# Limix: Limiting Lamport Exposure to Distant Failures in Globally-Managed Distributed Systems

**Cristina Băsescu**[1]
Digital Asset GmbH, Zürich, Switzerland

**Georgia Fragkouli**
ETH Zürich, Switzerland

**Enis Ceyhun Alp**
Parity Technologies, London, UK

**Michael F. Nowlan**
Reflect, Somerville, MA, US

**Jose M. Faleiro**
Stealth Startup, San Francisco, CA, US

**Gaylor Bosson**
Taurus Group SA, Lausanne, Switzerland

**Kelong Cong**
Zama AI, Paris, France

**Pierluca Borsò-Tan**
EPFL, Lausanne, Switzerland

**Vero Estrada-Galiñanes**
The DECENT Lab, Dübendorf, Switzerland

**Bryan Ford**
EPFL, Lausanne, Switzerland

## Abstract

Globalized computing infrastructures offer the convenience and elasticity of globally managed objects and services, but lack the resilience to distant failures that localized infrastructures such as private clouds provide. Providing *both* global management and resilience to distant failures, however, poses a fundamental problem for configuration services: How to discover a possibly migratory, strongly-consistent service/object in a globalized infrastructure without dependencies on globalized state? Limix is the first metadata configuration service that addresses this problem. With Limix, global strongly-consistent data-plane services and objects are insulated from remote gray failures by ensuring that the definitive, strongly-consistent metadata for any object is always confined to the same region as the object itself. Limix guarantees availability bounds: any user can continue accessing any strongly consistent object that matters to the user located at distance $\Delta$ away, insulated from failures outside a small multiple of $\Delta$. We built a Limix metadata service *based on the key-value interface of* CockroachDB. Our experiments on Internet-like networks and on AWS, using realistic trace-driven workloads, show that Limix enables global management and significantly improves availability over the state-of-the-art.

**2012 ACM Subject Classification** Computer systems organization → Distributed architectures

**Keywords and phrases** Distributed systems, Availability, Fault tolerance, Strong-consistency, Coordination systems, Lamport exposure

**Digital Object Identifier** 10.4230/OASIcs.NINeS.2026.3

**Supplementary Material** *Software (Implementation and Evaluation of Limix)*: `https://github.com/dedis/limix` archived at `swh:1:dir:2c8a39b2bf7288ca123ed287a428bbce9962a566`

**Funding** *Cristina Băsescu*: Cristina Basescu's work was partly funded by EU H2020 grant agreement 825377 (UNICORE), Handshake Development Inc. under grant no. 536892, and Oracle under grant no. 538003.

**Acknowledgements** We thank the NINeS anonymous reviewers and our shepherd, Aurojit Panda, for their valuable feedback. We also thank Pantea Karimi for her contributions on earlier versions of this paper, and David Lazar, Kirill Nikitin, and the members of the DEDIS lab at EPFL for the productive discussions and helpful suggestions that helped shape this paper.

[1] Corresponding author. This research was conducted while Cristina was affiliated with EPFL.





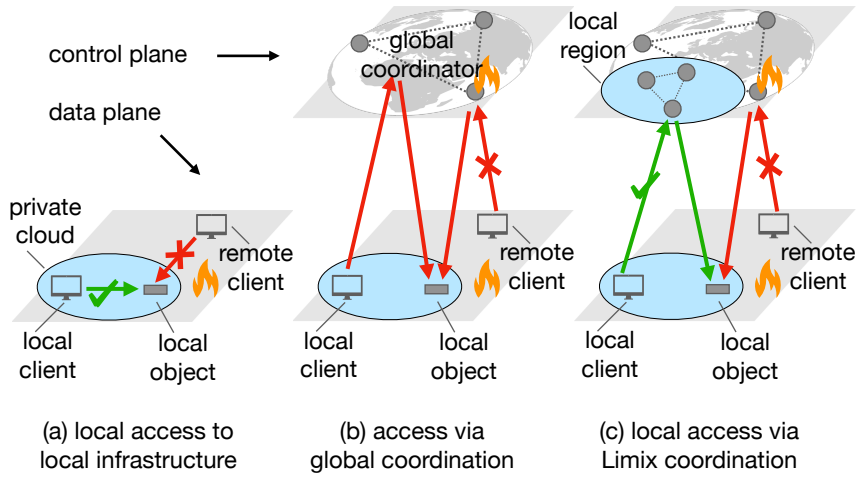



**Figure 1** Lamport exposure [7] of: traditional (a) local or (b) global infrastructure; (c) Limix global infrastructure.

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
>
> Leslie Lamport

> All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection.
>
> David Wheeler

## 1 Introduction

Organizations today face a choice between *localized* and *globalized* computing infrastructure, each alternative carrying important tradeoffs. Localized infrastructure hosted at the organization's own site(s), such as private clouds, carry higher internal management burdens but offer greater local autonomy, resilience to distant failures beyond the organization's control, and can be necessary to satisfy data privacy or digital sovereignty concerns. Globalized infrastructure such as public clouds, in contrast, offer many global management benefits: e.g., the convenience of instantiating objects or services on demand without worrying about their location, maximum elasticity in provisioning and adapting to changes in load, and the ability to migrate existing data and services without having to interrupt access or change their names.

Is it possible to achieve the local autonomy, failure resilience, and digital sovereignty benefits of localized infrastructure, together with the global management benefits of today's public clouds? Achieving the best of both worlds appears fundamentally hard, in part

because this choice boils down to a basic indirection conundrum. For clients to find and access localized infrastructure such as private cloud services, we can simply embed a locally-scoped or already resolved location directly into the identifiers of objects and services being accessed. Accesses by local clients to local objects can be simple, robust, and insulated from remote failures or network partitions outside the relevant domain, as illustrated in Figure 1a. But these objects are then "fixed" and cannot be migrated or managed globally without changing their names.

Global management, in contrast, requires "another level of indirection": typically a distributed coordination or metadata service, allowing clients located anywhere to discover the location and status of any object or service of interest. These services, however, expose clients to global *gray failures* [21] such as network partitions [4, 17, 20], misconfigurations [29, 31], or cascading failures [27, 6, 4, 20], far beyond of the organization's geographic locality or domain of control – i.e., exposure to "the failure of a computer you didn't even know existed." This global failure exposure usually applies even when both the client and the target data or service are localized to the same network or region and have connectivity in the underlying network [6, 4]. Even if the target data might be weakly consistent [26, 12, 16], metadata is usually strongly consistent for many reasons [24, 5] such as correct liveness determination, access control, and accounting. Dependence on globally geo-replicated metadata, however, can prevent even local clients from accessing local data if a majority of metadata replicas fail or become unreachable, as illustrated in Figure 1b.

Cell-based configuration services like Physalia [6] improve failure resilience for users within the same cell. These provider-managed cells, however, do not offer users direct transparency into or control over each user's *Lamport exposure* [7] – the set of infrastructure components whose failure could affect the user – discussed in Section 2.1. For example, a write operation in a cell followed by a read operation from another cell requires data discovery across cells. But discovery caches across cells do not limit exposure, analogous to Figure 1b. Thus, while cell-based configuration services limit the effect of failures to within the cell of the failure, they offer no guarantees for user activity that crosses cell boundaries.

To address this conundrum we introduce Limix, the first distributed coordination architecture that enables global management while also guaranteeing that localized accesses within a region of interest are insulated from global failures beyond that region, as illustrated in Figure 1c. Limix ensures that the definitive, strongly-consistent metadata for any data-plane object or service is always collocated in the same region as the object itself. Metadata in Limix thus enjoys a *fate-sharing* relationship [11] with both the target object and with any local clients accessing the object from within the same region – such as within an organization's own internal network, or within a relevant geopolitical domain such as a country. Metadata remains strongly-consistent and geo-replicated, but Limix confines the definitive replicas of an object's strongly-consistent metadata to the same region as the object itself.

**Challenges and Contributions.** Limix's design decisions for practical global manageability and resilience become apparent when we target applications where the locations from which data-plane items are accessed change dynamically. Data stores and locality management services with dynamic data access locality already migrate strongly-consistent data close to users [24, 5]. Limix ensures that users can continue accessing such nearby items under remote failures even during migration, while preserving strongly-consistent access.

Constraining the placement of strongly-consistent metadata in localities creates the further efficiency and scalability challenge of enabling any clients outside an object's current region to find the object without incurring the costs of either replicating all location information proactively across all regions, or potentially having to search all regions during any metadata query. Limix builds on techniques from compact graph summarization theory [32, 33] to limit

the bandwidth and processing costs of these global searches to a small multiple of the baseline cost of querying a single global metadata service. The metadata-access costs of Limix's failure insulation is thus only about 2× in the common case of an object administratively localized to a single region. Since metadata query costs usually represent only a small fraction of the total "end-to-end" costs of accessing most data-plane services, a 2× metadata query cost increase is insignificant overall to most applications, and is much lower than the $N\times$ metadata cost increase that cell-based architectures with $N$ distributed discovery services (or the proactive replication of location hints across all $N$ regions) would otherwise incur. Further, Limix ensures by design that not only availability but also metadata access latencies observed by local clients are insulated from global outages or slowdowns, and that they closely reflect the best local communication latencies available on the underlying network.

To evaluate Limix's applicability and performance, we prototyped a Limix configuration service for an exposure-limiting key/value store. For metadata/configuration storage, our prototypes use CockroachDB [24], a widely-used, strongly-consistent distributed data store. Our experiments running realistic workloads based on metropolitan traffic traces on AWS, and on a testbed simulating realistic scenarios, show that Limix outperforms Physalia's availability during reconfigurations while providing strong availability guarantees. Our experiments further explore the tradeoffs between Limix's overheads and availability guarantees: at scale, the dynamic load overhead is logarithmic in the number of sites and network width.

In summary, the contributions of this paper are:

- The design of Limix, the first distributed metadata coordination service that enables global management while protecting local accesses from distant failures.
- An autozoning scheme ensuring by design that a user accessing any data at a distance $\Delta$ away is protected from all failures occurring beyond a small multiple of $\Delta$.
- A theoretical analysis of the load per site in a Limix deployment with autozoning.
- A prototype implementation of Limix and Physalia on top of CockroachDB with a comparative evaluation.

## 2 Background and Motivation

This section gives the necessary background for a strongly-consistent configuration service like Limix. We first explain that the CAP theorem imposes restrictions on the availability Limix can achieve, and that Limix does not conflict with the CAP theorem when prioritizing availability of local accesses. Second, we focus on Limix's choice to prioritize user-perceived availability, for the data-plane objects that matter to users. Reducing the Lamport exposure, which is the user-centric viewpoint of Limix, contrasts to Physalia's provider-centric viewpoint, i.e., blast radius. Our discussions with risk-sensitive customers share Limix's viewpoint. Finally, we argue that access locality is prevalent in globalized applications, and thus, Limix's focus on shielding local user activity leads to a sizeable increase in user-perceived availability.

### 2.1 Lamport exposure and blast radius

We informally define the *Lamport exposure* of any given user $U$ as the set of computing infrastructure – i.e., every "computer $U$ didn't even know existed" – that "can render $U$'s own computer unusable" [7]. Lamport exposure is thus meaningful only with respect to the activities of some user $U$.

Limix seeks to place a strong bound or "shield" on the Lamport exposure of any user $U$ whenever $U$ accesses data or services that are *local*, i.e., close to $U$ by any suitable distance metric. We may define locality based on administrative boundaries such as those of an organization or a country, or via metrics such as round-trip time (RTT). The availability

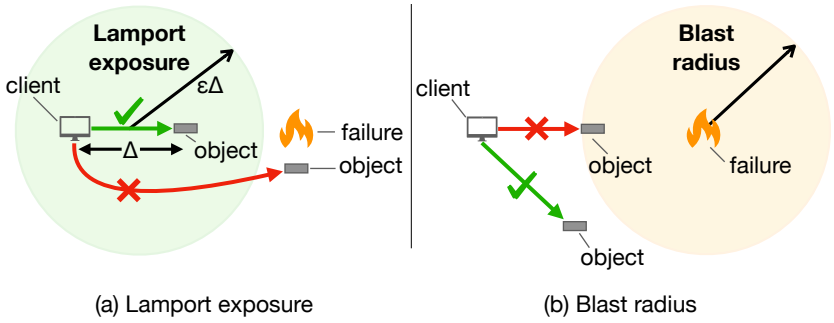



**Figure 2** Lamport exposure vs. blast radius.

of $U$'s local accesses should be unaffected by remote failures and partitions beyond Limix's Lamport exposure shield: not just by individual server failures but also by network partitions and *gray failures*, or partial failures that can cascade into correlated failures and partial partitions. Gray failures have a multitude of causes including malfunctioning switches and software bugs that partially drop traffic or prevent simplex communication [4, 27, 17, 20]. Figure 2a illustrates a Lamport exposure bound shielding a user's accesses to an object at distance $\Delta$ from failures beyond a distance $\epsilon\Delta$, for some small factor $\epsilon \geq 1$ (ideally 1, but this may be unachievable).

**Locality matters.** By limiting Lamport exposure, Limix seeks to offer availability guarantees to users for the data and services that matter most to them, which are often local. It is already common for data stores to migrate data close to users to improve performance [24, 5], even without guaranteeing availability. Privacy and sovereignty considerations often motivate organizations or governments to require a user's "data at rest" to remain within the user's country, or a region such as the EU – even if these policies cannot currently ensure that this data will *remain accessible* despite outages beyond the relevant borders. Similar considerations motivate many organizations to confine their most-critical data and applications to local infrastructure such as private clouds, giving up the benefits of global management. In general, people more willingly trust organizations and services perceived to be more local [28, 13]. We hope that Limix might enable providers to offer services to more locality- and sovereignty-conscious users who might currently avoid globally-managed infrastructure entirely.

In contrast to Limix, Physalia [6] is a cell-based architecture that aims for higher availability by limiting the *blast radius* of a failure to each cell. Blast radius represents the system components and objects that could be affected by the propagation of a failure or partition (Figure 2b). Blast radius is thus a complementary concept focused (or "centered") on the location of a *failure*, rather than on the location of a *user* potentially affected by it. From an infrastructure provider's perspective, reducing blast radius can reduce the number of users affected by any single failure. Being focused on the locations of failures and heavily dependent on internal details of the provider's infrastructure, however, blast radius does not offer obvious guarantees that appear directly meaningful to users.

## 2.2 Coordination and the CAP theorem

Strongly-consistent coordination systems (Zookeeper [22], etcd [3]) are an essential building block for large scale distributed applications. Distributed applications replicate their state in order to enhance resiliency to failures, and to decrease latency through proximity to

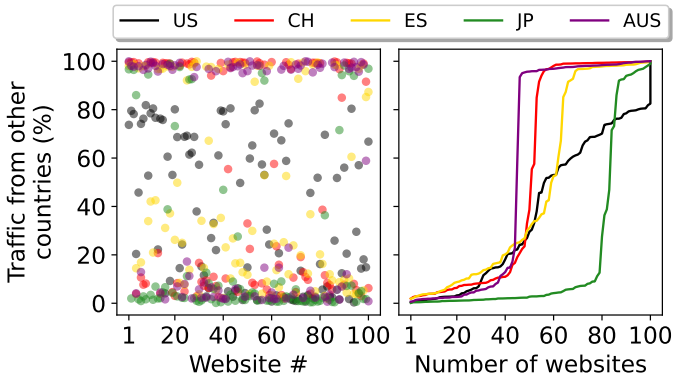



**Figure 3** Prevalence of access locality. Top 100 websites in 5 countries and their traffic percentage from other countries (individual websites and CDF).

clients. But an unsought side-effect is the need to coordinate these replicas. Hence the need for coordination services: These provide a basic set of operations – such as liveness determination, correct identification of the replicas storing a data item, lease holders, access control, accounting, etc. Because these functions need to be *correct*, coordination services implement consensus among the configuration replicas, ensuring strong consistency of the configuration.

Coordination systems often are not critical to application availability until failures occur. Applications routinely bypass the coordination system for configuration reads using leases [8, 24]. During partitions or failures, however, the data plane cannot bypass the configuration system, because it needs to reconfigure its data replicas. This is when the configuration system becomes critical to application availability and performance. Reconfiguration requires strongly-consistent configuration writes to agree on the new configuration. Until reconfiguration completes, the data-plane may be partially (e.g., operate in read-only mode only) or fully unavailable.

The requirements of configuration systems to be strongly consistent and available seem to conflict with the CAP theorem [15]: under partitions, a strongly-consistent (configuration) system cannot remain available (on both sides of a partition). However, Limix does not conflict with the CAP theorem when prioritizing availability of local accesses. Remote users may not be able to access remote data during partitions, but local users can, without breaking strong consistency, and in many cases as described above, local data is what interests users. For this reason, Limix collocates metadata with its data.

To decrease the risk of being affected by remote gray failures, Brooker et al. [6] suggest many smaller configuration service deployments instead of a network-wide "monolith". Deploying several configuration services instead of a single globalized deployment is one of the principles that Limix also applies. However, as opposed to deploying disjoint cells, Limix creates overlapping, redundant configuration service deployments, organized to provide availability guarantees for any user accessing any object or service.

## 2.3 Risk-sensitive customer perceptions

Recent discussions we had with globalized infrastructure customers – a large company and a non-profit organization, both international – informally confirmed to us their need for services that are globally-managed while providing availability guarantees in the face of

distant failures. Applications increasingly move to globalized infrastructures such as public clouds to benefit from the elasticity properties and lower management effort. Despite the flexibility of including custom clauses in the SLA, some customers are reluctant to trust these infrastructures: Because the number of reported outages is still uncomfortably high, they fear the reputation risk. One customer suggested that a good design that guarantees availability should recommend itself before getting to the SLAs. Finally, customers would consider leveraging a service that limits Lamport exposure – one referring to this as "the holy grail" – because its guarantees are well-understood and more meaningful compared to simply limiting the "blast radius" of a failure.

### 2.4 Estimating access locality

We observe that many applications and services exhibit bimodal access locality. A high proportion of accesses are by users mostly in a given country or other region representing the application's primary customer base or target audience. Other users of these same applications and services, however, tend to be globally distributed, accessing the service from anywhere (e.g., roaming employees or expats). Thus, applications must efficiently support global accessibility for global users, while prioritizing maximum availability and performance for local users representing the most critical target customers.

To test this bimodal-access hypothesis, we use the top 100 websites of five OECD countries (namely United States (US), Switzerland (CH), Spain (ES), Japan (JP) and Australia (AU)) as rough proxies for applications, and examine the access distributions they exhibit. Since these are the most visited websites in their respective countries, we conjecture that they have a strong local presence. In Figure 3, we show that these websites also have a global presence, as 17–56% (JP–AU) of the websites receive at least 50% of their traffic from outside of that country. The results of our simple study does not make any assumptions about the consistency models used by the websites, but it suggests that locally-prevalent applications are indeed globally relevant too.

## 3 Setting and Goals

In this section we discuss the main system components, our assumptions and our goals. Limix involves the following concepts:

**Items.** Limix is a configuration service for existing data-planes, such as a key-value store as in our prototype (Section 6). We define as *items* the access targets, e.g., key-value pairs. Limix manages the configuration of the data plane, which enables lookup and reconfiguration of data-plane item replicas. Limix interfaces with the existing data store to react to item creation and migration/reconfiguration by creating and migrating/changing the configuration.

**Sites.** Sites are the nodes that deploy Limix, which we assume to be connected through a network. For example, sites can be data centers, which is the use-case we explore in our prototype and evaluation. Limix can seamlessly make use of sites under the control of different providers.

**Clients requests.** Clients interact with Limix by submitting item lookup requests at a site, and Limix's availability guarantees apply once the client request reaches a site. Limix does not improve last mile availability, e.g., if the client cannot reach any site. We make no assumption about which site clients choose: Clients can submit lookup requests to any

site they wish, e.g., because they change their location, or for other reasons, and the lookup responses are always correct. However, Limix provides guarantees for the client w.r.t. the location of the client-chosen site. Therefore, clients choosing nearby sites based on RTTs would be sensible.

**Zones, authoritative zone, definitive replicas.** Limix provides availability guarantees for any client looking up an item at the granularity of *zones*. Limix deploys zones along sites, and a zone encompasses "the set of distributed system components, including servers, routers, network links, etc., that a user depends on for availability" [7] when looking up items. The data plane is zone-agnostic and does not require zone knowledge. But Limix tracks data-plane item location w.r.t. zones, in order to collocate configuration with its data. Specifically, Limix defines an item's location as the item's *authoritative zone*: this is the smallest zone containing a quorum of the item's data plane replicas. Limix does not constrain or change the location of *data replicas*; it merely tracks them. Then, in the same zone, Limix maintains the most recent state of the item's configuration, i.e., the *definitive configuration/metadata replicas* for that item. In contrast, non-authoritative zones may have only eventually-consistent configuration replicas for those items.

**Goals.** Limix has the following objectives:

- **Availability guarantees.** Provide strong availability guarantees that might be legally or contractually mandated to hold at all times, even during item migration.
- **Simultaneous constraints.** Satisfy simultaneous sovereignty and locality constraints, e.g., an Italy constraint is more restrictive in placement, while an EU constraint protects a larger set of users.
- **Load balancing.** Spread workload for item lookup e.g., avoid overloading small zones with global accesses. A user imposes load only on the zones the user's site is in.
- **Dynamic data plane.** Enable dynamic data planes to migrate data routinely, without restricting them with e.g., static partitions of data across regions.
- **Strong consistency.** Enforce that item lookup returns strongly-consistent items, or be unavailable for that item if the latest item version is not reachable.
- **Autozoning.** Provide an automatic zoning capability, which enforces locality constraints for all users, the vast majority of whom just "want things to work." We desire reasonable but fully-automatic risk-limiting policies requiring no specific understanding of the workload or administrative effort.

By collocating metadata replicas with the data replicas (i.e., in the authoritative zone), Limix ensures availability guarantees and strong consistency: Any user who could access the data can access the strongly-consistent metadata (i.e., definitive replicas) and locate the data. By replicating the definitive metadata replicas in all zones that contain the authoritative zone, Limix ensures load balancing and simultaneous availability guarantees at different localities: clients in any unpartitioned zone containing a quorum of item replicas can locate and access the item, despite failures outside the zone.

## 4 Design of Limix

This section outlines Limix's design in a step-by-step fashion for clarity. We first list Limix's challenges, then introduce Limix's per-zone configuration service, and explain how it limits maintenance loads on local zones. We then address the problem of satisfying multiple simultaneous exposure-limiting constraints on one item, ensuring that lookup replies follow data-plane consistency, and handling item migration.

## 4.1 Challenges

The goal of local availability under global management, coupled with the requirement of strong consistency for metadata, imposes key challenges on Limix's design.

**Consistency.** Because data and services must still be movable and accessible from anywhere under normal conditions, Limix must enable clients anywhere to locate a data item's current definitive state globally, while insulating local clients from global failures. This requirement implies replicating location information simultaneously across global and local deployment zones, which in turn exacerbates consistency challenges (Sections 4.2 and 4.3).

**Load balancing.** We must ensure that control-plane queries about globally-popular data do not overload a small, lightly-provisioned zone it may be located in. Limix addresses this challenge by systematically ensuring that each zone, local or global, serves only clients querying the service from *within the same zone*, and can therefore spread the access workload without risking overload from external queries (Section 4.4).

**Simultaneous item constraints.** Data plane items may have to satisfy more than one locality or sovereignty constraint – such as that local clients in Germany be insulated from failures outside Germany, *and* that all clients in the EU be insulated from failures outside the EU. This goal requires that Limix allows state replication across multiple *overlapping* zones (Section 4.5).

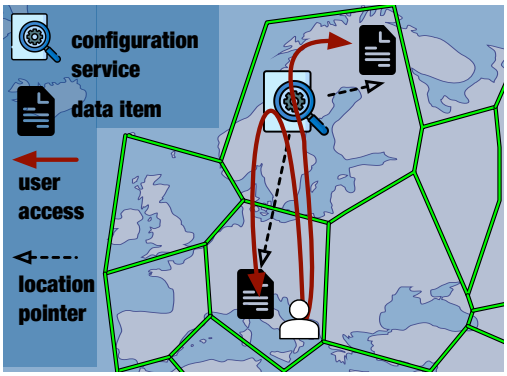
**Item migration.** Limix must maintain both strong consistency and local availability even during object migration: ensuring, for example, that data migrating from Germany to France remains immune to failures outside the EU even during the transition. To address this challenge, Limix uses a multi-phase process to migrate the data's definitive state while maintaining eventually-consistent location hints in larger zones beyond the data's origin and destination (Section 4.6).

## 4.2 Item lookup

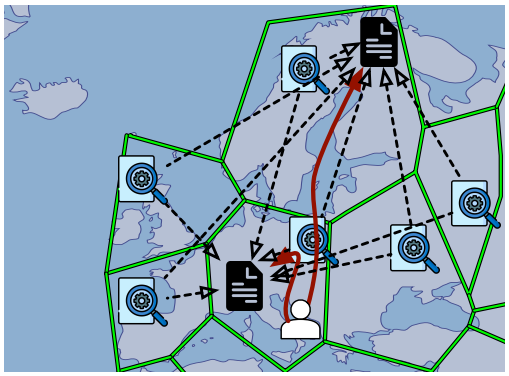
For zones to act collectively as a unified system, Limix needs to enable clients to find data that can be located anywhere, regardless of the client's zone. However, robust and efficient item lookup is challenging. Figure 4a illustrates a straightforward but inadequate approach, relying on a central service to store the configuration for item lookup. This service increases the client's exposure beyond the perimeter of the client's and data's common zone. The single zone may also become overloaded with requests from all zones.

Of course, this strawman could be easily made scalable by distributing the configuration service across many/all zones, using standard techniques such as consistent hashing of keys. However, consistent hashing still increases a site's Lamport exposure beyond the zone boundaries. Consider a client requesting data without having the data location cached. To resolve the location, the client might need to query configuration service sites in zones different from the zone holding the requested data. Partitions might prevent the client from reading the data location, even though they might not isolate the client from reaching the data itself. The problem with this approach is that data and corresponding metadata are not collocated in the same zone, and hence lack fate sharing [11].

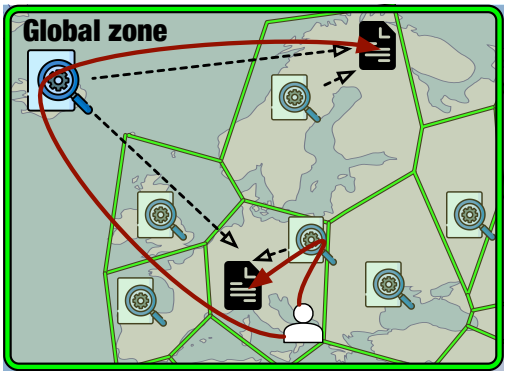
Limix thus needs to ensure that a client in a given zone can always find an item within the same zone using *only* resources within that zone. Efficiently collocating data and metadata in the same zone so that they have the same Lamport exposure represents the first challenge for Limix, which we address by having a distributed configuration service *per zone*.



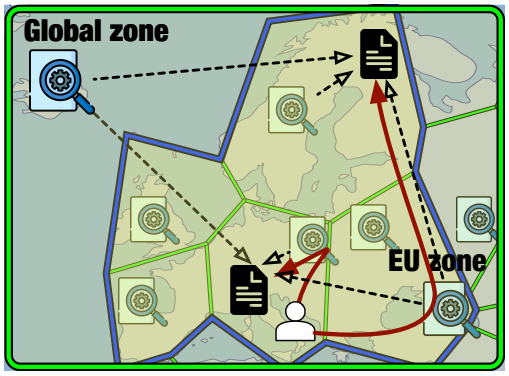

**(a)** Global CS, holds pointers to all items.



**(b)** Full CS per zone, each maintaining pointers to all items.




**(c)** Local CS per zone, each holding pointers *only* to local items.




**(d)** Limix CS with overlapping zones, holds pointers to local items.

**Figure 4** Strawmen for the Configuration Service (CS) and its management of pointers to items.

## 4.3 Configuration service consistency

As a next strawman addressing the item lookup challenge above, we could replicate *all* lookup pointers in *all* zones, as depicted in Figure 4b. However, this strawman introduces a consistency challenge: Because all zones' configuration services store pointers to all items, when an item is deleted or migrated, regardless of the zone where the item resided, all configuration services in all zones should be updated with the new pointer. If we required strongly consistent state for all zones' configuration services, we would increase an item's exposure to all zones, which is undesirable.

Limix addresses this exposure challenge as follows. Each zone's configuration service stores strongly consistent pointers only for the items inside the zone. If an item's configuration changes, only the configuration service in the item's zone needs to update the item's definitive configuration (i.e., location metadata) (see Section 3) immediately, on the critical path. Other zones' configuration services may update their metadata lazily with eventual consistency, outside the critical path of the item reconfiguration.

With this approach, each zone has its own configuration service that stores "location hints" for where an item was last known to be located. But this strawman invites a second challenge: How do clients locate items given that configuration services might (temporarily) store outdated pointers? We distinguish two causes of outdated pointers. The first reason is item migration, when Limix updates pointers outside the critical path. Could a client be

**Algorithm 1** Item lookup that the client calls at *site*.

```
1: procedure ItemLookup(site, key)
2:     for zone ∈ GetZones(site) in parallel do
3:         // Recursively follow pointers to authoritative zone
4:         while ! IsAuthoritative(zone, key) do
5:             zone ← ReadPointer(zone, key)
6:         return zone
```

unable to find an otherwise reachable item when the item migrates? Limix addresses the challenge by temporarily storing a *Permanently moved to* marker at the item's old location. On encountering this marker, a client follows it to the new location. Limix prevents long indirection paths by eventually updating all pointers, after which it deletes the marker. The client stops following pointers when it reaches the item's authoritative zone; Limix coordination ensures there is a single authoritative zone per item (Section 4.5).

The second reason for items to be outdated is during partitions. If Limix cannot reach some zones' configuration services, pointers will be stale. The main challenge is to ensure that clients do not return stale items because of stale pointers, which would break strong consistency. Limix clients rely on authoritative zone indicators, as explained above, to decide whether the pointers point to the most recent item version. However, there is one remaining issue when partitions heal and several migrations are in place: Pointer updates might arrive out-of-order, causing an old update to overwrite a newer one. We use versioning for pointers to avoid this situation. Every pointer update increases the pointer version and a pointer update occurs only if the update has a higher version than the existing pointer (Section 4.6). The update makes use of the compare-and-swap primitive typically offered by strongly-consistent KV stores.

## 4.4 Lookup load on (small, local) zones

The above strawman invites the question: What is the load on each zone's configuration service? Consider updating the configuration service after an item insertion or migration. Either the destination zone could push the new item location to all zones, or the client's zone could pull the item location on demand. Both approaches incur $O(n)$ load and communication overhead *per client request* for $n$ zones.

Limix instead spreads the lookup loads and limits query burden on small zones by organizing a default overlapping global zone. In our next strawman illustrated in Figure 4c, local zones store the location only for their local items and serve lookup requests only from local sites. In contrast, the global zone serves as the backup reference point, whose globally-distributed configuration service knows any item's location. Every zone propagates location updates to the global zone.

A client queries only its own local configuration service and the global one, without overloading other small zones. Thus, instead of $O(n)$, the overhead per update becomes $O(1)$. The global configuration service must service load from all $O(n)$ zones, but it has server capacity distributed across all $O(n)$ zones among which to share that load. Location updates propagate only eventually, off the critical path, to limit the source zone's exposure to outer failures.

## 4.5 Item placement and zone overlap

Aside from the global zone, we assumed so far that local zones are disjoint. This assumption has a significant limitation, however: it cannot support *simultaneous* exposure-limiting policies, which may apply by law or contractual obligation. An item located in Germany may need to be accessible by clients in Germany with Lamport exposure limited to sites within Germany, *and* ensure that any client in the EU can access the same item with exposure limited to the EU. Yet, with the above strawman configuration service, EU clients outside Germany must query the global configuration service, yielding global (not EU) exposure.

We address this problem by allowing local zones to overlap. All zones have a configuration service, and every zone propagates location updates to larger overlapping zones, up to the global zone. The global zone still acts as a master reference, holding pointers to all items. Update overhead increases slightly compared to the single global zone, from $O(1)$ to $O(v)$, where $v$ is the maximum overlap depth. However, this approach limits Lamport exposure to smallest zone containing both the item and the client accessing it. Location updates propagate only eventually, outside the critical path, to limit the source zone's exposure to failures outside its borders. Figure 4d illustrates the final configuration service.

**Item lookup revisited.** We review item lookup in the final Limix configuration service architecture. We begin with a simplified scenario where items are immutable, do not change location, and the lookup service pointers of all zones are up-to-date. In this scenario, we do not need to worry about pointer or item consistency. Section 4.6 addresses consistency and migration.

Alg. 1 describes item lookup. A client calls the lookup function on a site's coordinator, passing the item key as a parameter. The coordinator sends parallel lookup queries to the configuration services of all site's zones. The client follows the pointers in the responses until it finds the authoritative zone, or returns nil.

Why does item search not degrade the exposure guarantees? Each of the parallel lookup queries accesses the zone-private KV store, having dependencies only inside the zone. Other parallel searches might hang in case of partitions or slow performance. However, because each search executes and completes independently, parallel searches do not affect each other. Of these zones, the ones that reply first with a pointer chain leading to the authoritative zone determine the client's exposure for that item. If at least one such set of zones is partition-free, the client is *guaranteed* to find the item. Thus, the coordinator bounds a client's exposure to the smallest zone of the client that contains the item.

## 4.6 Lookup during item migration

The placement of items may need to change, for example due to client-perceived performance and load-balancing algorithms. Or this could be caused by a policy change, e.g., a new constraint on which existing zone(s) a particular item is allowed to be placed in or migrated to. Because items involve strongly-consistent state, a challenge is that Limix must maintain strong consistency for configuration during item's migration. Three questions arise at this point: (a) How can we ensure that the coordinator locates the latest version of an item, even during migration or partitions? (b) Given that clients could update data plane items anywhere in the system, how do we ensure another distant client, located in a different zone, finds a specific item? (c) How do we ensure item search does not degrade the exposure-limiting guarantees?

**Algorithm 2** Location update during item migration.

```
1: procedure UpdatePtrOnMigration(key, srcZ, dstZ)
2:     oldPtrVersion ← ReadPointer(srcZ, key).Version
3:     ptrVersion ← oldPtrVersion + 1
4:     CAS(srcZ, key, v = dstZ||ptrVersion, ptrVersion ≥ crt)
5:     CAS(dstZ, key, v = True, ptrVersion ≥ crt)
6:     do in parallel
7:         UpdatePtrOuterZ(srcZ, key, False, ptrVersion, >)
8:         UpdatePtrOuterZ(dstZ, key, dstZ, ptrVersion, ≥)
9: // Background garbage collection after updating all ptrs
10:    CAS(srcZ, key, v = nil, ptrVersion > crt)
11: procedure UpdatePtrOuterZ(zone, key, val, version, cmp)
12:    for outerZone ∈ getOuterZones(zone) in parallel do
13:        CAS(outerZone, key, val, cmp(version, crt))
```

Alg. 2 depicts item migration from the item's authoritative zone $sZone$ to the new authoritative zone $dZone$. When Limix receives a callback from the data store that an item is being migrated, it does the following: (0) Increment the item's most recent pointer version, by definition in the item's authoritative zone $sZone$ (lines 2-3). We use pointer compare-and-swap with versioning for the pointer updates below to avoid old pointers overwriting newer ones. (1) Update the pointer in the old zone with a forwarding reference, indicating the item is being migrated to the new zone and the old zone should no longer be considered authoritative (line 4); (2) After item migration, update the pointer in the new zone indicating migration is complete and the new zone is now authoritative, meaning that item is now usable at its new site (line 5); (3) In parallel and outside the critical item lookup path, update discovery service information in all zones for the item's old and new location (lines 6-8); (4) Garbage collection: The item's configuration may finally be deleted in the old zone (line 10).

The algorithm above guarantees that any client can locate any item as long as they share an unpartitioned zone, even during/after migration. After step 1, all clients following existing pointers for that item (using Alg. 1) that lead (directly or indirectly) to the former authoritative zone learn that the item is being migrated to the new zone. After step 2, when the item finishes migrating, all clients following existing pointers locate the new authoritative zone. All clients can now find the item, but possibly via a longer chain (e.g., if it moved from the UK to Italy, Spain clients may first follow the pointer to the UK, then to Italy). Lookup exposure *during migration* reflects the migration: The client's exposure for accessing the item is $Z_{Spain} \cup Z_{UK} \cup Z_{Italy}$. Eventually, when all pointers in step 3 finish updating, all clients in the system can locate the item by following a single pointer. Accordingly, the exposure *after migration* is $Z_{Spain} \cup Z_{Italy} = Z_{SW-Europe}$. Importantly, because pointer updates happen independently and in parallel, partitions outside Europe SW cannot disrupt the client from locating the item.

It is worth outlining the tradeoff between limiting exposure and the overhead of updating metadata during item migrations. In Limix, simultaneous migrations for the same item require updating the metadata pointers in all zones that contain the item before and after each migration step. The update overhead enables Limix to limit exposure of the item lookup during migration to the smallest unpartitioned zone containing the still in-flight migrations. In contrast, an architecture that maintains local configuration services per zone and additionally a global configuration, like in the strawman in Figure 4c, merely needs to update global zone pointers (akin to cache invalidation) and pointers of the migration zones. This reduces pointer-update overhead at the cost of global exposure during migrations.

## 5 Control Plane Zoning

Limix provides user-centric availability guarantees meaningful for users with respect to items they access. Limix can define zones as input *jurisdictions*, e.g., Germany, the EU, the World, which are useful when items are constrained by regulatory bounds, for example. However, sometimes administrative zoning policies may not be explicit. Or users may want formal guarantees on availability, alongside administrative policies.

This section presents one particular autozoning policy that limits Lamport exposure, and that can use any distance metric, such as network latency in our design. For *any user accessing any item*, autozoning limits the access's exposure – availability and performance – to a zone guaranteed to exist within a small RTT from the client and the item.

**RTT as exposure metric.** For creating auto-zones, Limix uses round-trip time (RTT) as the exposure metric. A zone's RTT diameter defines the zone's exposure: a lower RTT diameter gives a lower exposure to remote failures, thus higher availability. The intuition is that clients and items with a small pairwise RTT are likely geographically close [9], representing localized accesses for Limix. As RTTs increase, localities are less tight. RTT maps of sites deployed on cloud providers are stable, showing less than 6% month-to-month difference in median latency on Azure [18, 34]. During bootstrapping, Limix builds an inter-site RTT map: the sites measure their pair-wise RTTs, and then each pair of sites averages their link's RTT value so that the final RTT map is consistent across sites. Based on the RTT map, each site computes the auto-zones membership and Limix starts operating without any assumptions about the timing or location of partitions.

**Autozoning strawman.** To bound the exposure of *a particular client* to any item, we could simply build overlapping zones centered on the site that the particular client accesses. A naïve, non-scalable approach would be to build many concentric zones of slowly increasing radius. A more scalable aproach is to choose exponentially increasing zone radiuses, as depicted in Figure 5-Left. The tradeoff of exponential zone radii is less tight Lamport exposure bounds for the client accessing any item, at the scale benefit of building a number of zones logarithmic in network-wide RTT.

This simple strawman, however, only bounds exposure for clients accessing the system through sites located at the center of zones. If a client chooses a site with non-concentric zones (in line with Limix's goals of supporting dynamic access) then the client loses the tight exposure guarantee. Whereas, if the system simply deploys such zones centered around each site, then each site serves load from a prohibitive $O(N \log N)$ zones, where $N$ is the number of sites. We show next how to scalably build zones that bounds exposure for any client accessing any item through any site.

**Compact-graph approximations.** Autozoning builds on techniques from compact graph summarization theory [32, 33] to guarantee exposure for all clients and all objects, while optimizing the number of created zones, hence optimizing the system overhead. Autozoning has two goals: (1) Bounding the exposure of any user accessing any item; (2) Scaling to large deployments by incurring a logarithmic load on sites.

For the first goal, recall that the exposure of a client locating an item is given by the smallest zone containing both the client and the item. Our insight is to use compact graph techniques to formally guarantee an upper bound on the zone RTT diameter, hence bounding

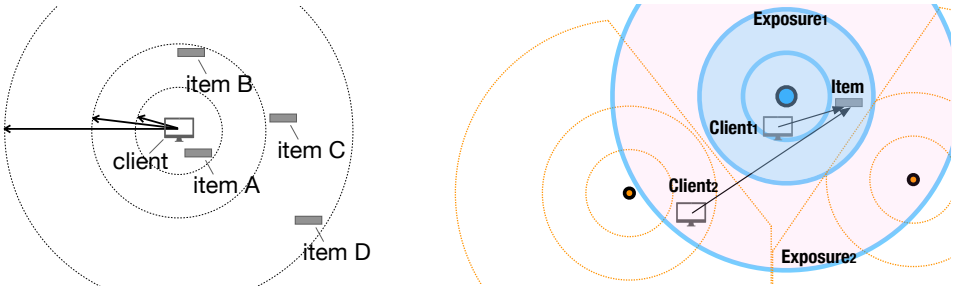



**Figure 5** Left: Autozoning strawman, bounding exposure for a single client accessing any item. Right: Limix autozoning, bounding exposure for *any* client accessing any item.

the client's exposure. Specifically, we want to ensure there exists a configuration service "close enough" to any client that has pointers to any item. Compact graph techniques approximate the distance between any two sites – in our case, between a client's site and an item – to *at most* $(2 \times k - 1) \times \overline{uv}$, where $\overline{uv}$ is the sites' RTT and $k$ is a parameter affecting exposure and load. Autozoning places sites into a zone if they lie within the approximate distance between a client's site and the item. Intuitively, building such zones for all user-item pairs ensures that *any* user $u$ looking up *any* item $i$ is *guaranteed* to find a "small enough" common zone of diameter *at most* $(2 \times k - 1) \times RTT(u, i)$.

The second goal of autozoning is to scale to large deployments. Limix relies on two techniques for scaling. First, from compact graph approximations theory, each site only needs to know about $O(\log N)$ other sites, where $N$ is the total number of sites in order to guarantee the exposure limits above. Even so, if we built all zones as described above – and imposed the zone deployment load on the member sites – the cost becomes prohibitive because of the large constants in $c * O(\log N)$. Instead, we use exponentially increasing zone radii, as in the strawman. As a result, each site participates in and runs a logarithmic number of zones. Figure 5-Right depicts the autozoning design, omitting the global zone for simplicity. Locality bounds become $i * 2^i$, where $i$ is the smallest zone such that $i * 2^i \geq (2 \times k - 1) \times RTT(u, v)$.

## 5.1 Autozoning algorithm

**Zone construction.** Alg. 3 depicts the zone construction. Compact-graph approximations use the sites as landmarks for approximating distances. Higher-level sites act as global landmarks to approximate large distances, whereas lower level sites act as local landmarks. Each site obtains level $i$ with probability $N^{-i/k}$ ($k$ represents the number of levels). Intuitively, a lower $k$ offers tighter bounds on distance approximation, which for Limix means lower exposure, at the cost of higher load on sites. To approximate distances, each sites maintains a set of sites as contact points, called its *bunch*. A site $u$ explores sites in ascending distance from itself, and adds a site $v$ in its bunch if $v$'s level $l_v$ is no smaller than that of any sites explored so far (including $u$). The sites $v$ closest to $u$ at every level form $u$'s witnesses (line 3). The inverse concept of a bunch is a *cluster*. $v$'s cluster is the set of sites around $v$, which are "close enough" to know about $v$ as a landmark (lines 4-5). Every site is a landmark and builds zones along its cluster, using exponentially increasing RTT diameters (lines 6-11).

**Lamport exposure bounds.** From the cluster construction, a site at level $k - 1$ has all sites in its cluster and, thus, creates a Global zone. Because site $w$ builds zones on its cluster, and $u$ and $v$ are in its cluster (in other words, $u$ and $v$ have $w$ in their recursive bunch), $u$ and $v$

**Algorithm 3** Autozoning at site $u$.

```
1: procedure BuildAutozones(u, Sites, nLevels, RTT)
2:     for v ∈ Sites do
3:         v.Witnesses ← CompWitnesses(Sites, nLevels, RTT)
4:         if RTT[u][v] < v.Witnesses[u.Level + 1] then
5:             u.Cluster ← u.Cluster ∪ v
6:     for v ∈ u.Cluster do
7:         for radius in i * 2^i do
8:             if RTT[u][v] < radius then
9:                 u.Zones[radius] ← u.Zones[radius] ∪ v
10:    for zone ∈ u.Zones do
11:        StartConfigurationService(zone)
```

are guaranteed to be in a zone of diameter at most $D = i * 2^i$, where $i$ is the smallest such that $i * 2^i \geq (2 \times k - 1) \times RTT(u, v)$. By construction, *any two* sites $u$ and $v$ – alternatively, a user contacting site $u$ to look up an item stored on $v$ – are guaranteed to find such a zone, providing guaranteed bounds on the Lamport exposure.

**Load.** The size of a site's bunch is a key property determining the number of zones that a site is a member of. From the probability distribution of level assignment, we expect to accept approximately $B = \frac{1}{n^{-1/k}}$ sites into $u$'s bunch at each level $i$. Thus, each site's bunch has, with high probability, size $|Bunch_u| \approx B \times k = B \times \log_B(N)$, which upper bounds the number of zones $u$ participates in. Factoring in the exponential zone diameter increase, $u$ participates on expectation in a polylogarithmic number of zones, or $O(\log N)$.

## 5.2 Scalability analysis

We study the system's scalability when the number of sites increases. We ask two questions: (1) *How many zones / configuration services should a site expect to run?*; and (2) *How much load should a site expect when running x zones?*

Figure 6 answers the first question, depicting a CDF of the expected number of zones per site. We analyzed different $N$ and $k$ parameters for a fixed network diameter $D$. The fixed overhead increases linearly per additional zone the site runs. But the dynamic load that each site serves depends on how many other sites are running the same zone, because only participating sites direct user requests to that zone.

Figure 7 depicts the expected dynamic load per site compared to the expected total number of zones in Limix. We consider a network diameter $D$ of 128 ms and each site issues 100 requests. For a set of $N$, $k$ and $D$, the expected number of zones is uniquely determined. The experiment varies $N$ and plots the expected load per site for systems with parameter $k = 2$ to $k = 5$. The analysis shows that, with bigger networks and more zones, the cost for each site grows with a slow logarithmic rate, which flattens for bigger $k$. We refer interested readers to the appendix for the detailed computation of the expected number of zones that a site is part of, which we used for computing the expected load per site. We conclude that Limix scales well on large wide-area systems.

# 6 Implementation

We implemented a Limix prototype of a configuration service backed by an existing data store. Limix stores its configuration in a per-zone strongly-consistent KV store: CockroachDB [24], a widely-used strongly-consistent data store. Although CockroachDB has rich functionality,

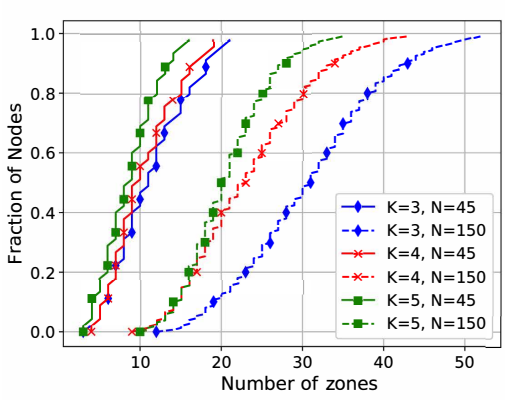



**Figure 6** Theoretical analysis: CDF of the number of zones per site.

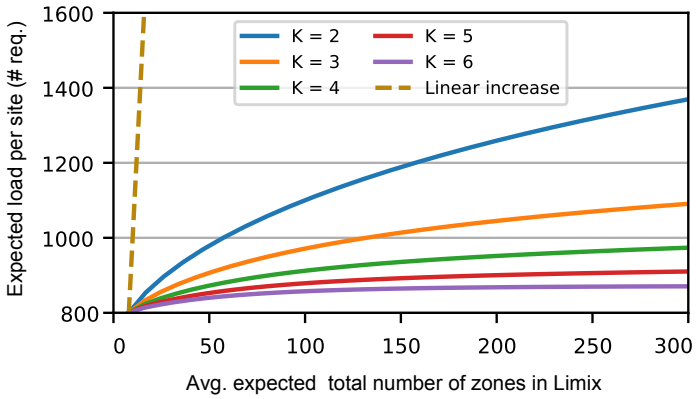



**Figure 7** Theoretical analysis: expected load per site if each site initiates 100 requests.

Limix only uses its basic KV store API read and compare-and-swap. A Limix coordinator runs on every site, providing an API to look up items with strong consistency guarantees. Our implementation is written in Go, with bash scripts for test infrastructure.

**Startup.** Each site in Limix runs a startup script, which takes an input a list of participating sites and either jurisdictions or autozoning with the number of levels. In the case of jurisdictions, the zone membership is given. For autozoning, each script measures its RTT to all other sites, and runs autozoning locally with the same parameters to learn the zone membership. The sites in each zone start a CockroachDB instance.

**Processing lookup requests.** Each site runs a Limix coordinator that provides a client API $itemLookup(key) : (zone, version)$, where version indicates the latest, strongly-consistent item pointer. Each site's coordinator queries (Algorithm 1) the configuration stores running on that site, i.e., CockroachDB deployments of the site's zones, through local Go *pq* queries [1] (a PostgreSQL-compatible driver). Each Limix coordinator running on a site part of zone $Z$ receives callbacks from the external data-plane store (CockroachDB deployment in our prototype) upon item creation, update, deletion or migration in the zone, which each coordinator uses to updates pointers and authoritative zone information (Alg. 2).

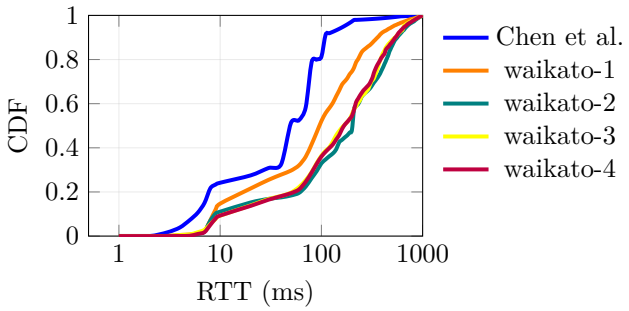



**Figure 8** CDF of reconfiguration RTTs.

## 7 Evaluation

We evaluated Limix's resilience to network partitions and overheads. Our first experimental setup considers jurisdictions, and evaluates Limix's overhead. Our second experimental setup focuses on Limix autozoning and its resilience, and compares Limix with Physalia.

**Testbeds.** We used two testbeds for our experiments, both orchestrated using Kubernetes. The **cluster testbed** runs 40 Kubernetes sites in a local cluster with a simulated network, which allows for more expeerimental flexibility. Each site requests 15GiB memory and one hyperthread of an Intel Xeon Gold 6240 CPU @ 2.60GHz. The delays between sites represent real-world delays of a globally distributed topology, as follows. Using CAIDA's Archipelago (Ark) Measurement Infrastructure [9], we selected 40 random monitors as site locations from the 90 available (Africa 8, Asia 13, Europe 26, North America 31, Oceania 4, South America 8). To compute the RTT between two sites at geographical distance $d$, we averaged the two monitors' median RTT for that distance as reported by CAIDA. Our RTT range is 0 to 602.25 ms.

The real-world testbed is deployed on Amazon Web Services (AWS) and spans 20 sites (US: East 4, West 4; Canada: central 2, Asia-Pacific: SouthEast 2, NorthEast 3, EU: Central 1, West 2, South 2). The server at each site has 16 GiB of DDR4 SDRAM, up to 10 Gbps bandwidth and 2vCPU on 3.1 GHz Intel Xeon processors. RTTs between sites range from 0.43 to 250.31 ms.

**Software setup.** Limix autozoning experiments use two levels for building zones through compact graph approximations, and zone diameters of $2(i-2)\sqrt{2}^i$, where $i \geq 3$. For the Physalia implementation, we use cells of up to 50 ms diameter, i.e., the 99th percentile write latency reported by the authors. Limix zones and Physalia cells run Core CockroachDB ver.20.1.

**Workloads.** We evaluate Limix using configuration writes (W), because they are the critical operations during gray failures (Section 2). Specifically, we use pairs of write (W-W) operations between pairs of sites. Each W-W pair concerns the same item. The operation pair emulates a reconfiguration for that item, e.g., after an item migration. The reconfiguration write issued by the second site has a strong consistency dependency on the first. The term "reconfiguration RTT" is the RTT between the two writers. The goal is to evaluate the exposure of the reconfiguration, given a known prior location of the configuration. In our experiments, we run the workload as Poisson arrivals, at a rate of 20 pairs per second.

**Table 1** Jurisdictions: features, measured availability.

| Feature | CockroachDB | | Limix |
|---|---|---|---|
| | Geo-replication | Private cloud | |
| Availability $Z_1$ reconfig. | 0% | 100% | 100% |
| Global configuration mgmt. | ✓ | × | ✓ |

We use real-world distributions for the reconfiguration RTTs based on metropolitan traffic traces. We chose metropolitan traffic traces as they capture more local traffic, as opposed to backbone links that might miss local traffic. Chen et al. [10] provide an RTT distribution over a 10Gbps metropolitan link – henceforth called trace 1. We compared this trace to four traces of the Waikato dataset [2], whose RTT distribution we extracted using a similar methodology as [10]: we matched data packets with the respective ACKs. These traces rely on public datasets and the extraction methodology avoids raising ethical issues. Figure 8 shows the cumulative distribution function (CDF) for the reconfiguration RTT. Given the similarity of the distributions, and the fact that trace 1 is more recent, we used trace 1 for generating the reconfiguration RTTs.

## 7.1 Jurisdictions: availability and costs

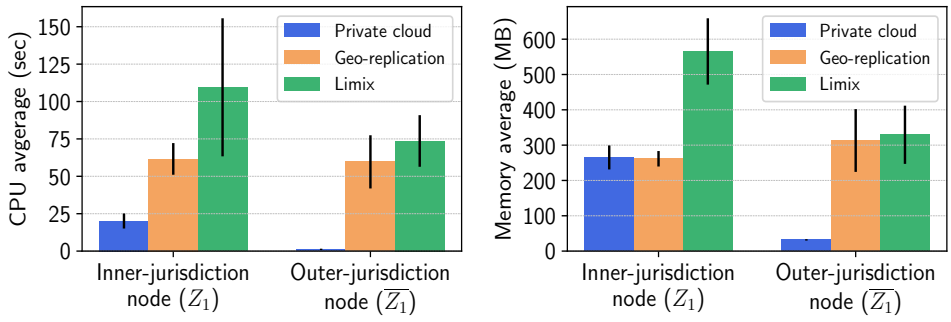
Our first experiment focuses on a simple Limix deployment in which each item exists in only one local zone, in addition to the default global zone. In this scenario we consider each of the disjoint local zones to represent an administratively-defined *jurisdiction*, such as a country. The experiment answers the question: "*What are the availability benefits and cost per Limix zone in this scenario?*"

**Methodology.** We ran the experiment on the cluster testbed. We focus on one particular jurisdiction centered around a CAIDA monitor in Europe West, and choose an RTT radius of 50 ms around the center, which roughly corresponds to the EU-West jurisdiction. Prior work has also shown that latencies of roughly 30 ms correspond to country-wide RTTs in Europe [10].

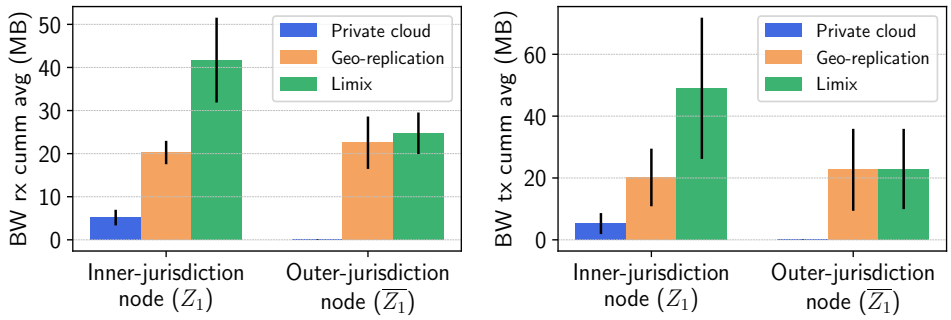
We are interested in the overheads during no-partition conditions, and in the availability during partitions. For this experiment, we run a synthetic workload with 2000 pairs of W-W operations: 1000 W-W pairs are for reconfiguration in EU-West, and 1000 for global reconfiguration. For the availability experiment, we run the same operations, but we disconnect EU-West from Global. We denote EU-West by $Z_1$ and Global \ EU-West by $\overline{Z_1}$. In Limix, sites in $Z_1$ are part of two zones, whereas sites in $\overline{Z_1}$ are part of one zone.

**Availability.** We compare Limix against two baselines: core CockroachDB with geo-replication in the Global zone – henceforth called Geo, and core CockroachDB deployed as a private cloud in EU-West only – henceforth called PCloud. Table 1 summarizes the features of the three designs. Geo offers global configuration management, but reconfiguration in $Z_1$ fails under partitions. The reason is that, under partition, $Z_1$ cannot reach a majority of configuration replicas. PCloud succeeds reconfigurations in $Z_1$ under partition, but offers no global configuration management, because all configuration is in $Z_1$. The experiment confirms that Limix offers both availability in $Z_1$ *and* global configuration management.

**Overheads.** For the same workload, we report the memory, CPU and bandwidth overheads under no-partition conditions. Figure 9 shows these overheads for sites in $Z_1$ and $\overline{Z_1}$. As expected, PCloud sites have the lowest overheads, but PCloud lacks global manageability.




**(a)** CPU. **(b)** Memory.




**(c)** Bandwidth RX. **(d)** Bandwidth TX.

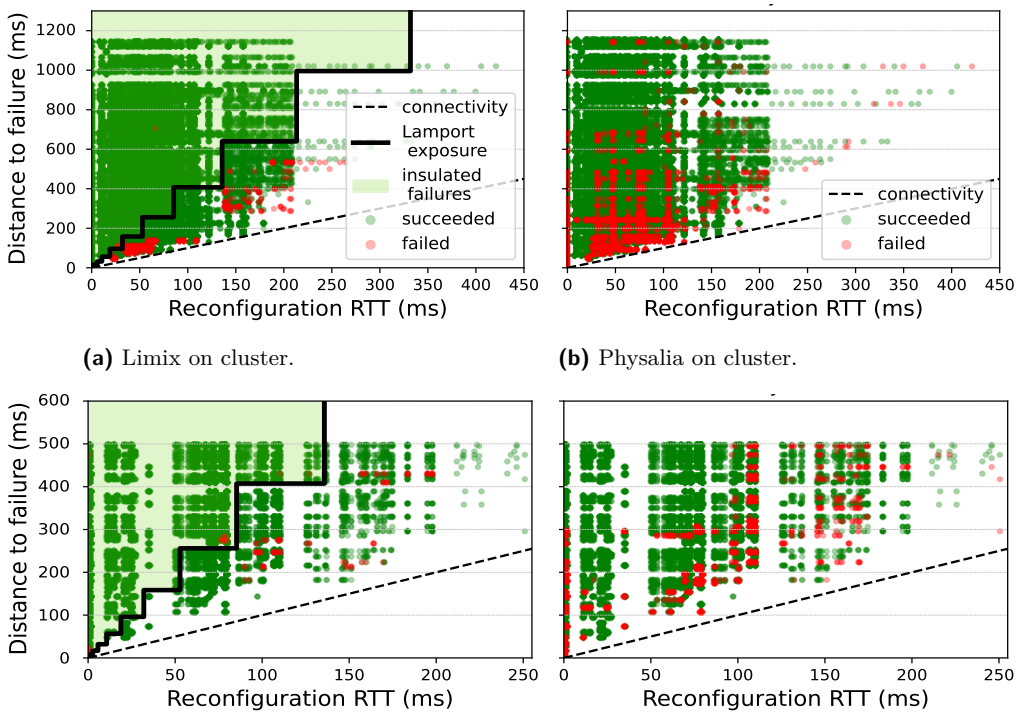
**Figure 9** Jurisdictions: compute, memory, and bandwidth overhead.

PCloud sites $\overline{Z_1}$ simply forward client requests to the closest site in $Z_1$, hence use almost no resources. The sites in $Z_1$ running PCloud have lower overheads than Geo sites because the PCloud deployment is smaller, and requires fewer resources for coordination between sites, for example. For its improved resilience guarantees, Limix sites in $Z_1$ spend about 2x the memory, CPU and bandwidth compared to Geo. The memory overhead stems from sites in $Z_1$ running two instances of CockroachDB: one corresponding to the inner- and the other to the outer-jurisdiction. To explain the CPU and bandwidth overheads, recall that, when a Limix site in $Z_1$ executes a write, it also writes to $\overline{Z_1}$. However, Limix sites in $\overline{Z_1}$, which do not have an improved resilience guarantee compared to Geo, have overheads very similar to Geo. We conclude that Limix is suitable for a highly configurable pay-as-you-go deployment, where extra resources spent increase availability.

## 7.2 Autozoning availability guarantees

This experiment evaluates to what extent the configuration of localized data is exposed to remote gray failures. This time, however, no jurisdictions are given: we use autozoning and test Limix's availability guarantees in comparison to Physalia. We ask the question: "*If a random site runs a reconfiguration for a random item, to what extend could remote failures cause the reconfiguration to fail?*"

**Methodology.** We generate pairs of writer clients; each pair performs a configuration write at a site chosen uniformly at random. The pairs are chosen as follows. We select 30 random sites, and in a random radius around each site of up to 150 ms chosen uniformly




**(a)** Limix on cluster.

**(b)** Physalia on cluster.

**(c)** Limix on AWS.

**(d)** Physalia on AWS.

**Figure 10** Comparison of availability of Limix and Physalia for different failure scenarios and testbeds.

at random, we generate 1000 interacting pairs. We disconnect the network at a distance $R_1 2 = x * R_1, x = \{1, 2, 3, 4, 5\}$, and run each experiment separately. We then depict the result of the interaction (success or fail) relative to the RTT between the interacting sites and the distance to the failure. Each pair writes a different key from the other pairs to avoid dependencies across pairs, hence the reconfiguration RTTs (i.e., of the interacting sites) are distributed uniformly at random.

We are interested in what dependencies the second writer might have on other sites, and how far in network distance these sites are: Dependencies on other sites could cause the second writer's configuration write to fail, e.g., by triggering a correlated failure. For this purpose, we select an area of a random radius R around the second writer, and we partition the network along the zone's border. The two writers are never partitioned from each other, because by design the second writer has a strong consistency dependency on the first, and then the second write would fail. By running this experiment, we test whether the second writer has dependencies outside the partitioned area.

Figures 10a and 10b depict the success or failure result for each writer-writer pair. The x-axis represents the reconfiguration RTT, and the y-axis represents the network distance to the partitions. Thanks to Limix's exposure guarantees, reconfigurations in Limix succeed more frequently than in Physalia. Both systems register failures below Limix's shield (black solid line), showing that they do have dependencies nearby. However, Physalia interactions also fail even when failures are above Limix's shield, i.e., relatively far from the sites. This is because Physalia does not provide availability guarantees for pairs interacting across cells: Even if two sites are relatively far from failures, if they are in different cells, their interaction

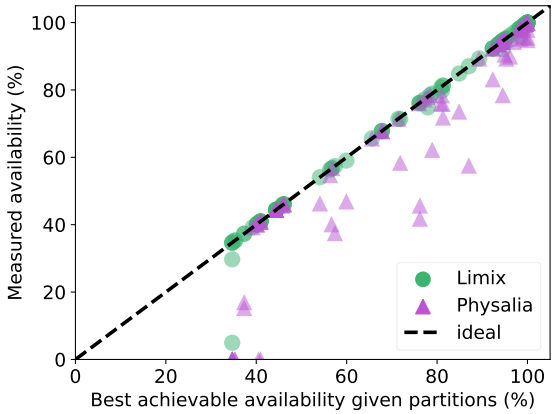



**Figure 11** Comparison of Limix to Physalia on AWS and realistic workloads.

depends on the infrastructure of both cells. If these cells are partitioned, their interaction might and does fail. Because these cells are non-overlapping, their combined scope and exposure can be significantly larger than the distance between the sites – subjecting the sites to a wider radius of failures. This is unfortunate, given that reconfigurations in Physalia are frequent. In contrast, Limix autozoning provides a clear availability guarantee, applicable to all interacting pairs: When failures are farther than the Lamport exposure bound of the two writers, reconfiguration is guaranteed to succeed.

## 7.3 Availability under real scenarios

This experiment, like the previous one, tests to what extent the configuration of localized data is exposed to remote gray failures, but on a real network and using realistic trace-based data. Because the AWS testbed has lower RTTs that are more clustered, Physalia cells are mostly between 10-30 ms RTT diameter, with a single cell up to 50 ms. Our methodology is similar to the experiment above, with the only exception that the workload of 1000 reconfiguration pairs of each experiment has a distribution of reconfiguration RTTs matching the one in trace 1 (Section 7). The workload is global, thus some interacting pairs cross the partition boundary.

Figures 10c and 10d plot the results only for the non-partitioned interactions. Our results generally mirror the ones we obtained on the cluster testbed, with Limix outperforming Physalia in almost all tested cases, and for the same reasons. There are, however, two notable differences from the previous experiment. First, Limix registers a few failures close to but above the shield. These failures correspond to some sites with a more unstable RTT than the others, whereas we depict the RTT measured at bootstrap time. However, the difference in RTT was minor, and for all the other sites we observed no violation. Second, we observe a more pronounced clustering effect of the plotted points, matching roughly our more clustered topology.

We also summarized the results for each tested workload as percentage of successes of the maximum possible availability for the non-partitioned pairs. Figure 11 shows that Limix's success rate is close to 100% in most cases, and significantly outperforms Physalia. We conclude that Limix provides strong guarantees on a variety of testbeds and workloads.

## 8 Related Work

**CAP tradeoffs.** Faced with partitions, some systems choose to relax consistency in favor of availability. Gemini [26] distinguishes access types that require a strongly- or eventually-consistent reply; this technique is known as segmentation [14]. Dynamo, a highly-available data store, takes a similar approach [12]. Seredinschi et al. [16] provide the user with several replies, increasing in consistency guarantees, enabling the client to perform speculative work. Local-first software [23] enables local access despite inaccessible service components by treating local data copies as primary and leaving inconsistency resolution to the application – fundamentally handling only weakly-consistent use cases. In contrast, Limix provides a coordination service ensuring that all accesses are strongly consistent.

**Availability during failures.** Several strongly-consistent systems employ replication to survive failures and partitions, however, they assume uncorrelated failures across sites [36, 25, 34, 30, 35]. Failures across geographical locations might not be independent, for several reasons: machines across sites run the same software and are vulnerable to the same bugs [17, 6]; sites' hard disks fill up at the same rate [6]; short-lived and, less frequently, long-lived (partial) partitions separate sites from each other, causing a domino of failures and ultimately unavailability [4]. Unlike the availability metric that Hauer et al. [19] recently proposed, which *reactively* analyzes failures after they occur, Limix *proactively* limits exposure in the first place. Glacier [17] employs massive replication of data, which Limix also does, to minimize the probability of data loss during large-scale correlated failures. As opposed to Limix, however, Glacier considers only data stores with immutable objects.

**Difference with prior workshop paper.** A preliminary version of our system, introducing Lamport exposure and describing the basic architecture, appeared in HotNets 2021 [7]. This paper significantly extends that work by scoping the definition of Lamport exposure to strongly-consistent coordination services for key/value stores in a multi-data-center deployment, builds the complete system and evaluates it against related work.

## 9 Discussion

**Limitations.** Limix implements the metadata service. While this is an important component, it does not by itself limit the Lamport exposure of the full service stack. We leave such dependencies for future work, such as upper-layer dependencies (e.g., on Javascript), power grid and network links whose failure could violate exposure-limiting policies (e.g., if communication between two sites in Germany crosses network links outside of Germany). This limitation could in principle be addresssed via deep structural dependency analysis [37], but it is outside the scope of this paper.

## 10 Conclusion

Can we achieve both the elasticity of globalized computing infrastructures and the resilience to distant failures of localized infrastructure? We show that this is possible by limiting the Lamport exposure. Limix is a configuration service that provides strong guarantees for a user's worst-case availability and performance in the presence of failures and partitions. Limix satisfies simultaneous bounds for any user accessing any item. Limix designs a control plane that limits exposure by running a separate lookup service per zone so that, if some

zones become partitioned and unavailable, other ones can respond instead. Limix's control plane supports administrative zones and existing strongly consistent data planes with item migration. But, it also defines an efficient and scalable autozoning algorithm with tight exposure bounds, guaranteeing that any user can access any data at distance $\Delta$ away, when failures occur beyond a small $O(\log N)$ multiple of $\Delta$. These techniques together enable Limix to achieve up to 50% better availability over the state-of-the-art, at a logarithmic overhead.

## References


1 The Go pq driver. URL: `https://github.com/lib/pq`.

2 The Waikato trace sets. URL: `https://wand.net.nz/wits/waikato/5/`.

3 CoreOS etcd, 2020. URL: `https://coreos.com/etcd/`.

4 Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

5 Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the shards: Managing datastore locality at scale with Akkio. In *Conference on Operating Systems Design and Implementation (OSDI)*, 2018.

6 Marc Brooker, Tao Chen, and Fan Ping. Millions of tiny databases. In *Conference on Networked Systems Design and Implementation (NSDI)*, 2020.

7 Cristina Băsescu and Bryan Ford. Immunizing systems from distant failures by limiting lamport exposure. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2021.

8 Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

9 CAIDA. CAIDA archipelago (ark) measurement infrastructure. `https://www.caida.org/projects/ark/`, 2021.

10 Xiaoqi Chen, Hyojoon Kim, Javed M. Aman, Willie Chang, Mack Lee, and Jennifer Rexford. Measuring tcp round-trip time in the data plane. In *Proceedings of the 2020 ACM SIGCOMM 2020 Workshop on Secure Programmable Network Infrastructure, SPIN@SIGCOMM 2020, Virtual Event, USA, August 14, 2020*, SPIN '20, pages 35–41, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3405669.3405823`.

11 David D. Clark. The design philosophy of the DARPA Internet protocols. In *ACM SIGCOMM*, August 1988.

12 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, SOSP '07, 2007.

13 Kathy Frankovic. Americans trust local governments over the federal government on covid-19. `https://today.yougov.com/topics/politics/articles-reports/2020/04/27/americans-trust-local-governments`, 2020.

14 S. Gilbert and N. Lynch. Perspectives on the CAP theorem. *Computer*, 45(2), February 2012. `doi:10.1109/MC.2011.389`.

15 Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), 2002. `doi:10.1145/564585.564601`.

16 Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 169–184, 2016. URL: `https://www.usenix.org/system/files/conference/osdi16/osdi16-guerraoui.pdf`.

17 Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

18 Osama Haq, Mamoon Raja, and Fahad R. Dogar. Measuring and improving the reliability of wide-area cloud paths. In *International Conference on World Wide Web (WWW)*, 2017.

19 Tamás Hauer, Philipp Hoffmann, John Lunney, Dan Ardelean, and Amer Diwan. Meaningful availability. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

20 Alex Hern. Fastly says single customer triggered bug behind mass internet outage, May 2021. URL: `https://www.theguardian.com/technology/2021/jun/09/fastly-says-single-customer-triggered-bug-that-caused-mass-outage`.

21 Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

22 Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8(9), 2010.

23 Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, pages 154–178, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3359591.3359737`.

24 Cockroach Labs. CockroachDB: Ultra-resilient SQL for global business, 2020. URL: `https://www.cockroachlabs.com/`.

25 Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, December 2001.

26 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

27 Tom Lianza and Chris Snook. Cloudflare outage. `https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/`, November 2020.

28 Ben Lobel. Local businesses are more trusted than large enterprises, finds survey by yell. `https://smallbusiness.co.uk/local-businesses-trusted-large-enterprises-2538416/`, 2020.

29 Netscout. Netscout security report, 2021. URL: `https://www.arbornetworks.com/resources/infrastructure-security-report`.

30 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, 2014.

31 Laura Stevens. Amazon finds the cause of its AWS outage: A typo, 2017. URL: `https://www.wsj.com/articles/amazon-finds-the-cause-of-its-aws-outage-a-typo-1488490506/`.

32 Mikkel Thorup and Uri Zwick. Approximate distance oracles. In *ACM Symposium on Theory of Computing*, pages 183–192, 2001. `doi:10.1145/1044731.1044732`.

33 Mikkel Thorup and Uri Zwick. Compact routing schemes. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–10, New York, NY, USA, 2001. ACM Press. `doi:10.1145/378580.378581`.

34 Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. Near-optimal latency versus cost tradeoffs in geo-distributed storage. In *Conference on Networked Systems Design and Implementation (NSDI)*, 2020.

35 Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

36 Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Symposium on Operating Systems Principles (SOSP)*, 2013.

37 Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. Heading off correlated failures through Independence-as-a-service. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014.

## A Appendix: Theoretical Analysis of Autozoning Algorithm Scalability

In this section, we compute the expected number of zones that a site is a part of. Let $N$ be the set of all sites and $n$ is the number of sites in the system. The system has $k$ levels, where sites are promoted from level $i$ to $i+1$ with probability $1/\beta$, giving $k = \log_\beta(n)$. Let $L_A$ denote the level of site $A$, and let $l_{\max}$ be the maximum level. Define $i_{\min}$ and $i_{\max}$ such that $2^{i_{\min}} = R_{\min}$ and $2^{i_{\max}} = R_{\max}$ (the network diameter). Note that $\forall A \in N : A$ has a coordinate $(x_A, y_A)$ and $x_A^2 + y_A^2 < (\frac{R_{max}}{2})^2$. The circle with center site $A$ and radius $2^i$ is defined as $C(A,i) = \{(x,y) \mid (x-x_A)^2 + (y-y_A)^2 < 2^{2i}\}$.

To find the number of zones that a site $A$ ($A \in N$) is a part of, we denote zones made by $B$ ($B \in N,\ B \neq A$) that $A$ is a part of that zone by $A \leftarrow B$ and zones made by $A$ by $A \leftarrow A$. Also, We denote all the zones that $A$ is a part of by $A \leftarrow *$.

$$E[\#\ of\ A \leftarrow *] = E[\#\ of\ A \leftarrow A] + (n-1)E[\#\ of\ A \leftarrow B] \tag{1}$$

To find the expected number of zones formed by $A$ we have:

$$E[\#\ of\ A \leftarrow A] = \sum_{l=0}^{l_{max}} P(L_A = l)E[\#\ of\ A \leftarrow A \mid L_A = l] \tag{2}$$

Based on how levels are assigned with parameter $\beta$, the probability that level of $A$ is $l$ is:

$$P(L_A = l) = Pr(l) = \begin{cases} (\frac{1}{\beta} - \frac{1}{\beta^{l+1}}), & 0 \le l < l_{max} \\ \frac{1}{\beta^{l_{max}}}, & l = l_{max} \end{cases}$$

We denote the biggest zone of $A$ by $zone_m^A$ . To compute Eq. (2), for $l < l_{max}$ we have:

$$E[\#\ of\ A \leftarrow A \mid L_A = l] = \sum_{i=i_{min}}^{i_{max}} P(C(A,i)\ is\ zone_{max}^A \mid L_A = l)(i - i_{min} + 1) \tag{3}$$

In Eq. (3), $(i - i_{min} + 1)$ is the number of zones that $A$ forms when $C(A,i)$ is the biggest zone of $A$. We use an over- and an under-approximation to bound the value of Eq. (3).

Node $A$ forms its zones based on the sites in its cluster. When another site $B$ is in $A$'s cluster, $A$ is in $B$'s bunch. It means that $A$ is closer to $B$ than any other system site with level $> L_A$. Figure 12 shows for over-approximation, the biggest possible radius of $A$'s zone is equal to the distance between A and closest site to $A$ with a level higher than $A$'s level.

For over-approximation we suggest Eq. (4). Note that $(1 - (\frac{2^i}{2^{i_{max}}})^2)$ is the probability that a site, for example $A' \in N$, is not in $C(A,i)$ and $\frac{n}{\beta^{l+1}}$ is the number of sites with level higher than $l$. We define:

$$f(i) = (1 - (\frac{2^i}{2^{i_{max}}})^2)^{\frac{n}{\beta^{l+1}}} - (1 - (\frac{2^{i+1}}{2^{i_{max}}})^2)^{\frac{n}{\beta^{l+1}}}$$

So for over-approximation and for $l < l_{max}$ we have:

$$\begin{aligned} &P(C(A,i)\ is\ zone_{max}^A \mid L_A = l) < P(\nexists\ A' \in N : L_A \le L_{A'},\ A' \notin C(A,i)) \\ &= (1 - (\frac{2^i}{2^{i_{max}}})^2)^{\frac{n}{\beta^{l+1}}} - (1 - (\frac{2^{i+1}}{2^{i_{max}}})^2)^{\frac{n}{\beta^{l+1}}} = f(i) \end{aligned} \tag{4}$$

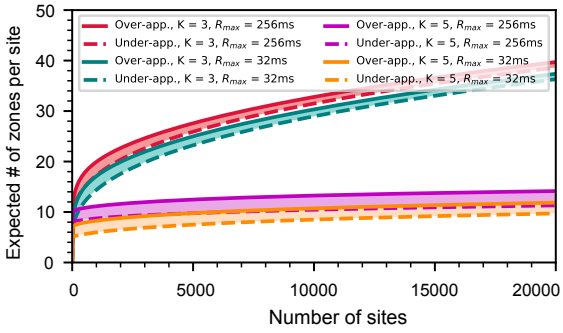



**Figure 12** Expected number of zones per site vs number of sites in Limix's autozoning.

For the sites with the highest level, the biggest zone has radius $R_{max}$, so the number of zones that they make is $(i_{max} - i_{min} + 1)$. Using it and replacing Eq. (4) in Eq. (2) we get:

$$E[\#\ of\ A \leftarrow A] < Pr(l_{max})(i_{max} - i_{min} + 1) + \sum_{l=0}^{l_{max}-1} Pr(l) \sum_{i=i_{min}}^{i_{max}-1} f(i)(i - i_{min} + 1) \tag{5}$$

Figure 12 for under-approximation suggests that the biggest possible radius of $A$'s zone is equal to half of the distance between $A$ and closest site to $A$ with a level higher than the level of $A$, so under-approximating of Eq. (3) we suggest that:

$$\begin{aligned} &P(C(A,i)\ is\ zone^{A}_{max} \mid L_A = l) > P(\nexists\ A' \in N : L_A \leq L_{A'}, A' \notin C(A, i+1)) \\ &= (1 - (\frac{2^{i+1}}{2^{i_{max}}})^2)^{\frac{n}{\beta^{l+1}}} - (1 - (\frac{2^{i+2}}{2^{i_{max}}})^2)^{\frac{n}{\beta^{l+1}}} = f(i+1) \end{aligned} \tag{6}$$

Replacing Eq. (6) inEq. (2) we get:

$$E[\#\ of\ A \leftarrow A] > Pr(l_{max})(i_{max} - i_{min} - 1) + \sum_{l=0}^{l_{max}-1} Pr(l) \sum_{i=i_{min}}^{i_{max}-2} f(i+1)(i - i_{min} + 1) \tag{7}$$

To find the expected number of zones formed by $B$ that $A$ is a part of, note that $A$ can be in one of zones made by $B$ only if $L_A$ is not higher than $L_B$.

$$\begin{aligned} &E[\#\ of\ A \leftarrow B] = \sum_{l=0}^{l_{max}} P(L_B = l) \sum_{l'=0}^{l_{max}} P(L_A = l') \times E[\#\ of\ A \leftarrow B \mid L_B = l, L_A = l'] \\ &= \sum_{l=0}^{l_{max}} P(L_B = l) \sum_{l'=0}^{l} P(L_A = l') \times E[\#\ of\ A \leftarrow B \mid L_B = l, L_A = l'] \\ &= \sum_{l=0}^{l_{max}} P(L_B = l) P(L_A \leq l) E[\#\ of\ A \leftarrow B \mid = l,\ L_A \leq l] \end{aligned} \tag{8}$$

In Eq. (8) for $l < l_{max}$, notice that when $L_A > L_B$, $A$ can not be included in $B$'s zones, so the sum over $l'$ is $\leq l$. We define $S(i) = (\frac{2^i}{2^{i_{max}}})^2$. The ring-shaped area between the 2 circles with the center of site $B$ and radiuses of $2^j$ and $2^{j+1}$ is denoted by $D(B, j)$. The probability

that a site $A \neq B$, is in $D(B,j)$ is $S(j+1)-S(j)$ which we denote by $\Phi(j) = S(j+1)-S(j)$.

$$E[\#\ of\ A \leftarrow B \mid L_B = l,\ L_A \leq l] = \sum_{j=i_{min}-1}^{i_{max}} P(A \in D(B,j)) \sum_{i=j+1}^{i_{max}} P(C(B,i) = zone_m^B)(i-j)$$

$$= S(i_{min}) \sum_{i=i_{min}}^{i_{max}} P(C(B,i) = zone_m^B)(i - i_{min} + 1)$$

$$+ \sum_{j=i_{min}}^{i_{max}-1} P(A \in D(B,j)) \sum_{i=j+1}^{i_{max}-1} P(C(B,i) = zone_m^B)(i-j)$$

$$= S(i_{min}) \sum_{i=i_{min}}^{i_{max}} P(C(B,i) = zone_m^B)(i - i_{min} + 1) + \sum_{j=i_{min}}^{i_{max}-1} \Phi(j) \sum_{i=j+1}^{i_{max}-1} P(C(B,i) = zone_m^B)(i-j) \tag{9}$$

In Eq. (9), $(i-j)$ is the number of zones formed by $B$ that $A$ is a part of if $A$ is within distance of $2^j$ and $2^{j+1}$ of site $B$ when $C(B,i)$ is the biggest zone formed by $B$. For $L_B = l_{max}$ all the sites, including $A$, are in $B$'s cluster and $B$'s biggest zone has radius $R_{max}$, so:

$$E[\#\ of\ A \leftarrow B \mid L_B = l_{max}] = S(i_{min})(i_{max} - i_{min} + 1) + \sum_{j=i_{min}}^{i_{max}-1} \Phi(j)(i_{max} - j) \tag{10}$$

Replacing Eq. (4) and Eq. (10) in Eq. (9) and replacing the result in Eq. (8), we get:

$$E[\#\ of\ A \leftarrow B] < Pr(l_{max})[S(i_{min})(i_{max} - i_{min} + 1) + \sum_{j=i_{min}}^{i_{max}-1} \Phi(j)(i_{max} - j)]$$

$$+ \sum_{l=0}^{l_{max}-1} Pr(l)(1 - \frac{1}{\beta^{l+1}}) \times [S(i_{min}) \sum_{i=i_{min}}^{i_{max}-1} f(i)(i - i_{min} + 1) + \sum_{j=i_{min}}^{i_{max}-1} \Phi(j) \sum_{i=j+1}^{i_{max}-1} f(i)(i-j)] \tag{11}$$

Replacing Eq. (6) and Eq. (10) in Eq. (9) and replacing the result in Eq. (8), we get:

$$E[\#\ of\ A \leftarrow B] > Pr(l_{max})[S(i_{min})(i_{max} - i_{min} + 1) + \sum_{j=i_{min}}^{i_{max}-1} \Phi(j)(i_{max} - j)]$$

$$+ \sum_{l=0}^{l_{max}-1} Pr(l)(1 - \frac{1}{\beta^{l+1}})$$

$$\times [S(i_{min}) \sum_{i=i_{min}}^{i_{max}-2} f(i+1)(i - i_{min} + 1) + \sum_{j=i_{min}}^{i_{max}-1} \Phi(j) \sum_{i=j+1}^{i_{max}-2} f(i+1)(i-j)] \tag{12}$$

By replacing Eq. (11) and Eq. (5) in Eq. (1), we have an over-approximation of expected number of zones that a site is a part of:

$$E[\#\ of\ A \leftarrow *] < Pr(l_{max})(i_{max} - i_{min} + 1) + \sum_{l=0}^{l_{max}-1} Pr(l) \sum_{i=i_{min}}^{i_{max}-1} f(i)(i - i_{min} + 1)$$

$$+ (n-1)[Pr(l_{max})[S(i_{min})(i_{max} - i_{min} + 1) + \sum_{j=i_{min}}^{i_{max}-1} \Phi(j)(i_{max} - j)] + \sum_{l=0}^{l_{max}-1} Pr(l)(1 - \frac{1}{\beta^{l+1}}) \times$$

$$[S(i_{min}) \sum_{i=i_{min}}^{i_{max}-1} f(i)(i - i_{min} + 1) + \sum_{j=i_{min}}^{i_{max}-1} \Phi(j) \sum_{i=j+1}^{i_{max}-1} f(i)(i-j)]]$$

(13)

By replacing Eq. (12) and Eq. (4) in Eq. (1), we have an under-approximation of expected number of zones that a site is a part of:

$$E[\#\ of\ A \leftarrow *] > Pr(l_{max})(i_{max} - i_{min} + 1) + \sum_{l=0}^{l_{max}-1} Pr(l) \sum_{i=i_{min}}^{i_{max}-2} f(i+1)(i - i_{min} + 1)$$
$$+ (n-1)[Pr(l_{max})[S(i_{min})(i_{max} - i_{min} + 1) + \sum_{j=i_{min}}^{i_{max}-1} \Phi(j)(i_{max} - j)] + \sum_{l=0}^{l_{max}-1} Pr(l)(1 - \frac{1}{\beta^{l+1}})$$
$$\times [S(i_{min}) \sum_{i=i_{min}}^{i_{max}-2} f(i+1)(i - i_{min} + 1) + \sum_{j=i_{min}}^{i_{max}-1} \Phi(j) \sum_{i=j+1}^{i_{max}-2} f(i+1)(i-j)]]$$

(14)