminate data to one another (e.g., see Hyder [9, § 1.4] and Aurora [32, § 3.2]), but such optimizations are typically easy to implement.

operating teams and maintenance procedures, reducing the cost of operation [6]. But more powerfully, log entries from different stores are now ordered with respect to each other. This makes it almost trivial to implement strongly consistent operations across data stores (shards) such as transactions, consistent snapshots, coordinated rollback, and consistent remote mirroring [5, § 3.2]. Such cross-store operations are needed by applications in the wild: For example, transactions across heterogeneous data stores appear in web services [29, § 3.1.3], cloud-based applications [16], microservice deployments [22].

Granted, instituting a single total order across all updates may not be the most *efficient* way to implement a particular cross-store operation—specialized solutions can be faster. But the log's value lies in its *generality* as a substrate enabling the interoperation of many heterogeneous data stores, both ones that exist today and ones invented tomorrow. So in the rest of this paper, we will focus on scaling a distributed shared log that is operated as such a substrate.

2.2 Target: Datacenter Scale

Given our aim to build a shared log that "scales to an entire datacenter", we now estimate the scale of the load such a log must sustain. Since we do not operate a datacenter, we will base our estimates on publicly available information, erring on the side of ambitious estimates when in doubt: We want to explore how high we can push the log's throughput while still being practically feasible.

A datacenter today comprises a collection of *racks*, each comprising a number of servers directly connected to a ToR switch; the ToR switches themselves are connected via some network topology. We assume that each rack consists of 40 servers [7] and that each pod in the datacenter has 1000 racks (the scale of Juniper data center networks [30]), totaling 40 000 servers in the datacenter[3]. We further assume that the log is served by half of the servers—these 20 000 *log servers* persist log entries and serve simple read/write requests for individual entries (in the style of Corfu's storage units [4]). The remaining servers execute data-store logic and serve as clients of the log.

An ideal log design must be able to sustain a workload that *saturates all the log servers*. To estimate such a workload, we assume that log entries are 1 KiB and that a log server can serve 24 million read/write requests per second for such entries;[4] this amounts to a *total log-server throughput* of $2.4 \times 10^7 \times 20\,000 = 4.8 \times 10^{11}$ (480 billion) requests per second. We assume that each log entry is replicated three ways, and so each log **append** translates to three requests to log servers and each log **read**, one request.

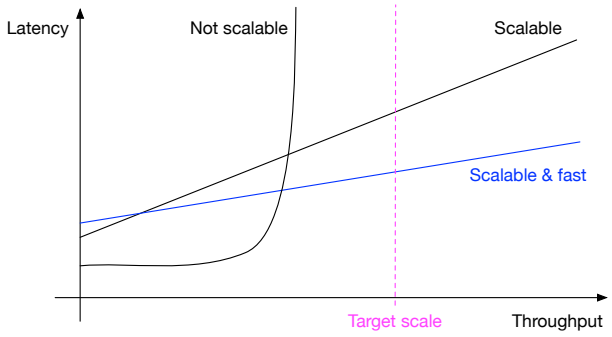

Figure 1: An illustration of latency-throughput curves for sequencing. Each point on a curve represents a system configuration that achieves the lowest latency for a given throughput, assuming unbounded resources.

Consider a log workload consisting of 25 % **append**s and 75 % **read**s. Let T_A denote the **append** throughput and T_R the read throughput *when all log servers are saturated*. Since **append** is the operation that makes scaling a challenge, we will solve for T_A:

$$T_A : T_R = 1 : 3, \qquad \text{(Workload mix)}$$

$$3T_A + T_R = 4.8 \times 10^{11}\ \text{s}^{-1}. \qquad \text{(Total log-server throughput)}$$

This yields $T_A = 8 \times 10^{10}\ \text{s}^{-1}$; that is, we will aim for a single-datacenter log that can sustain *80 billion appends per second*.

To be clear, this scale is enormous and possibly unprecedented: No prior design or deployment of distributed shared logs to our knowledge has been attempted at this magnitude. We pursue this ambitious goal driven not only by intellectual curiosity, but also by its potential to transform the way data stores are designed and operated within datacenters.

2.3 How (Scalable) Logs Are Built

Before discussing how one might achieve such scale, we summarize how distributed shared logs are commonly implemented, focusing on the **append** operation.

Roughly speaking, an **append**(v) operation takes two steps: (1) sequencing—assigning the next highest log index i to the new entry, and (2) replication—persisting the association $i \mapsto v$ at a sufficient number of log servers.[5] Replication is embarrassingly parallel; once there are enough replicas, sequencing becomes the throughput bottleneck [34].

[3] While different datacenter operators manage and deploy services differently, having independent services in each pod appears to be a common deployment practice.

[4] We based this estimate on the server specification of a high-end Ceph cluster [27], assuming a server is bottlenecked by its NIC. We expect a server to be equipped with enough NVMe SSDs to keep the network saturated. Any CPU bottleneck can be alleviated with hardware solutions (e.g., see Corfu's FPGA-based network-attached storage units [4, § 3.6]).

[5] The two steps can take place in either order [4, 17] or even in parallel [26], but this distinction has little bearing on our scaling discussion.

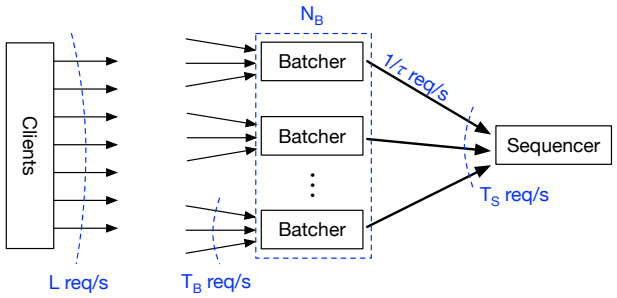

Figure 2: Scalable sequencing via batching. Total load is L; N_B batchers each processes a load of T_B (requests per second) and has a batching period of τ; the sequencer can process a load of T_S (batches per second).

So we zoom in on sequencing throughput. We can classify existing shared-log designs by whether or not their sequencing scheme is *scalable* (Figure 1):

Non-scalable sequencing The throughput of sequencing is capped by that of a single component—e.g., the sequencer in Corfu [4], a sequencing-layer replica in LazyLog [26]—resulting in a "hockey stick" latency curve (labeled "not scalable"). While such designs are simple and provide low latency, no computing component today is capable of processing 80 billion sequential requests per second.

Scalable sequencing Sequencing throughput grows with the amount of computing resources—at the expense of latency ("scalable" curves); a recent example is Scalog [17]. We must be in this regime, and ideally achieve reasonably low latency at our target scale ("scalable & fast" curve).

A standard recipe for scalable sequencing is batching: Requests are absorbed by a set of *batchers*, each of which forms batches that get periodically sent to a centralized sequencer. As illustrated in Figure 2, suppose each batcher can process T_B requests per second, and the sequencer T_S batches per second. To handle a total throughput of L, we need at least $N_B = L/T_B$ batchers. But to avoid overloading the sequencer, each batcher must send a load no more than T_S/N_B, i.e., one request per $\tau = N_B/T_S = L/(T_B \times T_S)$ time. Assuming latency is dominated by batching (which is likely at high load), the best achievable latency at throughput L would simply be τ.

We can now estimate the sequencing latency at our target throughput for a traditional, server-based scalable implementation (like Scalog). Taking $L = 80$ billion rps and estimating (single-threaded) server throughput to be $T_S = T_B = 1$ million rps, we get $\tau = 80$ ms. This latency is quite high for an intra-datacenter operation (not to mention a gross underestimate for Scalog, whose sequencer is Paxos-replicated [17, § 3]). RingWorld can reduce the sequencing latency using a ring of programmable switches to run the sequencing protocol.

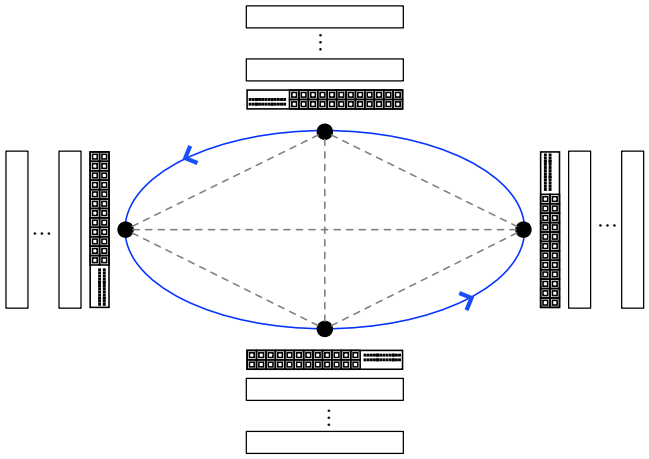

Figure 3: A RingWorld network with four racks. ToR switches are connected via a ring topology (blue ellipse) and a communication topology providing all-to-all connectivity (dashed lines).

3 RingWorld: A Ring of Switches

We now sketch RingWorld's design. Our main goal is to show that this design is feasible and that it achieves lower latency at our target scale. So we will focus on its distinguishing features and normal-case, performance-sensitive operations, omitting other details.

3.1 Overview

3.1.1 Replica Sets. The shared log is implemented jointly by log servers and ToR switches. As is standard [4, 17], we arrange log servers into *replica sets*, and each log entry is replicated in one replica set. We also follow Corfu's design in maintaining a global *projection* that maps a log index to a replica set, updated only during reconfiguration.

3.1.2 Topology. But unlike in a typical datacenter, in RingWorld the ToR switches are simultaneously connected via two network topologies (Figure 3):

Ring topology ToR switches are arranged in a directed ring, and two neighboring switches are connected via a direct link. We also assume some redundancy—e.g., connecting each switch also to the next-plus-one switch. This topology is reserved for token-passing in the sequencing primitive (§3.2.1).

Communication topology Switches are also connected in a typical datacenter topology (e.g., a tree-based one [1])

for data transmission. This topology is used for dissemination of data.

We can implement this arrangement by, say, starting from a tree-based topology and taking a few ports from each ToR switch to connect them in a ring.

3.1.3 Failure & Communication. We assume log servers may crash but can recover using stable storage [20] and that ToR switches are fail-stop with no persistent state. We assume that a channel exists between a log server and its ToR switch (via direct link) and between any two ToR switches (via network topology), and that all channels are fair lossy (meaning, roughly, that with retransmissions messages are eventually delivered).

We say that a log server is *reachable* if the server is up and its ToR switch has not failed. The RingWorld protocol is always safe, but can make progress only when (1) a majority of log servers in each replica set are reachable, and (2) a ring can be formed on the live switches using the ring topology.[6] In line with §2.2, we assume each replica set comprises three servers, tolerating one unreachable server per replica set.

3.2 Log Implementation

RingWorld implements two primitives on ToR switches, sequencing and replication, on top of which log operations are built. We now sketch how these primitives and operations work under normal conditions (i.e., when no switches fail).

3.2.1 Sequencing Primitive. Recall that the key to building a scalable log is to scale up sequencing. We achieve this by (1) performing sequencing on high-speed, programmable ToR switches, and (2) distributing load among the switches using a moving-sequencer scheme [15, § 4.2].

Specifically, the ToR switches constantly pass a *token* around the ring (in the form of a packet). The token records the largest log index h given out thus far, and whichever switch holds the token $\langle h \rangle$ gets to hand out log indices starting at $h + 1$. When a sequencing request arrives at a ToR switch, it remains pending (e.g., via recirculation where the packet is forwarded back to the switch), while the switch tracks the number r of pending requests. When the switch receives the token $\langle h \rangle$, it assigns log indices $h + 1$ to $h + r$ to the batch of pending requests and passes the token $\langle h + r \rangle$ to its successor. Thus, the switch dispatches one batch of sequencing requests every time the token comes around.

RingWorld's moving-sequencer algorithm has its roots in the total-order-broadcast literature from decades prior [11, 33]. But unlike in those instantiations, the RingWorld ring is placed on modern, high-speed programmable switches, disjoint from the log entries' destinations (log servers).

[6] A ring can always be formed using the communication topology, but a ring with non-direct links would have much higher latency.

3.2.2 Replication Primitive. This primitive replicates data associated with a log index to the replica set assigned by the projection and awaits acknowledgements from a majority.

For this purpose, each log index i is assigned to a *leader switch*—say, switch $(i \bmod N_R)$, where N_R is the number of racks. Each switch maintains (1) a commit index L, below which all log indices have been successfully replicated, and (2) a status map, which tracks the acknowledgements received for log indices currently being replicated.[7] When replication begins for index i, the leader switch puts i into the status map and multicasts the data to the replica set for i; it marks each acknowledgement received in the map and completes when the count reaches a majority. (We say that index i is now *committed*.)

Note that implementing the replication logic on switches is not necessary for scalability: Data for different log indices can be replicated in parallel, driven by servers. However, offloading this logic to switches improves fault tolerance since switches fail less frequently than servers [18]; this is an important concern given the number of servers involved at our scale. Offloading also improves server CPU efficiency.

3.2.3 Log Operations. Given the sequencing and replication primitives, log operations can be easily implemented:

append A log client (i.e., data-store server) sends an append request for data v to any ToR switch, which assigns it log index i and forwards it to the leader switch for i. The leader replicates under index i the data v and the current *view number* [25] (to detect stale data after failures), and returns the index to the client.

read A log client sends a read request for index i to the leader switch for i. If the entry has not been committed, the switch returns an error, and the client library will retry. Otherwise, the switch forwards the request to a majority in the replica set for i; they return the data and view number to the client, which adopts the data in the latest view. Furthermore, it should be possible to optimize reads to contact a single server in the common case, consistent with our calculations (§2.2).[8]

3.2.4 Failure Recovery. When a switch fails, the system must (1) reestablish the ring, (2) reassign log indices to the ring members, recompute commit indices, and complete any pending replications, and (3) restart token passing. We anticipate implementing these steps on the switch CPUs using well-established techniques: e.g., using a membership protocol

[7] Because a switch is leader for indices congruent modulo N_R, the map can be a ring buffer storing entries for indices $L + N_R$, $L + 2N_R$, etc. Each entry represents the set of log servers that have acknowledged the corresponding log index; this set can be represented as a bitmap as in P4xos [13].

[8] For example, given that a log index has been committed, getting its data *from the current view* from a single server is sufficient.

for Step (1) (like in Totem [2]) and adapting MultiPaxos's leader-change protocol [31, § 2.4] for Step (2).

3.3 Scalability and Performance

How well would RingWorld perform at our target load? Let us assume that the ring spans only the log servers, amounting to 500 racks (§2.2), and that the delay for passing the token one hop on the ring is 10 µs. Estimating a switch's single-pipeline throughput to be 1 billion packets per second, it is apparent that 500 switches provide ample capacity to sequence our target load of 80 billion appends per second. The sequencing latency would be dominated by the ring delay of $500 \times 10\,\mu s = 5\,ms$; this is a 16× reduction from the 80 ms latency estimated for a server-based solution (§2.2)!

3.4 Benefits Beyond Scalability

Up until now, we have focused on the scalability benefits of RingWorld. But we believe RingWorld's design potentially offers other benefits worth exploring. For example:

Failure detection and fencing The ToR switch is at a vantage point to quickly and accurately detect failures of log servers; once it suspects a server has failed, it can fence off the server by refusing to forward requests to it. This can significantly simplify protocols [23].

Subscription and filtering A data-store server may wish to *subscribe* to new log entries—but only those that match a certain condition [5]. The ToR switch can filter out entries that do not match the condition, reducing the load on the network and the server.

4 Conclusion

Prior work has argued that distributed shared logs are the *right way* to build consistent and fault-tolerant distributed data stores, but log scalability affects the scalability of the resulting data store. In this paper, we have argued that it is feasible to build a datacenter-wide distributed shared log that can support 80 billion appends per second using hardware that is available today. Consequently, scalability should no longer be considered an impediment to using logs when building data stores, and the community should investigate other trade-offs when architecting stores in this manner.

Acknowledgments

We thank the anonymous reviewers, Mahesh Balakrishnan, and members of the UC Berkeley NetSys Lab and Sky Computing Lab for their feedback.

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, August 17-22, 2008*. ACM, 63–74. doi:10.1145/1402958.1402967

[2] Yair Amir, Louise E. Moser, P. M. Melliar-Smith, Deborah A. Agarwal, and P. Ciarfella. 1995. The Totem Single-Ring Ordering and Membership Protocol. *ACM Trans. Comput. Syst.* 13, 4 (1995), 311–342. doi:10.1145/210223.210224

[3] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.* 31, 4 (2013), 10. doi:10.1145/2535930

[4] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 1–14. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/balakrishnan

[5] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: distributed data structures over a shared log. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 325–340. doi:10.1145/2517349.2522732

[6] 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, and Yee Jiun Song. 2021. Log-structured Protocols in Delos. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 538–552. doi:10.1145/3477132.3483544

[7] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Morgan & Claypool Publishers. doi:10.2200/S00874ED3V01Y201809CAC046

[8] Philip A. Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. 2015. Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1295–1309. doi:10.1145/2723372.2737788

[9] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. 2011. Hyder - A Transactional Record Manager for Shared Flash. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 9–20. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper2.pdf

[10] Michael Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, Brian N. Bershad and Jeffrey C. Mogul (Eds.). USENIX Association, 335–350. http://www.usenix.org/events/osdi06/tech/burrows.html

[11] Jo-Mei Chang and Nicholas F. Maxemchuk. 1984. Reliable Broadcast Protocols. *ACM Trans. Comput. Syst.* 2, 3 (1984), 251–273. doi:10.1145/989.357400

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed

Database. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 251–264. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[13] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. 2020. P4xos: Consensus as a Network Service. *IEEE/ACM Trans. Netw.* 28, 4 (2020), 1726–1738. doi:10.1109/TNET.2020.2992106

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, Thomas C. Bressoud and M. Frans Kaashoek (Eds.). ACM, 205–220. doi:10.1145/1294261.1294281

[15] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.* 36, 4 (2004), 372–421. doi:10.1145/1041680.1041682

[16] Akon Dey, Alan D. Fekete, and Uwe Röhm. 2015. Scalable distributed transactions across heterogeneous stores. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. IEEE Computer Society, 125–136. doi:10.1109/ICDE.2015.7113278

[17] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert van Renesse. 2020. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 325–338. https://www.usenix.org/conference/nsdi20/presentation/ding

[18] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*. ACM, 350–361. doi:10.1145/2018436.2018477

[19] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. doi:10.1145/78969.78972

[20] Michel Hurfin, Achour Mostéfaoui, and Michel Raynal. 1998. Consensus in Asynchronous Systems Where Processes Can Crash and Recover. In *The Seventeenth Symposium on Reliable Distributed Systems, SRDS 1998, West Lafayette, Indiana, USA, October 20-22, 1998, Proceedings*. IEEE Computer Society, 280–286. doi:10.1109/RELDIS.1998.740510

[21] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 691–707. doi:10.1145/3477132.3483541

[22] Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Xiangyao Yu, and Matei Zaharia. 2023. Epoxy: ACID Transactions Across Diverse Data Stores. *Proc. VLDB Endow.* 16, 11 (2023), 2742–2754. doi:10.14778/3611479.3611484

[23] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. 2015. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*. ACM, 9:1–9:16. doi:10.1145/2741948.2741976

[24] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, Huayong Wang, Shanyang Liu, Lulu Chen, Zhiwu Wu, Haonan Qiu, Derui Liu, Gexiao Tian, Chao Han, Shaozong Liu, Yaohui Wu, Zicheng Luo, Yuchao Shao, Junping Wu, Zheng Cao, Zhongjie Wu, Jiaji Zhu, Jinbo Wu, Jiwu Shu, and Jiesheng Wu. 2023. More Than Capacity: Performance-oriented Evolution of Pangu in Alibaba. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, Ashvin Goel and Dalit Naor (Eds.). USENIX Association, 331–346. https://www.usenix.org/conference/fast23/presentation/li-qiang-deployed

[25] Barbara Liskov and James Cowling. 2012. *Viewstamped Replication Revisited*. Technical Report MIT-CSAIL-TR-2012-021. MIT. https://pmg.csail.mit.edu/papers/vr-revisited.pdf

[26] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. 2024. LazyLog: A New Shared Log Abstraction for Low-Latency Applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*, Emmett Witchel, Christopher J. Rossbach, Andrea C. Arpaci-Dusseau, and Kimberly Keeton (Eds.). ACM, 296–312. doi:10.1145/3694715.3695983

[27] Mark Nelson. 2024. *Ceph: A Journey to 1 TiB/s*. https://ceph.io/en/news/blog/2024/ceph-a-journey-to-1tibps/

[28] Sheng Qi, Xuanzhe Liu, and Xin Jin. 2023. Halfmoon: Log-Optimal Fault-Tolerant Stateful Serverless Computing. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 314–330. doi:10.1145/3600006.3613154

[29] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 4–18. doi:10.1145/3514221.3526120

[30] Amin Vahdat. 2022. *Jupiter evolving: Reflecting on Google's data center network transformation*. https://cloud.google.com/blog/topics/systems/the-evolution-of-googles-jupiter-data-center-network

[31] Robbert van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *ACM Comput. Surv.* 47, 3 (2015), 42:1–42:36. doi:10.1145/2673577

[32] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1041–1052. doi:10.1145/3035918.3056101

[33] Brian Whetten, Todd Montgomery, and Simon M. Kaplan. 1994. A High Performance Totally Ordered Multicast Protocol. In *Theory and Practice in Distributed Systems, International Workshop, Dagstuhl Castle, Germany, September 5-9, 1994, Selected Papers (Lecture Notes in Computer Science, Vol. 938)*, Kenneth P. Birman, Friedemann Mattern, and André Schiper (Eds.). Springer, 33–57. doi:10.1007/3-540-60042-6_3

[34] Michael J. Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. 2021. Scaling Replicated State Machines with Compartmentalization. *Proc. VLDB Endow.* 14, 11 (2021), 2203–2215. doi:10.14778/3476249.3476273

[35] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. 2023. Electrode: Accelerating Distributed Protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 1391–1407. https://www.usenix.org/conference/nsdi23/presentation/zhou