



# Millions of Tiny Databases

Marc Brooker, Tao Chen, and Fan Ping, *Amazon Web Services*

<https://www.usenix.org/conference/nsdi20/presentation/brooker>

This paper is included in the Proceedings of the  
17th USENIX Symposium on Networked Systems Design  
and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Open access to the Proceedings of the  
17th USENIX Symposium on Networked  
Systems Design and Implementation  
(NSDI '20) is sponsored by



# Millions of Tiny Databases

Marc Brooker  
*Amazon Web Services*

Tao Chen  
*Amazon Web Services*

Fan Ping  
*Amazon Web Services*

## Abstract

Starting in 2013, we set out to build a new database to act as the configuration store for a high-performance cloud block storage system (Amazon EBS). This database needs to be not only highly available, durable, and scalable but also strongly consistent. We quickly realized that the constraints on availability imposed by the CAP theorem, and the realities of operating distributed systems, meant that we didn't want one database. We wanted millions. Physalia is a transactional key-value store, optimized for use in large-scale cloud control planes, which takes advantage of knowledge of transaction patterns and infrastructure design to offer both high availability and strong consistency to millions of clients. Physalia uses its knowledge of datacenter topology to place data where it is most likely to be available. Instead of being highly available for all keys to all clients, Physalia focuses on being extremely available for only the keys it knows each client needs, from the perspective of that client.

This paper describes Physalia in context of Amazon EBS, and some other uses within Amazon Web Services. We believe that the same patterns, and approach to design, are widely applicable to distributed systems problems like control planes, configuration management, and service discovery.

## 1 Introduction

Traditional architectures for highly-available systems assume that infrastructure failures are statistically independent, and that it is extremely unlikely for a large number of servers to fail at the same time. Most modern system designs are aware of broad failure domains (data centers or availability zones), but still assume two modes of failure: a complete failure of a datacenter, or a random uncorrelated failure of a server, disk or other infrastructure. These assumptions are reasonable for most kinds of systems. Schroder and Gibson found [51] that (in traditional datacenter environments), while the probability of a second disk failure in a week was up to 9x higher when a first failure had already occurred, this correlation drops off

to less than 1.5x as systems age. While a 9x higher failure rate within the following week indicates some correlation, it is still very rare for two disks to fail at the same time. This is just as well, because systems like RAID [43] and primary-backup failover perform well when failures are independent, but poorly when failures occur in bursts.

When we started building AWS in 2006, we measured the availability of systems as a simple percentage of the time that the system is available (such as 99.95%), and set Service Level Agreements (SLAs) and internal goals around this percentage. In 2008, we introduced AWS EC2 Availability Zones: named units of capacity with clear expectations and SLAs around correlated failure, corresponding to the datacenters that customers were already familiar with. Over the decade since, our thinking on failure and availability has continued to evolve, and we paid increasing attention to *blast radius* and correlation of failure. Not only do we work to make outages rare and short, we work to reduce the number of resources and customers that they affect [55], an approach we call *blast radius reduction*. This philosophy is reflected in everything from the size of our datacenters [30], to the design of our services, to operational practices.

Amazon Elastic Block Storage (EBS) is a block storage service for use with AWS EC2, allowing customers to create block devices on demand and attach them to their AWS EC2 instances. Volumes are designed for an annual failure rate (AFR) of between 0.1% and 0.2%, where failure refers to a complete or partial loss of the volume. This is significantly lower than the AFR of typical disk drives [44]. EBS achieves this higher durability through replication, implementing a chain replication scheme (similar to the one described by van Renesse, et al [54]). Figure 1 shows an abstracted, simplified, architecture of EBS in context of AWS EC2. In normal operation (of this simplified model), replicated data flows through the chain from client, to primary, to replica, with no need for coordination. When failures occur, such as the failure of the primary server, this scheme requires the services of a *configuration master*, which ensures that updates to the order and membership of the replication group occur atomically, are

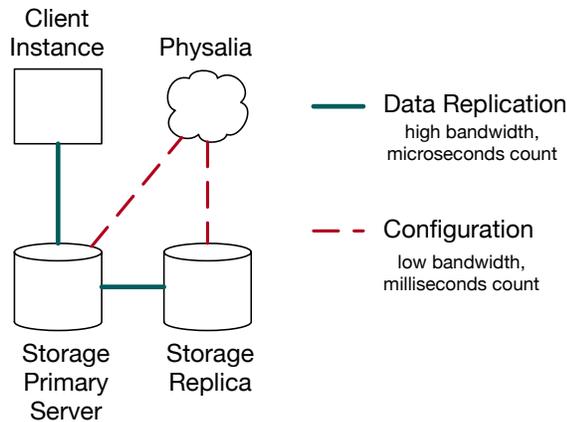


Figure 1: Simplified model of one EBS volume connected to an AWS EC2 instance. For the volume to be available for IO, either both the master and replica, or either storage server and Physalia, must be available to the instance.

well ordered, and follow the rules needed to ensure durability.

The requirements on this *configuration master* are unusual. In normal operation it handles little traffic, as replication continues to operate with no need to contact the *configuration master*. However, when large-scale failures (such as power failures or network partitions) happen, a large number of servers can go offline at once, requiring the *master* to do a burst of work. This work is latency critical, because volume IO is blocked until it is complete. It requires strong consistency, because any eventual consistency would make the replication protocol incorrect. It is also most critical at the most challenging time: during large-scale failures.

Physalia is a specialized database designed to play this role in EBS, and other similar systems at Amazon Web Services. Physalia offers both consistency and high availability, even in the presence of network partitions, as well as minimized blast radius of failures. It aims to fail gracefully and partially, and strongly avoid large-scale failures.

## 1.1 History

On 21 April 2011, an incorrectly executed network configuration change triggered a condition which caused 13% of the EBS volumes in a single Availability Zone (AZ) to become unavailable. At that time, replication configuration was stored in the EBS control plane, sharing a database with API traffic. From the public postmortem [46]:

When data for a volume needs to be re-mirrored, a negotiation must take place between the AWS EC2 instance, the EBS nodes with the volume data, and the EBS control plane (which acts as an authority in this process) so that only one copy of the

data is designated as the primary replica and recognized by the AWS EC2 instance as the place where all accesses should be sent. This provides strong consistency of EBS volumes. As more EBS nodes continued to fail because of the race condition described above, the volume of such negotiations with the EBS control plane increased. Because data was not being successfully re-mirrored, the number of these calls increased as the system retried and new requests came in. The load caused a brown out of the EBS control plane and again affected EBS APIs across the Region.

This failure vector was the inspiration behind Physalia's design goal of limiting the *blast radius* of failures, including overload, software bugs, and infrastructure failures.

## 1.2 Consistency, Availability and Partition Tolerance

As proven by Gilbert and Lynch [22], it is not possible for a distributed system to offer both strong consistency (in the sense of linearizability [31]), and be available to all clients in the presence of network partitions. Unfortunately, all real-world distributed systems must operate in the presence of network partitions [6], so systems must choose between strong consistency and availability. Strong consistency is non-negotiable in Physalia, because it's required to ensure the correctness of the EBS replication protocol. However, because chain replication requires a configuration change during network partitions, it is especially important for Physalia to be available during partitions.

Physalia then has the goal of optimizing for availability during network partitions, while remaining strongly consistent. Our core observation is that we do not require all keys to be available to all clients. In fact, each key needs to be available at only three points in the network: the AWS EC2 instance that is the client of the volume, the primary copy, and the replica copy. Through careful placement, based on our system's knowledge of network and power topology, we can significantly increase the probability that Physalia is available to the clients that matter for the keys that matter to those clients.

This is Physalia's key contribution, and our motivation for building a new system from the ground up: infrastructure aware placement and careful system design can significantly reduce the effect of network partitions, infrastructure failures, and even software bugs. In the same spirit as Paxos Made Live [12], this paper describes the details, choices and tradeoffs that are required to put a consensus system into production. Our concerns, notably blast radius reduction and infrastructure awareness, are significantly different from that paper.

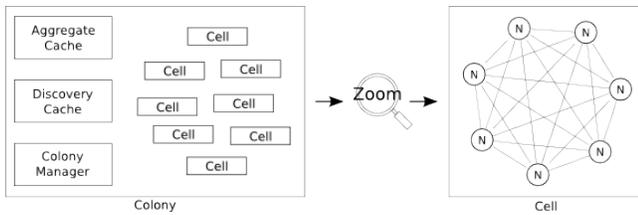


Figure 2: Overview of the relationship between the colony, cell and node.

## 2 The Design of Physalia

Physalia’s goals of blast radius reduction and partition tolerance required careful attention in the design of the data model, replication mechanism, cluster management and even operational and deployment procedures. In addition to these top-level design goals, we wanted Physalia to be easy and cheap to operate, contributing negligibly to the cost of our dataplane. We wanted its data model to be flexible enough to meet future uses in similar problem spaces, and to be easy to use correctly. This goal was inspired by the concept of *misuse resistance* from cryptography (GCM-SIV [27], for example), which aims to make primitives that are safer under misuse. Finally, we wanted Physalia to be highly scalable, able to support an entire EBS availability zone in a single installation.

### 2.1 Nodes, Cells and the Colony

The Portuguese man o’ war (*Physalia physalis*) is not one animal, but a siphonophore: a colonial organism made up of many specialized animals called zooids. These zooids are highly adapted to living in the colony, and cannot live outside it. Nevertheless, each zooid is a stand-alone organism, including everything that is required for life. Physalia’s high-level organization is similar: each Physalia installation is a *colony*, made up of many *cells*. The *cells* live in the same environment: a mesh of *nodes*, with each node running on a single server. Each *cell* manages the data of a single partition key, and is implemented using a distributed state machine, distributed across seven nodes. Cells do not coordinate with other cells, but each node can participate in many cells. The colony, in turn, can consist of any number of cells (provided there are sufficient nodes to distribute those cells over). Figure 2 captures the relationship between colony, cell and node. Figure 3 shows the cell: a mesh of nodes holding a single Paxos-based distributed state machine, with one of the nodes playing the role of *distinguished proposer*.

The division of a colony into a large number of cells is our main tool for reducing radius in Physalia. Each node is only used by a small subset of cells, and each cell is only used by a small subset of clients.

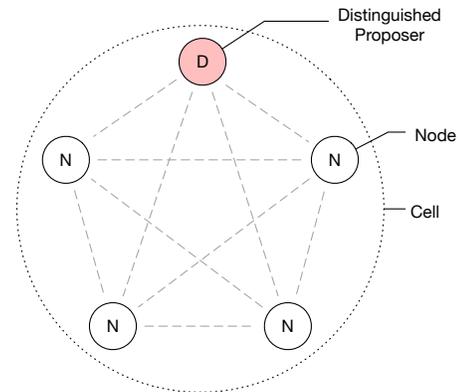


Figure 3: A cell is a group of nodes, one of which assumes the role of distinguished proposer.

Each Physalia colony includes a number of control plane components. The control plane plays a critical role in maintaining system properties. When a new cell is created, the control plane uses its knowledge of the power and network topology of the datacenter (discovered from AWS’s datacenter automation systems) to choose a set of nodes for the cell. The choice of nodes balances two competing priorities. Nodes should be placed close to the clients (where *close* is measured in logical distance through the network and power topology) to ensure that failures *far away* from their clients do not cause the cell to fail. They must also be placed with sufficient diversity to ensure that small-scale failures do not cause the cell to fail. Section 3 explores the details of placement’s role in availability.

The cell creation and repair workflows respond to requests to create new cells (by placing them on under-full nodes), handling cells that contain failed nodes (by replacing these nodes), and moving cells closer to their clients as clients move (by incrementally replacing nodes with closer ones).

We could have avoided implementing a separate control-plane and repair workflow for Physalia, by following the example of elastic replication [2] or Scatter [23]. We evaluated these approaches, but decided that the additional complexity, and additional communication and dependencies between shards, were at odds with our focus on blast radius. We chose to keep our cells completely independent, and implement the control plane as a separate system.

### 2.2 Physalia’s Flavor of Paxos

The design of each cell is a straightforward consensus-based distributed state machine. Cells use Paxos [35] to create an ordered log of updates, with batching and pipelining [48] to improve throughput. Batch sizes and pipeline depths are kept small, to keep per-item work well bounded and ensure short time-to-recovery in the event of node or network failure. Physalia uses a custom implementation of Paxos written

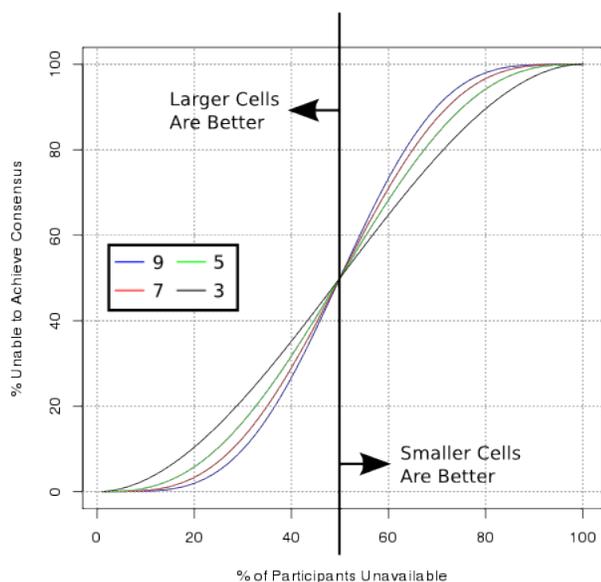


Figure 4: The size of cells is a trade-off between tolerance to large correlated failures and tolerance to random failures.

in Java, which keeps all required state both in memory and persisted to disk. In typical cloud systems, durability is made easier by the fact that systems can be spread across multiple datacenters, and correlated outages across datacenters are rare. Physalia’s locality requirement meant that we could not use this approach, so extra care in implementation and testing were required to ensure that Paxos is implemented safely, even across dirty reboots.

In the EBS installation of Physalia, the cell performs Paxos over seven nodes. Seven was chosen to balance several concerns:

- *Durability* improves exponentially with larger cell size [29]. Seven replicas means that each piece of data is durable to at least four disks, offering durability around 5000x higher than the 2-replication used for the volume data.
- Cell size has little impact on mean *latency*, but larger cells tend to have lower high percentiles because they better reject the effects of slow nodes, such as those experiencing GC pauses [17].
- The effect of cell size on *availability* depends on the type of failures expected. As illustrated in Figure 4, smaller cells offer lower availability in the face of small numbers of uncorrelated node failures, but better availability when the proportion of node failure exceeds 50%. While such high failure rates are rare, they do happen in practice, and a key design concern for Physalia.

Partition Key	Key	Value: int
	Key	Value: str
	Key	Value: bool
Partition Key	Key	Value
	Key	Value

Figure 5: The Physalia schema.

- Larger cells consume more resources, both because Paxos requires  $O(\text{cellsize})$  communication, but also because a larger cell needs to keep more copies of the data. The relatively small transaction rate, and very small data, stored by the EBS use of Physalia made this a minor concern.

The control plane tries to ensure that each node contains a different mix of cells, which reduces the probability of correlated failure due to load or poison pill transitions. In other words, if a poisonous transition crashes the node software on each node in the cell, only that cell should be lost. In the EBS deployment of Physalia, we deploy it to large numbers of nodes well-distributed across the datacenter. This gives the Physalia control plane more placement options, allowing it to optimize for widely-spread placement.

In our Paxos implementation, proposals are accepted optimistically. All transactions given to the proposer are proposed, and at the time they are to be applied (i.e. all transactions with lower log positions have already been applied), they are committed or ignored depending on whether the write conditions pass. The advantage of this optimistic approach is that the system always makes progress if clients follow the typical optimistic concurrency control (OCC) pattern. The disadvantage is that the system may do significant additional work during contention, passing many proposals that are never committed.

### 2.3 Data Model and API

The core of the Physalia data model is a partition key. Each EBS volume is assigned a unique partition key at creation time, and all operations for that volume occur within that partition key. Within each partition key, Physalia offers a transactional store with a typed key-value schema, supporting strict serializable reads, writes and conditional writes over any combination of keys. It also supports simple in-place operations like atomic increments of integer variables. Figure 5 shows the schema: one layer of partition keys, any number (within operational limitations) of string keys within a partition, and one value per key. The API can address only one partition key at a time, and offers strict serializable batch and conditional operations within the partition.

The goal of the Physalia API design was to balance two

competing concerns. The API needed to be expressive enough for clients to take advantage of the (per-cell) transactional nature of the underlying store, including expressing conditional updates, and atomic batch reads and writes. Increasing API expressiveness, on the other hand, increases the probability that the system will be able to accept a transition that cannot be applied (a poison pill). The Physalia API is inspired by the Amazon DynamoDB API, which supports atomic batched and single reads and writes, conditional updates, paged scans, and some simple in-place operations like atomic increments. We extended the API by adding a compound read-and-conditional-write operation.

Physalia’s data fields are strong but dynamically typed. Supported field types include byte arrays (typically used to store UTF-8 string data), arbitrary precision integers, and booleans. Strings are not supported directly, but may be offered as a convenience in the client. Floating-point data types and limited-precision integers are not supported due to difficulties in ensuring that nodes will produce identical results when using different software versions and hardware (see [24] and chapter 11 of [1]). As in any distributed state machine, it’s important that each node in a cell gets identical results when applying a transition. We chose not to offer a richer API (like SQL) for a similar reason: our experience is that it takes considerable effort to ensure that complex updates are applied the same way by all nodes, across all software versions.

Physalia provides two consistency modes to clients. In the consistent mode, read and write transactions are both linearizable and serializable, due to being serialized through the state machine log. Most Physalia clients use this consistent mode. The eventually consistent mode supports only reads (all writes are consistent), and offers a consistent prefix [7] to all readers and monotonic reads [53] within a single client session. Eventually consistent reads are provided to be used for monitoring and reporting (where the extra cost of linearizing reads worth it), and the discovery cache (which is eventually consistent anyway).

The API also offers first-class leases [25] (lightweight time-bounded locks). The lease implementation is designed to tolerate arbitrary clock skew and short pauses, but will give incorrect results if long-term clock rates are too different. In our implementation, this means that the fastest node clock is advancing at more than three times the rate of the slowest clock. Despite lease safety being highly likely, leases are only used where they are not critical for data safety or integrity.

In the Physalia API, all keys used to read and write data, as well as conditions for conditional writes, are provided in the input transaction. This allows the proposer to efficiently detect which changes can be safely batched in a single transaction without changing their semantics. When a batch transaction is rejected, for example due to a conditional put failure, the proposer can remove the offending change from the batch and re-submit, or submit those changes without batching.

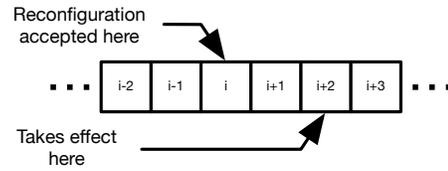


Figure 6: Changes in membership are placed into the log, but only take effect some time later (pictured here is  $\alpha = 2$ )

## 2.4 Reconfiguration, Teaching and Learning

As with our core consensus implementation, Physalia does not innovate on reconfiguration. The approach taken of storing per-cell configuration in the distributed state machine and passing a transition with the existing jury to update it follows the pattern established by Lamson [37]. A significant factor in the complexity of reconfiguration is the interaction with pipelining: configuration changes accepted at log position  $i$  must not take effect logically until position  $i + \alpha$ , where  $\alpha$  is the maximum allowed pipeline length (illustrated in Figure 6). Physalia keeps  $\alpha$  small (typically 3), and so simply waits for natural traffic to cause reconfiguration to take effect (rather than stuffing no-ops into the log). This is a very sharp edge in Paxos, which doesn’t exist in either Raft [42] or Viewstamped Replication [41].

Physalia is unusual in that reconfiguration happens frequently. The colony-level control plane actively moves Physalia cells to be close to their clients. It does this by replacing far-away nodes with close nodes using reconfiguration. The small data sizes in Physalia make cell reconfiguration an insignificant portion of overall datacenter traffic. Figure 7 illustrates this process of movement by iterative reconfiguration. The system prefers safety over speed, moving a single node at a time (and waiting for that node to catch up) to minimize the impact on durability. The small size of the data in each cell allows reconfiguration to complete quickly, typically allowing movement to complete within a minute.

When nodes join or re-join a cell they are brought up to speed by *teaching*, a process we implement outside the core consensus protocol. We support three modes of teaching. In the bulk mode, most suitable for new nodes, the teacher (any existing node in the cell) transfers a bulk snapshot of its state machine to the learner. In the log-based mode, most suitable for nodes re-joining after a partition or pause, the teacher ships a segment of its log to the learner. We have found that this mode is triggered rather frequently in production, due to nodes temporarily falling behind during Java garbage collection pauses. Log-based learning is chosen when the size of the missing log segment is significantly smaller than the size of the entire dataset.

Finally, packet loss and node failures may leave persistent holes in a node’s view of the log. If nodes are not able to find

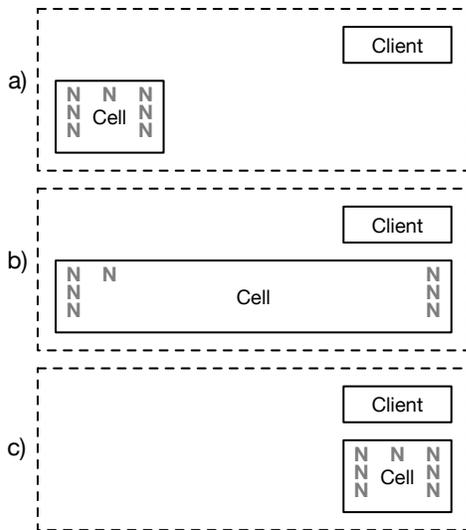


Figure 7: When Physalia detects that a cell’s client has moved (a), it replaces nodes in the cell with ones closer to the client (b), until the cell is entirely nearby the client (c).

another to teach them the decided value in that log position (or no value has been decided), they use a whack-a-mole learning mode. In whack-a-mole mode, a learner actively tries to propose a no-op transition into the vacant log position. This can have two outcomes: either the acceptors report no other proposals for that log position and the no-op transition is accepted, or another proposal is found and the learner proposes that value. This process is always safe in Paxos, but can affect liveness, so learners apply substantial jitter to whack-a-mole learning.

## 2.5 The Discovery Cache

Clients find cells using a distributed *discovery cache*. The *discovery cache* is a distributed eventually-consistent cache which allow clients to discover which nodes contain a given cell (and hence a given partition key). Each cell periodically pushes updates to the cache identifying which partition key they hold and their node members. Incorrect information in the cache affects the liveness, but never the correctness, of the system. We use three approaches to reduce the impact of the *discovery cache* on availability: client-side caching, forwarding pointers, and replication. First, it is always safe for a client to cache past discovery cache results, allowing them to refresh lazily and continue to use old values for an unbounded period on failure. Second, Physalia nodes keep long-term (but not indefinite) forwarding pointers when cells move from node to node. Forwarding pointers include pointers to all the nodes in a cell, making it highly likely that a client will succeed in pointer chasing to the current owner provided that it can get to at least one of the past owners. Finally, because the discovery cache is small, we can economically keep many

copies of it, increasing the probability that at least one will be available.

## 2.6 System Model and Byzantine Faults

In designing Physalia, we assumed a system model where messages can be arbitrarily lost, replayed, re-ordered, and modified after transmission. Message authentication is implemented using a cryptographic HMAC on each message, guarding against corruption occurring in lower layers. Messages which fail authentication are simply discarded. Key distribution, used both for authentication and prevention of unintentional Sybil-style attacks [20] is handled by our environment (and therefore out of the scope of Physalia), optimising for frequent and low-risk key rotation.

This model extends the “benign faults” assumptions of Paxos [11] slightly, but stops short of Byzantine fault tolerance<sup>1</sup>. While Byzantine consensus protocols are well understood, they add significant complexity to both software and system interactions, as well as testing surface area. Our approach was to keep the software and protocols simpler, and mitigate issues such as network and storage corruption with cryptographic integrity and authentication checks at these layers.

## 3 Availability in Consensus Systems

State-machine replication using consensus is popular approach for building systems that tolerate faults in single machines, and uncorrelated failures of a small number of machines. In theory, systems built using this pattern can achieve extremely high availability. In practice, however, achieving high availability is challenging. Studies across three decades (including Gray in 1990 [26], Schroeder and Gibson in 2005 [50] and Yuan et al in 2014 [57]) have found that software, operations, and scale drive downtime in systems designed to tolerate hardware faults. Few studies consider a factor that is especially important to cloud customers: large-scale correlated failures which affect many cloud resources at the same time.

### 3.1 Physalia vs the Monolith

It is well known that it is not possible to offer both all-clients availability and consistency in distributed databases due to the presence of network partitions. It is, however, possible to offer both consistency and availability to clients on the majority side of a network partition. While long-lived network partitions are rare in modern datacenter networks, they do occur, both due to the network itself and other factors (see Bailis and Kingsbury [6] and Alquraan et al [5] for surveys of

<sup>1</sup>This approach is typical of production consensus-based systems, including popular open-source projects like Zookeeper and etcd

causes of network partitions). Short-lived partitions are more frequent. To be as available as possible to its clients, Physalia needs to be on the same side of any network partition as them. For latency and throughput reasons, EBS tries to keep the storage replicas of a volume close to the AWS EC2 instances the volumes are attached to, both in physical distance and network distance. This means that client, data master and data replica are nearby each other on the network, and Physalia needs to be nearby too. Reducing the number of network devices between the Physalia database and its clients reduces the possibility of a network partition forming between them for the simple reason that fewer devices means that there's less to go wrong.

Physalia also optimizes for blast radius. We are not only concerned with the availability of the whole system, but want to avoid failures of the whole system entirely. When failures happen, due to any cause, they should affect as small a subset of clients as possible. Limiting the number of cells depending on a single node, and clients on a single cell, significantly reduce the effect that one failure can have on the overall system.

This raises the obvious question: does Physalia do better than a monolithic system with the same level of redundancy? A monolithic system has the advantage of less complexity. No need for the discovery cache, most of the control plane, cell creation, placement, etc. Our experience has shown that simplicity improves availability, so this simplification would be a boon. On the other hand, the monolithic approach loses out on partition tolerance. It needs to make a trade-off between being localized to a small part of the network (and so risking being partitioned away from clients), or being spread over the network (and so risking suffering an internal partition making some part of it unavailable). The monolith also increases blast radius: a single bad software deployment could cause a complete failure (this is similar to the node count trade-off of Figure 4, with one node).

## 3.2 Placement For Availability

The EBS control plane (of which the Physalia control plane is a part) continuously optimizes the availability of the EBS volume  $P(A_v)$  to the client AWS EC2 instance, and the EBS storage servers that store the volume. This is most interesting to do when the client instance is available. If the volume is unavailable at the same time as the client instance, we know that the instance will not be trying to access it. In other words, in terms of the availability of the volume ( $A_v$ ), and the instance ( $A_i$ ), the control plane optimizes the conditional probability  $P(A_v|A_i)$ . The ideal solution to this problem is to entirely co-locate the volume and instance, but EBS offers the ability to detach a volume from a failed instance, and re-attach it to another instance. To make this useful, volumes must continue to be durable even if the instance suffers a failure. Placement must therefore balance the concerns of

having the volume close enough for correlated availability, but far enough away for sufficiently independent durability to meet EBS's durability promise.

As an example, consider an idealized datacenter with three levels of network (servers, racks and rows) and three power domains (A, B and C). The client instance is on one rack, the primary copy on another, and replica copy on a third, all within the same row. Physalia's placement will then ensure that all nodes for the cell are within the row (there's no point being available if the row is down), but spread across at least three racks to ensure that the loss of one rack doesn't impact availability. It will also ensure that the nodes are in three different power domains, with no majority in any single domain.

This simple scheme faces two challenges. One is that real-world datacenter topology is significantly more complex, especially where datacenters contain multiple generations of design and layout. Another is that EBS volumes move by replication, and their clients move by customers detaching their volumes from one instance and attaching them to another. The Physalia control plane continuously responds to these changes in state, moving nodes to ensure that placement constraints continue to be met.

## 3.3 Non-Infrastructure Availability Concerns

Another significant challenge with building high-availability distributed state machines is correlated work. In a typical distributed state machine design, each node is processing the same updates and the same messages in the same order. This leads the software on the machines to be in the same state. In our experience, this is a common cause of outages in real-world systems: redundancy does not add availability if failures are highly correlated. Having all copies of the software in the same state tends to trigger the same bugs in each copy at the same time, causing multiple nodes to fail, either partially or completely, at the same time. Another issue is that the correlated loads cause memory, hard drives, and other storage on each host to fill up at the same rate. Again, this causes correlated outages when each host has the same amount of storage. Poison pill transactions may also cause outages; these are transactions that are accepted by the cell but cannot be applied to the state machine once consensus is reached.

Software deployments and configuration changes also contribute to downtime. Good software development practices, including code review and automated and manual testing, can reduce the risk of software changes but not entirely eliminate it. Incremental deployment, where code is rolled out slowly across the fleet and rolled back at the first sign of trouble, is a required operational practice for highly available systems. The fault-tolerant nature of distributed state machines makes this approach less effective: because the system is designed to tolerate failure in less than half of hosts, failure may not be evident until new code is deployed to half of all hosts. Prac-

tices like positive validation, where the deployment system checks that new nodes are taking traffic, reduce but do not eliminate this risk.

Poison pills are a particularly interesting case of software failure. A poison pill is a transaction which passes validation and is accepted into the log, but cannot be applied without causing an error. Pipelining requires that transactions are validated before the state they will execute on is fully known, meaning that even simple operations like numerical division could be impossible to apply. In our experience, poison pills are typically caused by under-specification in the transaction logic ("*what does dividing by zero do?*", "*what does it mean to decrement an unsigned zero?*"), and are fixed by fully specifying these behaviors (a change which comes with its own backward-compatibility challenges).

All of these factors limit the availability of any single distributed state machine, as observed by its clients. To achieve maximum availability, we need many such systems spread throughout the datacenter. This was the guiding principle of Physalia: instead of one database, build millions.

### 3.4 Operational Practices

Our experience of running large distributed systems is that operations, including code and configuration deployments, routine system operations such as security patching, and scaling for increased load, are dominant contributors to system downtime, despite ongoing investments in reducing operational defect rates. This conclusion isn't particular to the environment at AWS. For example, Jim Gray found in 1990 that the majority of failures of Tandem computers were driven by software and operations [26]. Operational practices at AWS already separate operational tasks by region and availability zone, ensuring that operations are not performed across many of these units at the same time.

Physalia goes a step further than this practice, by introducing the notion of *colors*. Each cell is assigned a color, and each cell is constructed only of nodes of the same color. The control plane ensures that colors are evenly spread around the datacenter, and color choice minimally constrains how close a cell can be to its clients. Physalia's very large node and cell counts make this possible. When software deployments and other operations are performed, they proceed color-by-color. Monitoring and metrics are set up to look for anomalies in single colors. Colors also provide a layer of isolation against load-related and poison pill failures. Nodes of different colors don't communicate with each other, making it significantly less likely that a poison pill or overload could spread across colors.

### 3.5 Load in Sometimes-Coordinating Systems

Load is another leading cause of correlated failures. Fundamentally, a consensus-based system needs to include more

than half of all nodes in each consensus decision, which means that overload can take out more than half of all nodes. Colors play a role in reducing the blast radius from load spikes from a few clients, but the load on Physalia is inherently spiky.

During normal operation, load consists of a low rate of calls caused by the background rate of EBS storage server failures, and creation of new cells for new volumes. During large-scale failures, however, load can increase considerably. This is an inherent risk of *sometimes-coordinating* systems like EBS: recovery load is not constant, and highest during bad network or power conditions. See Section 5.2.1 for a brief exploration of the magnitude of these spikes.

Per-cell Physalia throughput, as is typical of Paxos-style systems, scales well up to a point, with significant wins coming from increased batch efficiency. Beyond this point, however, contention and the costs of co-ordination cause goodput to drop with increased load (as predicted by Gunther's model [28]). To avoid getting into this reduced-goodput mode, cells reject load once their pipelines are full. While this isn't a perfect predictor of load, it works well because it decreases attempted throughput with increased latency (and is therefore stable in the control theory sense), and gets close to peak system throughput. Clients are expected to exponentially back off, apply jitter, and eventually retry their rejected transactions. As the number of clients in the Physalia system is bounded, this places an absolute upper limit on load, at the cost of latency during overload.

## 4 Testing

The challenge of testing a system like Physalia is as large as the challenge of designing and building it. Testing needs to cover not only the happy case, but also a wide variety of error cases. Our experience mirrors the findings of Yuan, et al [57] that error handling is where many bugs hide out, and Alquraan et al [5] that network partitions are rare events that easily hide bugs. As Kingsbury's Jepsen [33] testing work has found, many consensus implementations also have bugs in the happy path. Good testing needs to look everywhere. To get the coverage required, we needed to make the bar to building a new test case extremely low.

### 4.1 The SimWorld

To solve this problem, we picked an approach that is in wide use at Amazon Web Services, which we would like to see broadly adopted: build a test harness which abstracts networking, performance, and other systems concepts (we call it a *simworld*). The goal of this approach is to allow developers to write distributed systems tests, including tests that simulate packet loss, server failures, corruption, and other failure cases, as *unit tests* in the same language as the system itself. In this case, these *unit tests* run inside the developer's IDE (or with *junit* at build time), with no need for test clusters or other

infrastructure. A typical test which tests correctness under packet loss can be implemented in less than 10 lines of Java code, and executes in less than 100ms. The Physalia team have written hundreds of such tests, far exceeding the coverage that would be practical in any cluster-based or container-based approach.

The key to building a *simworld* is to build code against abstract physical layers (such as networks, clocks, and disks). In Java we simply wrap these thin layers in interfaces. In production, the code runs against implementations that use real TCP/IP, DNS and other infrastructure. In the *simworld*, the implementations are based on in-memory implementations that can be trivially created and torn down. In turn, these in-memory implementations include rich fault-injection APIs, which allow test implementors to specify simple statements like:

```
net.partitionOff(PARTITION_NAME, p5.  
    getLocalAddress());  
...  
net.healPartition(PARTITION_NAME);
```

Our implementation allows control down to the packet level, allowing testers to delay, duplicate or drop packets based on matching criteria. Similar capabilities are available to test disk IO. Perhaps the most important testing capability in a distributed database is time, where the framework allows each actor to have it's own view of time arbitrarily controlled by the test. Simworld tests can even add Byzantine conditions like data corruption, and operational properties like high latency. We highly recommend this testing approach, and have continued to use it for new systems we build.

## 4.2 Additional Testing Approaches

In addition to unit testing, we adopted a number of other testing approaches. One of those approaches was a suite of automatically-generated tests which run the Paxos implementation through every combination of packet loss and re-ordering that a node can experience. This testing approach was inspired by the TLC model checker [56], and helped us build confidence that our implementation matched the formal specification.

We also used the open source Jepsen tool [33] to test the system, and make sure that the API responses are linearizable under network failure cases. This testing, which happens at the infrastructure level, was a good complement to our lower-level tests as it could exercise some under-load cases that are hard to run in the *simworld*.

Finally, we performed a number of *game days* against deployments of Physalia. A game day is a failure simulation that happens in a real production or production-like deployment of a system, an approach that has been popular at Amazon for 20 years. Game days test not only the correctness of code, but also the adequacy of monitoring and logging, effectiveness of operational approaches, and the team's understanding of

how to debug and fix the system. Our game day approach is similar to the chaos engineering approach pioneered by Netflix [32], but typically focuses on larger-scale failures rather than component failures.

## 4.3 The Role of Formal Methods

TLA+ [36] is a specification language that's well suited to building formal models of concurrent and distributed systems. We use TLA+ extensively at Amazon [39], and it proved exceptionally useful in the development of Physalia. Our team used TLA+ in three ways: writing specifications of our protocols to check that we understand them deeply, model checking specifications against correctness and liveness properties using the TLC model checker, and writing extensively commented TLA+ code to serve as the documentation of our distributed protocols. While all three of these uses added value, TLA+'s role as a sort of automatically tested (via TLC), and extremely precise, format for protocol documentation was perhaps the most useful. Our code reviews, *simworld* tests, and design meetings frequently referred back to the TLA+ models of our protocols to resolve ambiguities in Java code or written communication. We highly recommend TLA+ (and its Pluscal dialect) for this use.

One example of a property we checked using TLA+ is the safety of having stale information in the discovery cache. For correctness, it is required that a client acting on stale information couldn't cause a *split brain* by allowing a group of old nodes to form a quorum. We started with the informal argument that the reconfiguration protocol makes  $f > \frac{N}{2}$  of the pre-reconfiguration nodes aware of a configuration change, and therefore aware if they have been deposed from the jury. In other words, at most  $\lfloor \frac{N}{2} \rfloor$  nodes may have been deposed from the jury without being aware of the change, and because they do not form a quorum they cannot pass *split brain* proposals. This argument becomes successively more complex as multiple reconfigurations are passed, especially during a single window of  $\alpha$ . Multiple reconfigurations also introduce an ABA problem when cells move off, and then back onto, a node. TLA+ and TLC allowed us to build confidence in the safety of our protocols in this complex case and cases like it.

## 5 Evaluation

Evaluating the performance of a system like Physalia is challenging. Performance, including throughput and latency, are important, but the most important performance metrics are how the system performs during extremely rare large-scale outages. We evaluate the performance of Physalia in production, and evaluate the design through simulations. We also use simulations to explore some particularly challenging whole-system aspects of Physalia.

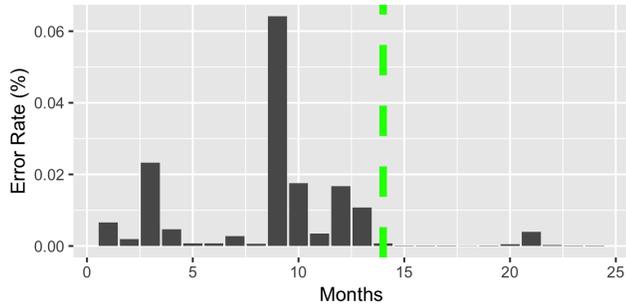


Figure 8: Mean availability of the configuration store from the perspective of the EBS primary, bucketed by month, for a production colony. The vertical line shows the deployment of Physalia in this datacenter, replacing a legacy system.

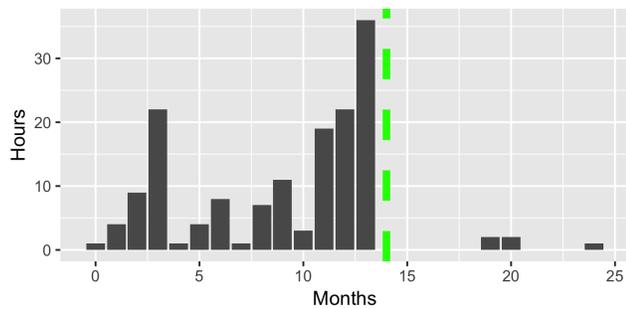


Figure 9: Number of hours per month where EBS masters experienced an error rate  $> 0.05\%$  in a production colony. The vertical line shows the deployment of Physalia.

## 5.1 Production Experience

Physalia is deployed in production in AWS, running in over 60 availability zones. Figure 8 shows the effect that its deployment has had on one measure of volume availability: how often the primary copy of the volume is able to contact the configuration store on the first try. The deployment of Physalia shows a clear ( $p = 7.7 \times 10^{-5}$ ) improvement in availability. Availability failures in the previous system were caused both by infrastructure failures and by transient overload (see Section 3.5).

Figure 9 shows the same data in a different way, looking at compliance against an internal error rate goal, significantly stricter than the external SLA for EBS. In this case, the internal goal is  $0.05\%$ , and we count the number of hours where this goal is exceeded.

In production deployments within AWS, Physalia deployments at availability-zone scale routinely serve thousands of requests per second. Latency varies between read and write optimizations. Linearizable reads can sometimes be handled by the distinguished proposer. Writes, on the other hand, need

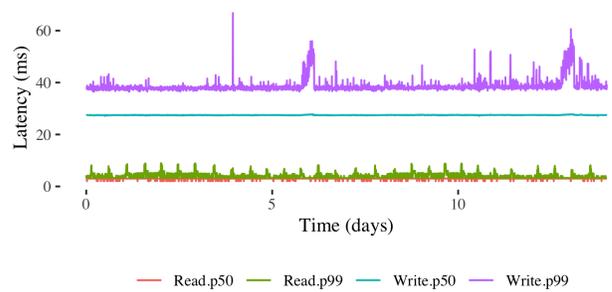


Figure 10: Physalia read and write latencies for one large-scale cluster. p50 is the 50th percentile, and p99 is the 99th.

to complete a Paxos round before they are committed, and therefore require substantially more communication. Figure 10 presents a multi-day view of read and write latency percentiles, calculated on a one-minute bucket. In this typical installation, reads take less than 10ms at the 99th percentile, and writes typically take less than 50ms.

In a distributed state machine, not only must operations be applied deterministically across all replicas, but they must be applied the same way by all production versions. Our operational and testing practices handle this edge case by testing between adjacent versions. Early in our production rollout, a bug in our deployment tools led to a *rollback* to an old version of the code base on a small number of nodes. These nodes applied transactions differently, simply not applying a conditional they didn't understand, leading to state to diverge on the cells where they were members. While we fixed this issue quickly with little customer impact, we took three important lessons away from it. First, Postel's famous robustness principle (*be conservative in what you do, be liberal in what you accept from others*) [45] does not apply to distributed state machines: they should not accept transactions they only partially understand and allow the consensus protocol to treat them as temporarily failed. Second, our testing processes needed to cover more than adjacent versions, and include strong mechanisms for testing rollback cases (both expected and unexpected). The third lesson is perhaps the most important: control planes should exploit their central position in a systems architecture to offer additional safety. When the rollback issue occurred, affected cells were corrupted in a way that caused the control plane to see them as empty, and available for deletion. The control plane dutifully took action, deleting the cells. Based on this experience, we modified the control plane to add rate limiting logic (don't move faster than the expected rate of change), and a *big red button* (allowing operators to safely and temporarily stop the control plane from taking action). Control planes provide much of the power of the cloud, but their privileged position also means that they have to act safely, responsibly, and carefully to avoid

introducing additional failures.

## 5.2 Design Validation via Simulation

The statistical behavior of a system as complex as Physalia can be difficult, if not intractable, to analyze in closed form. From early in the feasibility stages to production deployment, we used simulation to understand the dynamic behavior of the system, explore alternative system designs, and calculate baselines for our testing. In this section, we present some simulation results, and conclude by comparing the performance of the system to those results.

The availability offered by a Physalia deployment is highly sensitive to the failure modes of the underlying infrastructure, and the statistical properties of each of those failure modes. These results use a simplified (and outdated) model of a datacenter network: servers are organized into racks, each with a top-of-rack switch (*tor*), which in turn connects to one or more aggregation routers (*aggs*), which connect to one or more core routers (*cores*). Typical real-world networks contain some redundancy. For example, a *tor* is likely to connect to more than one *agg*. In these results we've left out redundancy for simplicity's sake, but the results are qualitatively similar (although the failure statistics are very different), once redundancy is considered.

One significant area that we explored with simulation is placement. Globally optimizing the placement of Physalia volumes is not feasible for two reasons, one is that it's a non-convex optimization problem across huge numbers of variables, the other is that it needs to be done online because volumes and cells come and go at a high rate in our production environment. Figure 11 shows the results of using one very rough placement heuristic: a sort of *bubble sort* which swaps nodes between two cells at random if doing so would improve locality. In this simulation, we considered 20 candidates per cell. Even with this simplistic and cheap approach to placement, Physalia is able to offer significantly (up to 4x) reduced probability of losing availability.

### 5.2.1 Simulations of System Load

As discussed in Section 3.5, load on Physalia can vary dramatically with different network conditions. Simulation of failures in different network topologies allows us to quantify the maximum expected load. Figure 12 shows the results of simulating *agg* failures (in the same model used above) on offered load to Physalia. A volume needs to call Physalia if the client AWS EC2 instance can get to either the master or replica EBS server, but the master and replica can't get to each other.

At small failure rates, expected load increases linearly with the count of failed devices, up to maximum of 29%. Beyond this, load drops off, as volumes become likely to be completely disconnected from the client. Multiplying this graph

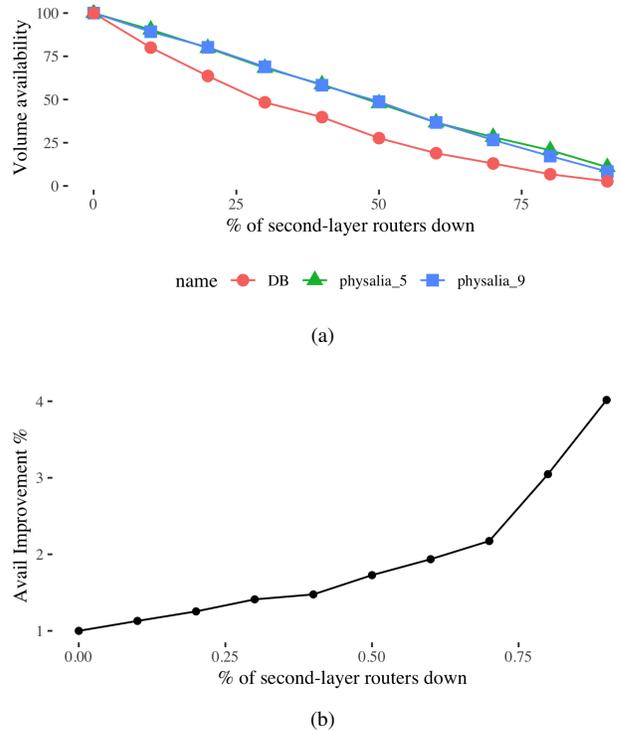


Figure 11: Simulated availability of volumes using Physalia, versus a baseline of a single-point database, under network partitions caused by device failures at the *agg* layer. (a) shows raw results for cell sizes 5 and 9, and (b) shows the ratio between Physalia and baseline availability.

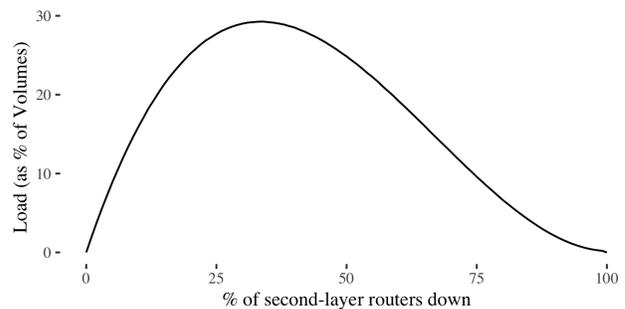


Figure 12: Load on Physalia vs. *agg* failure rate for a simulated 3-tier datacenter network.

(or, ideally one simulated on actual datacenter topology) with the expected values of device failures yields a graph of the expectation of the magnitude of maximum load on Physalia (or, indeed, any configuration master in an EBS-like replicated system). These results closely match what we have observed of the real-world behavior of the EBS deployment of Physalia.

## 6 Related Work

Physalia draws ideas from both distributed co-ordination systems and distributed databases. Distributed co-ordination systems, like Zookeeper [19], Chubby [9], Boxwood [38] and etcd [14], have the goal of providing a highly-available and strongly-consistent set of basic operations that make implementing larger distributed systems easier. Physalia's design approach is similar to some of these systems, being based on the state machine replication pattern popularized by the work of Schneider [49], Oki [40] and Lamson [37]. Physalia's key differences from these systems are its fine-grained consensus (millions of distributed state machines, rather than a single one), and infrastructure awareness. This makes Physalia more scalable and more resistant to network partitions, but also significantly more complex.

The problem of providing highly-available distributed storage in fallible datacenter networks faces similar challenges to global and large-scale systems like OceanStore [34] and Farsite [3], with emphasis on moving data close to its expected to improve availability and latency. While the design of Physalia predates the publication of Spanner [15] and CosmosDB, Physalia takes some similar design approaches with similar motivation.

Horizontal partitioning of databases is a long-established idea for both scaling and availability. Systems like Dynamo [18] and its derivatives dynamically move partitions, and rely on client behavior or stateless proxies for data discovery. Dynamic discovery of high-cardinality data, as addressed by Physalia's discovery cache and forwarding pointers, has been well explored by systems like Pastry [47] and Chord [52]. Optimizing data placement for throughput and latency is also a well-established technique (such as in Tao [8], and Dabek et al [16]), but these systems are not primarily concerned with availability during partitions, and do not consider blast radius.

Physalia's approach to infrastructure-aware placement reflects some techniques from software-defined networking (SDN) [21]. Another similarity with SDN (and earlier systems, like RCP [10]) is the emphasis on separating control and data planes, and allowing the data plane to consist of simple packet-forwarding elements. This reflects similar decisions to separate Physalia from the data plane of EBS, and the data and control planes of Physalia itself.

Infrastructure awareness, an important part of Physalia's contribution, seems to be an under-explored area in the systems literature. Some systems (like SAUCR [4], and the model proposed by Chen et al [13]) are designed to change operating modes when infrastructure failures occur or request patterns change, but we are not aware of other database explicitly designed to include data placement based on network topology (beyond simple locality concerns).

## 7 Conclusion

Physalia is a classic consensus-based database which takes a novel approach to availability: it is aware of the topology and datacenter power and networking, as well as the location of the clients that are most likely to need each row, and uses data placement to reduce the probability of network partitions. This approach was validated using simulation, and the gains have been borne out by our experience running it in production at high scale across over 60 datacenter-scale deployments. Its design is also optimized to reduce blast radius, reducing the impact of any single node, software, or infrastructure failure.

While few applications have the same constraints that we faced, many emerging cloud patterns require strongly consistent access to local data. Having a highly-available strongly-consistent database as a basic primitive allows these systems to be simpler, more efficient, and offer better availability.

## Acknowledgments

The authors would like to thank David R. Richardson, Madhu Parthasarathy, Divya Jain, Surya Dhoolam, and the Physalia team for their work in supporting, developing and refining Physalia. We would also like to thank Marc Levy, Allan Vermeulen and James Hamilton for reviewing the initial Physalia design. Allan Vermeulen's work on distributed systems testing deeply informed our approach.

## References

- [1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. doi:10.1109/IEEESTD.2008.4610935.
- [2] Hussam Abu-Libdeh, Robbert van Renesse, and Ymir Vigfusson. Leveraging sharding in the design of scalable replication protocols. In *Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC '13*, New York, NY, USA, 2013. Association for Computing Machinery. URL: <https://doi.org/10.1145/2523616.2523623>, doi:10.1145/2523616.2523623.
- [3] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, December 2002. URL: <http://doi.acm.org/10.1145/844128.844130>, doi:10.1145/844128.844130.
- [4] Ramnathan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Fault-tolerance, fast and slow: Exploiting failure asynchrony in distributed systems. In

- 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 390–408, Carlsbad, CA, October 2018. USENIX Association. URL: <https://www.usenix.org/conference/osdi18/presentation/alagappan>.
- [5] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 51–68, Carlsbad, CA, October 2018. USENIX Association. URL: <https://www.usenix.org/conference/osdi18/presentation/alquraan>.
- [6] Peter Bailis and Kyle Kingsbury. The network is reliable. *Commun. ACM*, 57(9):48–55, September 2014. URL: <http://doi.acm.org/10.1145/2643130>, doi:10.1145/2643130.
- [7] Philip A Bernstein and Sudipto Das. Rethinking eventual consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 923–928. ACM, 2013.
- [8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook’s distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013. USENIX. URL: <https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson>.
- [9] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI ’06*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1298455.1298487>.
- [10] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2, NSDI’05*, pages 15–28, Berkeley, CA, USA, 2005. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1251203.1251205>.
- [11] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation, OSDI ’99*, pages 173–186. USENIX, 1999. URL: <http://pmg.csail.mit.edu/papers/osdi99.pdf>.
- [12] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing, PODC ’07*, pages 398–407, New York, NY, USA, 2007. ACM. URL: <http://doi.acm.org/10.1145/1281100.1281103>, doi:10.1145/1281100.1281103.
- [13] Ang Chen, Hanjun Xiao, Andreas Haeberlen, and Linh Thi Xuan Phan. Fault tolerance and the five-second rule. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association. URL: <https://www.usenix.org/conference/hotos15/workshop-program/presentation/chen>.
- [14] Cloud Native Computing Foundation. etcd: A distributed, reliable key-value store for the most critical data of a distributed system, 2019. URL: <https://etcd.io/>.
- [15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google’s globally-distributed database. In *OSDI*, 2012.
- [16] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a dht for low latency and high throughput. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1, NSDI’04*, page 7, USA, 2004. USENIX Association.
- [17] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013. URL: <http://doi.acm.org/10.1145/2408776.2408794>, doi:10.1145/2408776.2408794.
- [18] Giuseppe DeCandia, Deniz Hastorun, Madan Jambani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP ’07*, pages 205–220, New York, NY, USA, 2007. ACM. URL: <http://doi.acm.org/10.1145/1294261.1294281>, doi:10.1145/1294261.1294281.

- [19] Tobias Distler, Frank Fischer, Rüdiger Kapitza, and Siqi Ling. Enhancing coordination in cloud infrastructures with an extendable coordination service. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management, SDMM '12*, pages 1:1–1:6, New York, NY, USA, 2012. ACM. URL: <http://doi.acm.org/10.1145/2405186.2405187>, doi:10.1145/2405186.2405187.
- [20] John (JD) Douceur. The sybil attack. In *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, January 2002. URL: <https://www.microsoft.com/en-us/research/publication/the-sybil-attack/>.
- [21] MV Fernando, Paulo Esteves, Christian Esteve, et al. Software-defined networking: A comprehensive survey. *PROCEEDINGS OF THE IEEE*, 2015.
- [22] Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. URL: <http://doi.acm.org/10.1145/564585.564601>, doi:10.1145/564585.564601.
- [23] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11*, page 15–28, New York, NY, USA, 2011. Association for Computing Machinery. URL: <https://doi.org/10.1145/2043556.2043559>, doi:10.1145/2043556.2043559.
- [24] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991. URL: <https://doi.org/10.1145/103162.103163>, doi:10.1145/103162.103163.
- [25] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, SOSP '89*, pages 202–210, New York, NY, USA, 1989. ACM. URL: <http://doi.acm.org/10.1145/74850.74870>, doi:10.1145/74850.74870.
- [26] Jim Gray. A census of tandem system availability between 1985 and 1990. Technical Report TR-90.1, HP Labs, 1990.
- [27] Shay Gueron and Yehuda Lindell. Gcm-siv: Full nonce misuse-resistant authenticated encryption at under one cycle per byte. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 109–119. ACM, 2015.
- [28] Neil J. Gunther. A general theory of computational scalability based on rational functions. *CoRR*, abs/0808.1431, 2008. URL: <http://arxiv.org/abs/0808.1431>, arXiv:0808.1431.
- [29] James Lee Hafner and KK Rao. Notes on reliability models for non-mds erasure codes. Technical report, IBM Research Division, 2006.
- [30] James Hamilton. Aws re:invent 2016: Tuesday night live with james hamilton. URL: <https://www.youtube.com/watch?v=AyOAJFNPAaA>.
- [31] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. URL: <http://doi.acm.org/10.1145/78969.78972>, doi:10.1145/78969.78972.
- [32] Yury Izrailevsky and Ariel Tseitlin. The Netflix Simian Army, 2011. URL: <https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116>.
- [33] Kyle Kingsbury. Jepsen, 2019. URL: <https://jepsen.io/>.
- [34] John Kubiawicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IX*, pages 190–201, New York, NY, USA, 2000. ACM. URL: <http://doi.acm.org/10.1145/378993.379239>, doi:10.1145/378993.379239.
- [35] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001), pages 51–58, December 2001. URL: <https://www.microsoft.com/en-us/research/publication/paxos-made-simple/>.
- [36] Leslie Lamport. Who builds a house without drawing blueprints? *Communications of the ACM*, 58(4):38–41, 2015.
- [37] Butler W. Lampson. How to build a highly available system using consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms, WDAG '96*, pages 1–17, London, UK, 1996. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=645953.675640>.

- [38] John MacCormick, Nick Murphy, Marc Najork, Chandu Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Symposium on Operating System Design and Implementation (OSDI)*, pages 105–120. USENIX, December 2004.
- [39] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardouff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.
- [40] Brian M. Oki. *Viewstamped Replication for Highly Available Distributed Systems*. PhD thesis, MIT, 8 1988.
- [41] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC ’88, pages 8–17, New York, NY, USA, 1988. ACM. URL: <http://doi.acm.org/10.1145/62546.62549>, doi:10.1145/62546.62549.
- [42] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC’14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=2643634.2643666>.
- [43] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’88, pages 109–116, New York, NY, USA, 1988. ACM. URL: <http://doi.acm.org/10.1145/50202.50214>, doi:10.1145/50202.50214.
- [44] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST ’07, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1267903.1267905>.
- [45] J. Postel. Dod standard transmission control protocol. RFC 761, RFC Editor, January 1980.
- [46] David R. Richardson. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. URL: <https://aws.amazon.com/message/65648/>.
- [47] Antony I. T. Rowstron and Peter Druschel. Passtry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Middleware ’01*, pages 329–350, London, UK, UK, 2001. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=646591.697650>.
- [48] Nuno Santos and Andre Schiper. Optimizing Paxos with batching and pipelining. *Theoretical Computer Science*, 496:170–183, 2013. doi:10.1016/j.tcs.2012.10.002.
- [49] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. URL: <http://doi.acm.org/10.1145/98163.98167>, doi:10.1145/98163.98167.
- [50] Bianca Schroeder and Garth Gibson. A large-scale study of failures in high-performance-computing systems. Technical Report CMU-PDL-05-112, Carnegie Mellon University, 2005.
- [51] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST ’07, Berkeley, CA, USA, 2007. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1267903.1267904>.
- [52] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM ’01, pages 149–160, New York, NY, USA, 2001. ACM. URL: <http://doi.acm.org/10.1145/383059.383071>, doi:10.1145/383059.383071.
- [53] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149. IEEE, 1994.
- [54] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI’04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1251254.1251261>.
- [55] Peter Vossball. Aws re:invent 2018: How aws minimizes the blast radius of failures. URL: <https://www.youtube.com/watch?v=swQbA4zub20>.

- [56] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '99, pages 54–66, London, UK, UK, 1999. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=646704.702012>.
- [57] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, Broomfield, CO, 2014. USENIX Association. URL: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan>.