# Mutating *etcd* Towards Edge Suitability

Andrew Jeffery
University of Cambridge
Cambridge, United Kingdom
andrew.jeffery@cst.cam.ac.uk

Heidi Howard
Azure Research, Microsoft
Cambridge, United Kingdom
heidi.howard@microsoft.com

Richard Mortier
University of Cambridge
Cambridge, United Kingdom
richard.mortier@cst.cam.ac.uk

## ABSTRACT

In the edge environment servers are no longer being co-located away from clients, instead they are being co-located with clients away from other servers, focusing on reliable and performant operation. Orchestration platforms, such as *Kubernetes*, are a key system being transitioned to the edge but they remain unsuited to the environment, stemming primarily from their critical key-value stores. In this work we derive requirements from the edge environment showing that, fundamentally, the design of distributed key-value datastores, such as *etcd*, is unsuited to meet them. Using these requirements, we explore the design space for distributed key-value datastores and implement two successive mutations of *etcd* for different points: *mergeable etcd* and *dismerge*, trading linearizability for causal consistency based on CRDTs. *mergeable etcd* retains the linear revision history but encounters inherent shortcomings, whilst *dismerge* embraces the causal model. Both stores are local-first, maintaining reliable performance under network partitions and variability, drastically surpassing *etcd*'s performance, whilst maintaining competitive performance in reliable settings.

The source code this project is available at
https://github.com/jeffa5/mergeable-etcd

## 1 INTRODUCTION

More compute resources are becoming available near the edge of the network leading to an increasing interest in deploying services there. These services can perform aggregation closer to the edge, reducing the volume of data to be sent to the cloud as well as offering clients more local operations [26]. They can typically be deployed in mini data centers [12] — small, mostly ISP operated, compute sites. With each site being geographically distributed, networks between edge sites can have higher latency than intra-datacenter communication coupled with increased likelihood of network partitions. This is further exacerbated by resource limitations at each site, requiring efficient use of those resources.

Resource aggregation is critical to this environment, exploiting the numerous but geodistributed resources each site offers. Aggregating sites into larger clusters enables running larger jobs with higher availability, capitalising on the deployed resources and the periodicity of demand. A single large cluster also eases management and operation of the services, offering them higher availability across sites through efficient orchestration.

*Kubernetes* [7] is a *container* orchestration platform based on Google's Borg [34] system. *Kubernetes* is used by a majority of the top 500 companies in the world [9], managing deployments of services over thousands of nodes, handling failures automatically [31]. In large data centers it leverages the low latency networks, having a centralised control-plane and datastore, *etcd* [4], for coordinating the various actions. Despite its prevalence in data centers, there is

growing interest in deploying it to mini data centers close to the edge with specific projects targeting this use case [5, 6].

*etcd* is a distributed, but logically centralised, key-value store. It is widely used for cloud applications, including *Kubernetes*, *Rook*, *CoreDNS* and *M3* [4]. Due to its critical place in these systems it is a key factor for them being suitable to deploy to the edge. In fact, *etcd* has already been shown to have scalability limitations under best-case scenarios [22], which would only be exacerbated at the network edge with its higher latency cross-site links. As *etcd* is critical to cloud applications' operation, they are also bounded by *etcd*'s ability to tolerate higher latencies and network faults, impacting scalability and reliability [16–19].

In the process of analysing and deriving requirements from the edge environment we present the design and implementation of two successive adaptations to *etcd*: *mergeable etcd* and *dismerge* trading linearizability [21, 35] for causal consistency [27, 29, 35] with Conflict-free Replicated DataTypes (CRDTs) [28, 33]. These target the edge environment with limited heterogeneous resources whilst maintaining as close semblance to *etcd* as possible to minimise programming model differences and thus respective changes in the systems built around *etcd*. They explore two different points in the design space, *mergeable etcd* focusing on maintaining compatibility with *etcd* and its linear history, and *dismerge* exploring the impacts of changes of exposing the causal history explicitly. From these design choices, we show that both datastores maintain consistent performance under network partitions and variability, surpassing *etcd*'s performance, whilst also remaining competitive in reliable settings at the edge. Our contributions in this paper are as follows:

(1) We analyse the requirements for edge focused distributed key-value stores, Section 2.
(2) We outline design trade-offs to cater for these requirements, Section 3.
(3) We present the implementation of the two datastores exploring different parts of this design space, Section 4.
(4) We evaluate the systems highlighting *mergeable etcd*'s and *dismerge*'s ability to operate with consistent performance under larger cluster sizes and added latency, Section 5.
(5) We discuss the implications of the changes applied on broader systems, particularly *Kubernetes*, Section 6.

## 2 BACKGROUND AND MOTIVATION

### 2.1 *etcd*

*etcd* is a Raft-based [30] linearizable distributed key-value store, requiring majority quorums. It exposes a straightforward API with the ability to get ranges of values, write values and delete ranges as well as being able to do these within transactions, all over a single flat key-space. Another aspect of *etcd* is its ability for clients
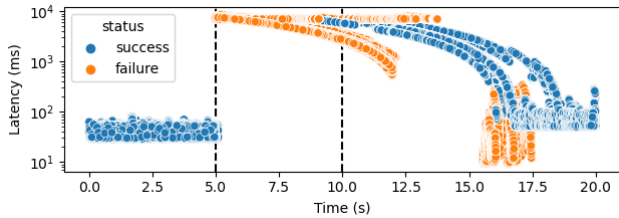
**Figure 1: Impact of a network partition on a 3 node *etcd* cluster. See Section 5.1 for more details.**

to *watch* values and get updates pushed to them directly, through *watch streams*, without the need to poll. Due to its use of linearizability *etcd* can struggle to perform adequately at scale [22], this is fundamentally limited by the fault-tolerance model it adopts [14]. Since only a leader can process write requests, or linearizable reads, client requests must either target the leader directly or be forwarded, adding extra latency out of the control of the client. Due to its fault-tolerance model *etcd* is unable to process requests without communicating with a majority of nodes (Figure 1), leaving partitioned sites unable to adapt. *etcd* can also exhibit subtle failure conditions under misbehaving networks [23].

*etcd* makes the following guarantees about its Key-Value API [8]:

**Atomicity**  Operations complete entirely or not at all.

**Durability**  Completed operations are durable and a read operation never returns data that is not durable.

**Consistency**  Operations are linearizable.

**Completeness of watches**  Watch events never observe partial events for a single operation.

**Global revision**  Each mutating request is assigned a strictly monotonically increasing revision number.

## 2.2  Edge environment

We focus on miniature data centers and compute at the network edge. These sites are resource constrained in multiple dimensions: CPU, memory, and networking. Near-edge compute sites are typically small but larger in number to provide closer operation to the user. This large scale places emphasis on avoiding overheads from cross-site communication which can be costly but also unreliable in latency, bandwidth and consistency due to competing with user traffic.

Applications running at the edge and serving user traffic want low latency operation, to be able to handle a dynamic environment, avoid cross-site dependencies and be able to progress independently of other sites.

## 2.3  Deriving requirements

From the characteristics of the edge environment and the expectations of applications relying on datastores such as *etcd*, we derive the following requirements for datastores deployed at the edge:

*Site-local reads.*  To serve applications with low latency and avoid cross-site communication reads need to be site-local. This can be viewed similarly to a content-delivery network [32] which has content cached at the edge to reduce latency of operations. Implied

by site-local reads, each node needs to maintain all historical data for each key locally. This limits the overall quantity of data that can be stored but is key in enabling site-local reads with history.

*Site-local writes.*  Further to site-local reads we also want a system that supports site-local writes. This ensures that the system can operate even when network connectivity is impaired.

*Performance.*  Since edge applications need to be performant for user expectations as well as supporting lots of work at the edge we require the datastore to be performant.

*Resource efficiency.*  In addition to performance, we want our application to be efficient in its storage, using a small overhead compared to the raw data storage requirement.

Of these requirements, *etcd* is only able to fulfil site-local reads when serializable reads are used, which is uncommon in our experience. Site-local writes are never possible in *etcd* clusters of more than one node. Performance will be covered more in the evaluation section (Section 5), but its architecture is targeted towards cloud data center deployments. Due to this targeting, it is also not the most resource efficient, ideally running on large multi-core machines.

## 2.4  Application deployment

Given deployments of *etcd* to the edge, we observe three main strategies based off *Kubernetes*: single-site (K3s) [5], cross-site (vanilla) [7], and cloud-centric (KubeEdge) [6]. Figure 2 shows the layout of these and Table 1 highlights the requirements they satisfy from the point of view of a single edge site, assuming *etcd* would be deployed at each control plane node. Blast radius considers what would be impacted if a site with control-plane node gets disconnected from everything else.

Datastores based on eventual consistency, such as Cassandra [15], can be deployed in equivalent configurations but still do not satisfy the requirements. Since data is partitioned across nodes, each node does not store all data, violating the site-local reads requirement, writes are also not guaranteed to be served locally, depending on replication requirements.

Running small clusters of datastores such as *etcd* at the center of large systems such as *Kubernetes* leaves the large systems vulnerable to broader faults, particularly at the edge. As these systems become distributed across data centers for fault-tolerance, or edge sites for locality, they may retain access to only one datastore node. When this datastore node becomes unable to process requests, due to failure, all attached clients are unable to perform their actions. This creates a very large blast radius for the core distributed key-value store, commonly relying on majority replication with a cluster size of 3 or 5.

## 3  DESIGN SPACE

Table 2 highlights the key differences in the datastores presented. This focuses around four primary points in the design space: consistency of data, how history is addressed, durability of data, and how values are represented. In this section we explore the choices each datastore makes within these parameters.
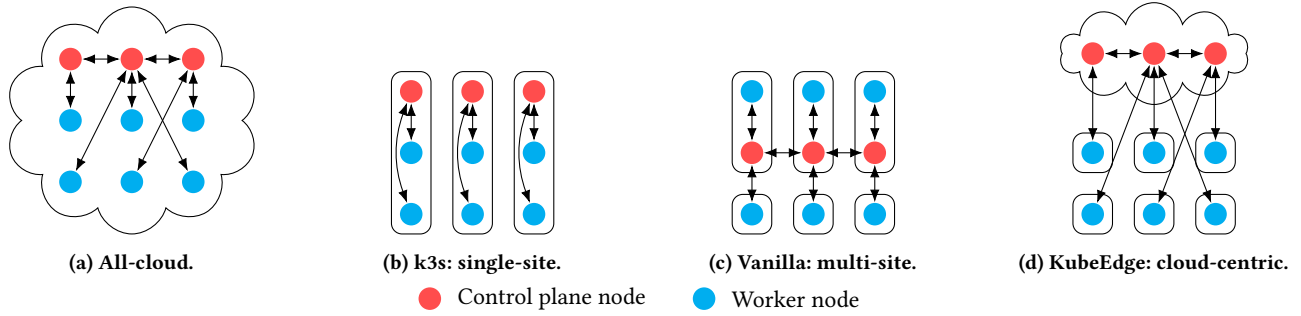
(a) All-cloud.  (b) k3s: single-site.  (c) Vanilla: multi-site.  (d) KubeEdge: cloud-centric.

● Control plane node  ● Worker node

**Figure 2: Edge application deployment strategies. Boxes indicate edge sites, arrows indicate potential connections.**

**Table 1: Comparison of requirements met by *etcd* deployed with deployment strategies from Figure 2.**

| Case | Site-local reads | Site-local writes | Efficiency | Management | Blast radius |
|------|------------------|-------------------|------------|------------|--------------|
| All-cloud | Yes | Yes | Great | Single cluster | Single site |
| Single-site | Yes | Yes | Wasted resources | Lots of clusters | Single site |
| Multi-site | No | No | Great | Single cluster | Multiple sites |
| Cloud-centric | No | No | Bandwidth cost | Single cluster | All edge sites |

**Table 2: Comparison of properties of the datastores.**

| Store | Consistency | Fault tolerance | History addressing | Durability | Values |
|-------|-------------|-----------------|--------------------|------------|--------|
| *etcd* | Linearizable | $2f + 1$ | Integer counter | Majority of nodes | Bytes |
| *mergeable etcd* | Causal | $f + 1$ | Integer counter | Single node | Operator-defined |
| *dismerge* | Causal | $f + 1$ | Hash graph heads | User dependent | Operator-defined |

## 3.1 Consistency and fault tolerance

> **Lesson**: Strong consistency is an availability and scalability bottleneck.
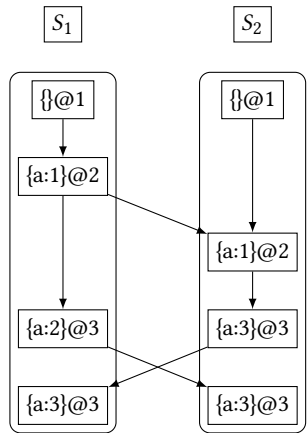
*etcd* uses strong consistency, particularly linearizability, to replicate values between stores. This means that, for $f$ node failures, it requires $2f + 1$ to be in the cluster. In cloud environments, *etcd* can make assumptions of node homogeneity, for both node sizes and network links. However, near the edge these assumptions, particularly those of the network links, may not hold. This impacts the scalability of the cluster, and ultimately the availability it can provide. Therefore, the heterogeneous nature of the edge leads to the imbalance of fault tolerance across sites explored in Section 2.4. Since *etcd* is the critical core of many systems, it is notable that this limitation of fault-tolerance directly impacts systems considerably bigger than itself.

A weaker variant of using linearizability, which enables stronger availability, is causal consistency. This can be easily implemented with CRDTs. This model enables the data viewed at different nodes of a system to differ, with the guarantee that it will converge in the steady-state. In practice, this enables pushing replication of updates between nodes from happening eagerly to happening lazily. This decouples nodes, enabling them to tolerate more heterogeneous network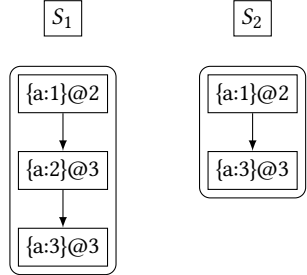 links, including handling updates whilst experiencing complete partitions from the cluster. To tolerate $f$ node failures these systems require only $f$+1 nodes in the cluster. This decoupling also enables these clusters to scale better, being able to match the deployment scale of edge sites. This makes the applications built on these systems able to be more performant and reliable.

*3.1.1 Mutable histories.* One challenge in adapting the data model of *etcd* to work with causal consistency is that the previously immutable history becomes mutable. Figure 3a shows the process of two peers synchronizing whilst having writes from separate clients. The first write is to $S_1$ which synchronizes with $S_2$ without it having concurrent writes, so they both remain consistent. However, both nodes then receive concurrent writes to the same key, *a*. This means that they will both use the same revision for this update, 3, but have different values for the key. When they next synchronize this value needs to be made consistent across the replicas and in this case the value from $S_2$ wins over the value from $S_1$. If the client who last wrote to $S_1$ retrieves the value for *a* again, it will see the updated value 3 at the same revision. This mutable history is a consequence of the causal consistency coupled with *etcd*'s global revision counter.

Due to lazy synchronizations, datastores can have an imbalance of updates made to them. If the same key is altered on different nodes concurrently then upon a merge the one with the higher revision may dominate the other. This can even be due to updates on other keys in the store, artificially progressing the revision counter

(a) Sequence of updates to two *mergeable etcd* datastores. History is mutable (revision 3 on $S_1$).



(b) Sequence of corresponding watch updates.

**Figure 3: Updates and watches at *mergeable etcd*. Notation in the form {*key* : *value*}@*revision*.**

before the same key is then updated. This dominating behaviour is worst when synchronization is infrequent, particularly likely in times of failures such as network partitions. *mergeable etcd* is more vulnerable to this behaviour than *dismerge* due to the way that they address changes.

*3.1.2 Watching values.* When a client requests a stream of watch events from a server it is guaranteed to observe complete changes, knowing the history is immutable. Since the history can change in *mergeable etcd*, two watch streams (connected to different servers) may observe different values at the same revision, breaking this guarantee. When the two servers synchronize they will have a consistent view of the values but the clients may not be updated with the result of this conflict-resolution. When synchronizing the servers can send watch events for values if the revision is newer, or even the same as that last sent as long as the incoming value is the *winner*. For example, in Figure 3b the server $S_1$ would send the new update for revision 3 whilst server $S_2$ does not need to as it has already sent that value. The first client will have a local conflict and so should forget its past value and accept the newer one, whilst the latter client retains the original value.

## 3.2 Addressing history

**Lesson**: Linear histories prevent all changes being addressed under causal consistency.

*etcd* maintains the history of all values, making them addressable with an integer counter, Figure 4a. This provides users with a unique handle for changes which they can use to look back in time, or resume watch streams from a known last position. This counter is suitable under linearizability as there can only be one update for each revision. With causal consistency, this breaks down because changes can be made to multiple nodes in parallel, thus they may get the same revision assigned. When the nodes synchronize, the updates will effectively conflict in the history space, breaking the expectation that the revision counter is a unique handle, Figure 4b. Additionally, updates synchronized from nodes can appear in the past. This poses challenges for sending updates over watch streams as the clients expect to already have observed the latest version, and so should not be sent an update for a past revision. However, due to the nature of the update clients may care about it and wish to update the value after merging the representations, this is not possible using the single counter revisions.

Instead, when multiple nodes are accepting updates, we can use vector clocks to tag the updates, forming a directed acyclic graph (DAG) of changes, Figure 4c. This has the advantage that now every update has a unique identifier but the downside of the clocks growing, without removal. The clocks will grow linearly in size $O(n)$ for $n$ nodes in the cluster, which is large near the edge. These clocks would be included in every request to identify the current *revision* for clients. Rather than incur the overhead of sending these clocks over the network, we can view the updates as a hash DAG, similar to that of Git [13], Figure 4d. Each update is uniquely represented by a single hash, which encompasses the operations in the update itself along with the hashes of its ancestors, scaling with $O(1)$ independently of the size of the cluster. This equates to every change being a "merge commit" of the frontier of the DAG. Since changes are now uniquely and efficiently addressed clients can always view the history at the point in time of each individual hash, or provide a group of hashes to observe the data at a point where multiple changes are simultaneously visible.

Clients can obtain the current set of frontier hashes for a node. However, unlike the revision counter from *etcd*, the set of frontier hashes is not guessable or predictable for clients. However, the revision field is typically used for addressing the *observed* history of the datastore, particularly during watch streams. When clients request watch updates for keys, they maintain a record of the last revision they encountered from an update. When they restart they can use this as an opaque identifier to the datastore as a placeholder to pick up from where they last observed. Since the revision counter is treated as opaque, the frontier hashes can be used similarly.

## 3.3 Durability

**Lesson**: Lack of individual change addressing leads to difficult durability management.
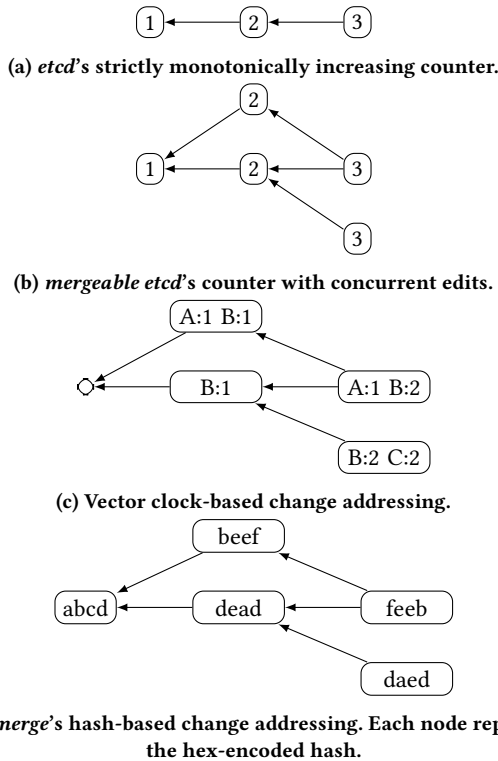
(a) *etcd*'s strictly monotonically increasing counter.



(b) *mergeable etcd*'s counter with concurrent edits.



(c) Vector clock-based change addressing.



(d) *dismerge*'s hash-based change addressing. Each node represents the hex-encoded hash.

Figure 4: Revision representations visualised.

```
#[derive(Reconcile, Hydrate, Serialize, Deserialize)]
struct Deployment {
  image: String,
  replicas: u32,
}
```

Listing 1: Example of using typed values. Based on a *Kubernetes Deployment* resource.
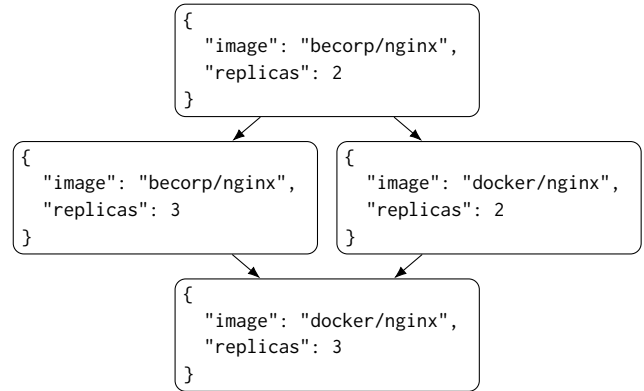


Figure 5: Example of concurrently modifying two values, based on the datatype from Listing 1.

## 3.4 Value representation

**Lesson**: Introspecting values at the datastore can provide semantic updates.

When *etcd* replicates changes to other nodes, obtaining consensus over them, the changes are made durable at each node before they acknowledge it. This ensures that, even in the event that the entire cluster restarts simultaneously, the change will still be accessible. When replication is lazy, as with causal consistency, the change is only made durable on the node processing the change before responding to the client. Upon replicating the change to other nodes it becomes durable on them, however, since this is a background process the client has no information about which nodes have received a given change.

As we have seen in the previous section, a revision counter prevents individual changes being addressed, posing an issue for detecting what nodes have made it durable. Importantly, a single revision counter means that clients must assume the change only ever has durability at the node it was performed at. However, using hashes for changes, and making them uniquely addressable, we regain the ability to query nodes for their durable changes. The information of what changes each node has can be included in the synchronization protocol such that a single node will be able to inform a client of the replication status of a change made there. Clients can then use this information to wait for a particular replication threshold for their changes to suit them.

Treating the values as opaque bytes, as *etcd* does, can make for efficient, and application agnostic, handling of requests. If *etcd* were to support structured values, such as JSON, it would still be going through consensus on the individual updates, despite them potentially being to distinct parts of the datatype. By enabling concurrent writes with *mergeable etcd* and *dismerge* using raw bytes for values, the conflict-resolution is very coarse-grained, being at the level of entire values. Supporting introspection of the value, based on a datatype, natively enables the datastore to be able to provide more fine-grained conflict-resolution, such as allowing concurrent mutations to different parts of the datatype. For instance, for orchestration workloads we may have two controllers operating concurrently that perform separate jobs. One is responsible for updating the image to point to the correct location, the other is an autoscaler, responsible for ensuring enough instances of the application are available to handle the demand. In *etcd*, these updates must happen one before the other, requiring the second to re-apply the update locally before sending to the datastore again, effectively being last-writer wins. With *mergeable etcd* and *dismerge* though, the updates do not need to be strictly ordered, they will merge together when both changes are present at a datastore node. Figure 5 highlights this difference based on the datatype in Listing 1.
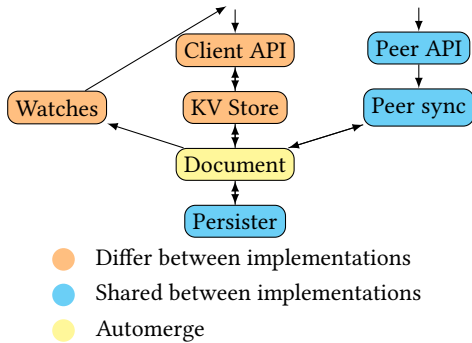
Figure 6: *mergeable etcd* and *dismerge* architecture.

## 4 IMPLEMENTATION

*mergeable etcd* and *dismerge* share a similar architecture, serving an *etcd*-like API but based around a CRDT document to enable decentralized operation. They are both implemented in Rust using the Automerge CRDT library at the core. The Automerge CRDT document is single-threaded with other threads in used to handle client requests. For persistence they can use either an in-memory store, a raw filesystem, or an embedded key-value store.

Additionally, *dismerge* no longer needs to track the revision counter and related fields: create revision and mod revision for each value. It also adds the API implementation for tracking replication status with peers as well as logic for calculating the responses.

### 4.1 Architecture

Figure 6 shows the architecture of *mergeable etcd* and *dismerge*. Both datastores focus on being *horizontally* scalable rather than *vertically* scalable. This is in order to span multiple edge sites for availability, rather than large single site deployments. As such, they do not use up all available cores, instead using only a few threads. Requests pass through the *etcd*-compatible gRPC API and into the key-value store. This key-value store contains an Automerge [1] CRDT document of keys and values. Changes to the document are prepared in this module before being persisted to disk through the persister. Once the changes have been persisted they pass back up through the gRPC API to the client. On the return through the KV store the updated value gets propagated to any watchers and the syncing thread is notified of changes so that it can share the updates with peers.

Operations on the Automerge CRDT document are single-threaded, focusing on limited edge resources, using other available threads to scale client request protocol handling, making changes durable, and communicating with peers.

### 4.2 Data model

Listing 2 shows the data model for *mergeable etcd*, stored in the Automerge document with some example data. The kvs is the main storage for key-value data with each key having a map of the revisions that exist for it. Deleted values are represented by null at the given revision. This enables efficiently handling queries for current and past data. Each key can also have an associated

```
{   "kvs": {
        "key1": { "revs": {
                    "001": [118, 97, ...],
                    "003": null },
                "lease_id": 1 }
    },
    "leases": { "1": null },
    "cluster": { "cluster_id": 2,
                "revision": 3 },
    "members": {
        0: { "name": "default",
            "peer_urls": [],
            "client_urls": [] }
    }
}
```

Listing 2: Data model for *mergeable etcd*. Values under *revs* are the encoded bytes.

lease identifier, which is only applicable to the latest value of the data. Leases are stored separately in the leases key to support efficiently enumerating possible leases in the datastore. Metadata about the cluster is stored in the cluster key including the ID of the cluster and the current revision. Finally, the list of cluster members is stored in the members key, mapping their ID to their name, URLs for peer connections, and URLs for client connections.

Listing 3 shows the data model for *dismerge*, stored in the Automerge document with some example data. It shares most aspects with *mergeable etcd*'s data model, namely leases and members. The kvs is the main storage for key-value data with each key storing the latest value and the ID of any lease associated with it, rather than the entire history. This does not need to store the entire history as that is maintained within and queryable from Automerge directly. Deleted values have no key in the kvs object. Metadata about the cluster is stored in the cluster key but notably no revision field is needed compared to *mergeable etcd* as the hashes of the document are obtainable from Automerge.

These data models grow with each client update, enabling historical queries but incurring an overhead to store all the data. *etcd* supports compaction of the revision history to reduce the storage space, preventing access to revisions older than the compaction point. This is not directly supported in *mergeable etcd* or *dismerge* due to a lack of support for *garbage collection* in Automerge at present, though support is available in other libraries [10].

*Consistent initialization.* To ensure that all nodes in a cluster can accept and merge changes from peers they need to start with a consistent state. Initialization logic on each node sets this up in a consistent way on first start by setting the document's actor ID to 0 and creating empty objects for the key-values, server meta information, members, and leases. For *mergeable etcd* this initialization also sets the initial revision to 1. This creates a change with a predictable hash from which all changes can branch off from.

### 4.3 API Guarantees

While retaining the same wire-level API, the change of consistency model impacts the guarantees that *mergeable etcd* can make. The

**Table 3: API guarantee comparison of the datastores.**

| Store | Atomicity | Durability | Consistency | Write ordering | Watch events | Revision uniqueness |
|-------|-----------|------------|-------------|----------------|--------------|---------------------|
| *etcd* | Yes | Majority | Linearizability | Total order | Unordered, complete | Globally |
| *mergeable etcd* | Yes | Locally | Causal | Partial order | Unordered, incomplete | Pre-conflict |
| *dismerge* | Yes | Locally | Causal | Partial order | Unordered, complete | Globally |

```
{   "kvs": {
        "key1": { "value": [118, 97, ...],
                  "lease_id": 1 }
    },
    "leases": { "1": null },
    "cluster": { "cluster_id": 2 },
    "members": {
        0: { "name": "default",
             "peer_urls": [],
             "client_urls": [] }
    }
}
```

**Listing 3: Data model for *dismerge*. Values under *value* are the encoded bytes.**



**Figure 7: Example of the synchronization process. The message from Node 1 to Node 3 gets lost and later Node 3 obtains the change via periodic sync.**

adaptations with respect are highlighted in Table 3. Atomicity refers to how operations are performed: *mergeable etcd* performs them atomically originally, but merging can make the result non atomic, due to the lack of unique revision addressing. *dismerge* provides atomic request handling due to the unique addresses. For durability, *mergeable etcd* and *dismerge* both only persist to the local node before returning to the client to avoid reliance on the network connectivity to other nodes. *mergeable etcd* and *dismerge* both also provide only partial ordering of writes, that is due to writes being able to be processed at different nodes concurrently, before synchronizing the nodes and merging the data. Watch events are always unordered, particularly as for *dismerge* there is no total order to base them off. Notably, *mergeable etcd* can send incomplete watch events: those that may not contain all of the modifications for that revision related to the watch; this is because merging other changes from peers can mutate an old revision, leading to previously sent watch event being potentially incomplete. Merging changes in *dismerge* can never modify an existing revision, and so the watch events are always complete. Revisions for *mergeable etcd* are also only unique before a node synchronizes with another that has a different operation at the same revision; that is: the revisions are only unique pre-conflict. *dismerge* avoids this by bringing back globally unique addresses suitable capturing the causality more accurately.

## 4.4 Durability

*etcd* stores the contents of the datastore on-disk using the bolt [3] embedded key-value database. It uses a flat structure to store the values at all revisions in history, up to the point of the last compaction. *mergeable etcd* stores values in an Automerge document. Doing so produces *changes* that encapsulate the operations performed to the document. It is these *changes* that *mergeable etcd*

persists in its embedded key-value database on-disk. This does mean that the document needs to be loaded into memory before it is queryable, so *mergeable etcd* can end up using more memory than *etcd* to hold the actual document. Making CRDTs space-efficient, in both in-memory and on-disk formats, is currently an active area of work [20, 24].

## 4.5 Synchronization

Automerge is an operation-based CRDT, meaning that it only needs to send changes that the peer does not already have, rather than the full state. *mergeable etcd* and *dismerge* split synchronization into two main cases: optimistic and pessimistic. In optimistic synchronization, a node immediately broadcasts a change, generated from a client request, to its synchronization peers. This enables fast replication in the best-case, when network the network is partition-free. This method is very simple, making it low-overhead and efficient to implement. Changes are not forwarded past the initial synchronization peer. When the network has partitions, these changes may be missed by peers, or peers may not be in the synchronization peers of a node, but should get the change. To solve this, pessimistic periodic synchronization is performed. This synchronization uses the protocol built into Automerge, based on Kleppmann and Howard's Byzantine Eventual Consistency protocol [25] to synchronize the changes. The small number of round trips, typically one, required to synchronize aids in minimising the resource requirements and latency when peers have diverged. Peers propagate all seen changes, enabling transitive connectivity of nodes. Periodic replication has more overhead than optimistically broadcasting changes as it has to calculate the set of changes to send from the document based on an estimation of what the peer has. Figure 8 highlights this; producing changes is equivalent to the optimistic broadcasting. Additionally,
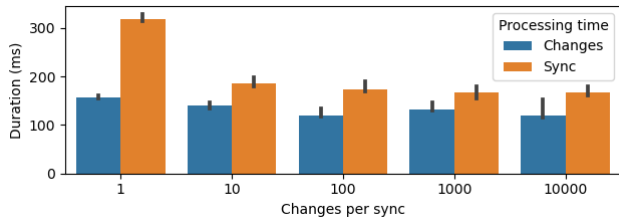
Figure 8: Time spent producing changes and performing periodic synchronization. Two documents concurrently producing an equal number of changes before synchronizing. Each change writes a new value to a shared key. 10,000 changes performed in total with 10 repeats.
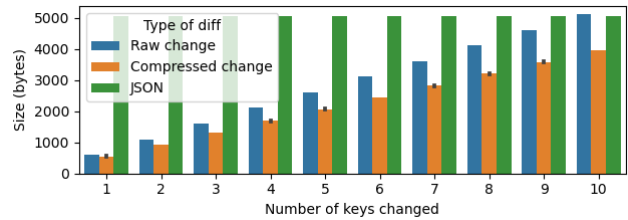


Figure 9: Size of change diff in varying over the number of keys changed. Keys were integers, values were random strings of 500 characters. The JSON case is the size of the total JSON-encoded data.

this has to be done on a peer-by-peer basis, adding extra load with more peer connections.

The topology of a *mergeable etcd* cluster is a complete network. This is based off of the architecture for *etcd* since leaders should be able to communicate with a majority of nodes. However, given *mergeable etcd*'s design to scale horizontally, this communication can quickly become cumbersome due to $O(n^2)$ connections for $n$ nodes. This becomes less of a concern as the synchronization of changes is transitive and the protocol rarely sends changes peers already have. Alternatively, instead of using a complete network, *mergeable etcd* can be configured with a list of peers to communicate with which form a subgraph of the network. It is the responsibility of the operator to configure this subgraph and to ensure that there is sufficient redundancy in the deployment. Future work could extend the peer communication to share addresses of nodes and actively monitor and build a topology based on environmental factors such as latency and redundancy. This would ease operational aspects of the cluster while also being able to react internally to failures and changes in cluster membership. However, this is left as future work due to it being highly dependent on deployment scenario.

### 4.6 Typing the values

Treating the values as opaque bytes, as *etcd* does, can make for efficient handling of requests but forces last-writer-wins semantics when doing conflict resolution with CRDTs. In practice, these opaque bytes often have a structure similar to JSON, consisting of nested maps and lists. Since Automerge supports JSON datatypes natively we can offer improved behaviour under conflicting updates to values. *mergeable etcd* and *dismerge* clusters can be specialised to custom datatypes for values that will be stored in the cluster. This specialisation is performed at compile-time using a operator-provided implementation provided in Rust, Listing 1. This implementation is responsible for parsing the bytes from the wire representation into its datatype and updating the stored value in the CRDT, enabling capturing the intent of changes. For reads, the implementation is responsible for extracting the value from the CRDT and converting it to bytes to send on the wire. For instance, if updating items in a JSON dictionary then the conflict resolution can allow concurrent edits to different keys easily rather than just accepting one of the objects. We provide pre-built variants of the

datastores supporting raw bytes as well as JSON. Applications using a specialised variant of *mergeable etcd* or *dismerge* with custom datatypes can also handle translation of data to prior and future schemas as well as validation of datastored. Using custom datatypes also enables more complex datatypes to be used, for instance using counters rather than plain integers or enriching datastored to support other conflict resolution strategies.

Due to the custom datatypes producing minimal *diff*s of the value, this can reduce the amount of data to replicate and persist, Figure 9 highlights this over a number of keys being changed. For the edge environment, this can drastically reduce extra traffic between sites, leaving more bandwidth for user traffic. Each change in the datastore has additional, small, constant overhead beyond the bytes to encode the diff, this is particularly optimised in *mergeable etcd* where multiple client operations are grouped into a single change, whereas each client operation in *dismerge* creates a new change.

### 4.7 Exposed replication status

Now that the datastore's history can be addressed uniquely, we can expose more details to the clients. One key item is that clients may have differing requirements for the replication of their values before acting on them. *dismerge* can accommodate this by informing them of the replication status of a set of frontier hashes. On each synchronization with peers (periodic synchronization), a node gets an update of what the heads of the other nodes are, this also includes a notion of what frontier hashes both nodes have in common. From this, and a set of frontier hashes a client is interested in, the node can calculate which peer nodes have the change. This is limited to direct peers of a node but clients can iteratively query other nodes to gather more information if desired. With this information, clients can dynamically choose their replication factor without placing a significant extra burden on the server. This API is available as a unary endpoint where the client sends a request for a set of frontier hashes and receives a single response indicating, for each peer, whether they have the change corresponding to the hash.

### 4.8 Model overheads

Since *mergeable etcd* does not leverage the hash graph of Automerge it can batch multiple operations into a single *change*. By leveraging the hash graph for addressing *changes*, *dismerge* requires each
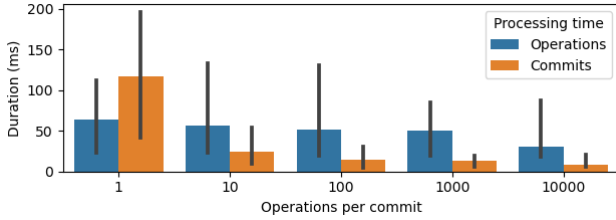
Figure 10: Time spent on operations and commits in Automerge varying operation counts per commit. Each operation writes to the same key in a *map*. Run for 10,000 operations with 100 repeats.

client operation to be in a separate *change*. This leads to a trade-off in the time spent processing the operations and the overhead of *committing* each *change*, explored in Figure 10. During committing of a change there is a need to calculate the hash of the encoded representation. This adds an overhead to processing a given number of client requests to serialize metadata for the change as well as the operation, before hashing. This can also impact the performance of individual operations due to the cache locality of data. Lower level changes in Automerge may be possible to optimise the overhead of calculating the hash but we do not delve into this in this work.

## 5 EVALUATION

We evaluate both *mergeable etcd* and *dismerge* in comparison to *etcd* starting at an edge-like deployment and then working towards a single node setup.

(1) How do *mergeable etcd* and *dismerge* handle a partition compared to *etcd*, particularly at scale? Section 5.2
(2) Assuming a reliable network without partitions, how does this change the performance of *etcd* at scale compared to the others? Section 5.3
(3) How would this performance differ if we were in a data-center-like environment? Section 5.4
(4) What overhead do *mergeable etcd* and *dismerge* add for single-node performance, given that clients will be working with their local node? Section 5.5

### 5.1 Setup

Benchmarks were run on a single Azure `Standard D64ds v5` (64 vcpus, 256 GiB memory) machine, running Ubuntu 20.04, with 3 repeats. Load is generated using an open-loop load generator and uses the YCSB workload A, which issues an equal ratio of updates and reads uniformly spread across the keyspace. All requests were sent to a single node, to mimic a workload at a single edge site, and load was sustained for 5 seconds. Keys are 18 bytes and values are 32 bytes, randomly generated. Each datastore node was run in a Docker container and limited to 2 CPUs to mimic limited edge resources. The datastore nodes are backed with a *tmpfs* to minimise the impact of disk latency. No additional latency is added between the nodes unless specified. All results presented are for successful requests. The setup models a client interacting with its local datastore node only, relying on it to process the operations. The client initially



(a) Latency of successful requests.



(b) Latency of failed requests by error condition, only from *etcd*.

Figure 11: Workload applied to a three node cluster. The leader node is partitioned from the cluster at approximately 5 seconds into the experiment, and this is cleared at 10 seconds in (dashed vertical lines). The $y$ axis is log-based.

connects directly to the local leader node, this avoids forwarding overhead in *etcd*. When the leader node is partitioned from the rest of the cluster, the leader will change and, after the partition heals, the client may be connected to a non-leader node. Partitions were injected with the use of `iptables`, delays were injected with the Linux `traffic controller` with a variation of 10% and a correlation of 25%.

### 5.2 Starting at the edge

At the edge, applications will be deployed across sites, needing to share data between these. The sites are geographically distributed with limited resources at each, network links can also be unreliable. As such, this section works within this context with a setup of three nodes spread over sites, connected over a 10ms link. The client is co-located with a node, initially the leader node and a partition is injected between the leader node and the rest of the cluster at approximately 5 seconds, before being healed at 10 seconds into the experiment.

Figure 11a shows the results of this experiment for each datastore. Initially, *etcd* has a higher latency due to the latency of the network between the nodes. During the partitioned period *etcd* is unable to service requests, internally queueing them until they time out. This is what leads to some requests issued before the partition heals to be processed. When the partition is healed the local node also has an overload of requests, as shown by the "too many requests" errors in Figure 11b. During this recovery time, the local node is also trying to obtain who the new leader is and forward requests to them for processing. This further exacerbates the latency of successful requests, and leads to more overload. Requests that end up being successfully handled after the partition is healed and a
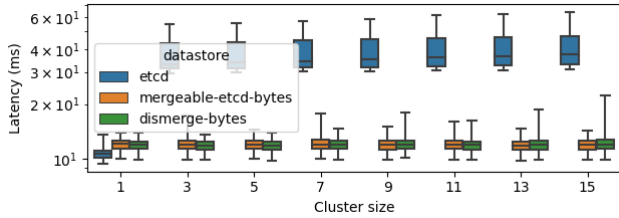
**Figure 12: Latency box plot of multiple nodes with 10ms latency on each link. Whiskers extend from the 1st to the 99th percentile. The $y$ axis is log-based.**



**Figure 13: Latency box plot of multiple nodes. Whiskers extend from the 1st to the 99th percentile.**

steady state is obtained now incur higher latency as the local node is no longer a leader, it must forward each request.

*mergeable etcd* and *dismerge* are able to continue processing requests during the partition, holding changes to be synchronized until the partition heals. This maintains reliable performance during the disruption and avoids costly recovery overheads after. The periodic synchronization will ensure that replicas obtain all of the missed changes.

### 5.3 Making the network reliable

Assuming that the network will be reliable, not experiencing partitions, we can view how the latency of the network affects the scale of the cluster more directly. This setup follows that of the previous section but no partition is injected during the experiment run, and so the leader node remains stable. Due to *etcd*'s eager replication, it is very sensitive to the performance of the network. Figure 12 presents plots of the latency distribution and peak throughput across different cluster sizes. Cluster sizes are generated from the $2f + 1$ function for *etcd* to maximise failure tolerance for $f$ failures.

For single node deployments there is no network latency incurred as no replication is performed. However, when adding nodes *etcd*'s latency drastically increases due to its requirement to replicate data to a majority of nodes in the processing of a request. As the cluster size increases, this incurs a marginal overhead to communicate with the nodes in the cluster. This highlights *etcd*'s sensitivity to the network latency for processing requests. This also makes the assumption that all links are homogeneous, in reality they are likely to be heterogeneous due to their geographical distribution and so some remote nodes could drastically impact the latency characteristics. This is further worsened when the leader changes as it could change to a site with slower connections to a majority, bottlenecking all requests on a single slow link.

*mergeable etcd* and *dismerge*, moving eager communication off the critical path, enable more consistently low-latency operation, even at larger cluster scales. They too will incur an overhead of communicating with a larger number of peers but this is expected to be significantly lower than the delay added to *etcd* due to the network latency. This can also be managed by not connecting all nodes to all nodes, instead forming a mesh network.
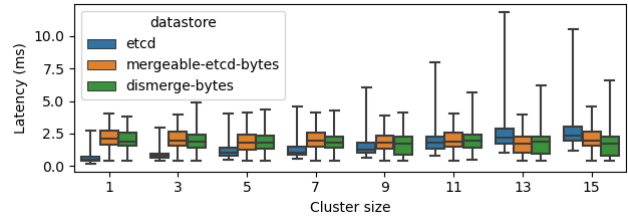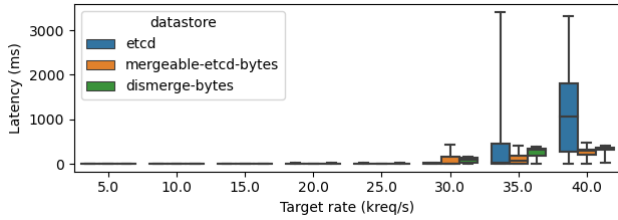
### 5.4 Providing an optimal network

Since *etcd* is targeted for cloud data center deployments we now evaluate its scalability in a setting with no latency, but still limited resources. This also highlights the overhead of added fault tolerance, something which may still be important to cloud applications and which may limit the resources each node can have. The impact of varying the cluster sizes can be observed in Figure 13, under a target rate of 10,000 requests per second. Generally, *etcd* encounters scaling issues in terms of latency with the increase in cluster size. Due to *etcd*'s optimised implementation, *mergeable etcd* and *dismerge* currently have a higher, but still small, fixed cost. Despite this and our analysis in the previous section suggesting that the overhead of communicating with more nodes is marginal for *etcd*, we observe that there is indeed an overhead incurred by *etcd* which seems to be non-trivial compared to the performance of small clusters. This trend implies a cross-over point where clusters of *etcd* with no latency overhead become less performant than *mergeable etcd* and *dismerge*. We project *etcd*'s latency to continue to get worse as cluster size increases due to the fundamentally increasing amount of work that the leader node must perform to replicate values and the eager nature of this.
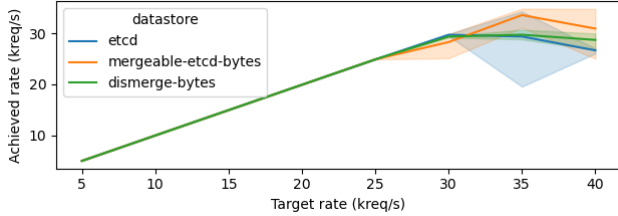
### 5.5 Collapsing the cluster

To compare the raw overhead of the data model that *mergeable etcd* and *dismerge* use internally we compare results of a single node handling requests. This avoids conflation with the synchronization process. From Figures 14a and 14b, we observe that all datastores can handle the load up to around 30,000 requests per second, after which throughput drops off for all. However, after this point *etcd* suffers significantly higher latency, not efficiently shedding or rejecting load. We can also observe the higher overhead within *dismerge* compared to *mergeable etcd* at higher rates due to the overhead of extra commits, discussed previously in Section 4.8.

Looking at Figure 14c we can observe that for lower rates *etcd* outperforms both *mergeable etcd* and *dismerge* in terms of latency. This is expected due to the extra overheads that the CRDT logic impose upon *mergeable etcd* and *dismerge*. When processing a write request, *etcd* simply needs to write it to the in-memory maps and caches before persisting the write, which is effectively a no-op due to the *tmpfs*.
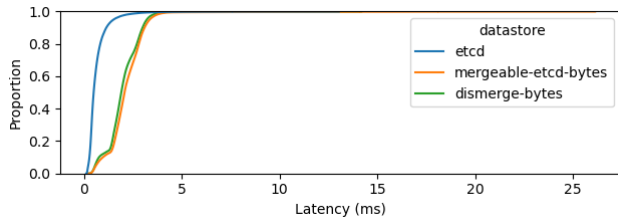
Errors begun to occur from the datastores from 30,000 requests per second.

(a) **Latency box plot. Whiskers extend from the 1st to the 99th percentile.**



(b) **Comparison of achieved rate with respect to the target rate. Repeat variance shown by the shaded region.**



(c) **Latency CDF at 10,000 requests per second to highlight differences at lower loads.**

**Figure 14: Single node results.**

## 6 IMPLICATIONS ON APPLICATIONS

Given the trade of linearizability for causal consistency applications may not function correctly without modification. More subtly, the difference in replication of data from eager to lazy can impact the durability guarantees of applications. Taking *Kubernetes* as an example application that already relies on *etcd*, we can imagine replacing it with *mergeable etcd*. *Kubernetes* primarily operates by storing a desired state of resources in *etcd*. This central view of the data is then acted upon by *controllers* that *watch* the data and react to new versions. These controllers perform operations such as creating new container resources (*Pods*) which are delivered to the scheduler and then allocated to a node. *Controllers* also handle higher-level resources such as *Deployments* which dictate the number of containers running in the cluster for a particular application.

Due to the controllers present in *Kubernetes*, we believe that *mergeable etcd* would enable it to remain functional. Any discrepancy of the data due to merging of concurrent interactions will be acted upon by the *controllers* and corrected. Importantly, *mergeable etcd* does not impact the integrity of the values, only which value would be presented. It is unclear whether *Kubernetes* provides stable

and sufficiently dampened control loops to handle higher latency between the datastore nodes, and thus more temporary divergence.

Under this model every partition of the datastore cluster effectively creates a replica of the entire cluster, starting new instances of applications on both sides of the partition to ensure replica counts are met. When the partition heals and the datastore nodes synchronize, there will be one cluster again the controllers will drive the state to that of the single cluster again.

One problematic piece of *Kubernetes* would be its guarantee of unique *Pod* names. This is typically not an issue as *Deployments* create *Pods* with randomised names, preventing collisions. However, *Kubernetes* manages stateful deployments with a *StatefulSet* which assigns numerically increasing names for the *Pods*. This could lead to multiple *Pods* with the same name existing in the cluster due to the weaker consistency in the datastore. One possible mitigation is to have the site-local *StatefulSet* controllers only manage the instances at their site, injecting a suffix for the site name into the pod name to make them unique again.

*Kubernetes*, storing resource definitions as a JSON-like protobuf schema, would be a prime candidate for exploring the use of the typed values in *mergeable etcd*. For instance, replica counts on *Deployment* resources could be modified concurrently to the other fields, such as the container image to be run. This enables concurrent updates to take effect, rather than requiring the initiators to retry their requests. For *Deployments* this is of interest to even higher-level controllers that might be in charge of updating the image or providing dynamic scaling.

Integrating *dismerge* into *Kubernetes* would be more invasive due changes in how history is addressed, but should be feasible under the above discussion and ultimately lead to a more intuitive model due to its immutable history.

## 7 RELATED WORK

Anna [36] is a distributed key-value store that targets performance at both single node and cloud-scale through a system of coordination-free actors. Anna also uses CRDTs for storage though uses a custom implementation rather than a library. Anna focuses on the core functionality of a distributed key-value store, not implementing related functionality such as watching keys. As such, it is not a direct competitor to *mergeable etcd* but provides good lessons if *mergeable etcd* were to need scaling to cloud-scale workloads.

Azure's CosmosDB [11] is a closed-source NoSQL database that provides many different consistency levels and with different API compatibility layers. This allowed CosmosDB to expose an *etcd*-compatible API whilst changing the consistency levels dynamically [2]. The database can also produce reports of the staleness of the data returned, enabling insight into the support of the application for weaker consistency levels which may lead to performance improvements.

## 8 CONCLUSION

Deploying platforms and applications near to the edge provides new challenges in delivering higher-level requirements. From these we derive lower-level key-value datastore requirements and show how *etcd* is unsuited to meet these. We explore the design space under these requirements, focusing on consistency, history addressing,

durability and value representation. This exploration then instigates the implementation of two new datastores, successively adapting *etcd* to be edge-suitable: *mergeable etcd* and *dismerge*. These datastores offer applications reliable local-first operation, enabling applications to continue operating under unreliable network conditions found at the edge. The performance is also considerably enhanced compared to *etcd*, providing consistent low-latency operation. Due to *etcd*'s popularity as a critical distributed key-value store, we envision new avenues for work focusing on local-first edge applications, avoiding eager coordination with other sites. Furthermore, this can be extended to cloud environments to enhance reliability as both *mergeable etcd* and *dismerge* offer competitive performance and can be scaled to much greater extents. More broadly, this work highlights a transition from servers being co-located with each other with distributed clients, to servers being co-located with clients but being distributed from other servers.

# REFERENCES

[1] Automerge. https://github.com/automerge/automerge-rs.
[2] Azure cosmos db api for etcd in preview. https://azure.microsoft.com/en-us/updates/azure-cosmos-db-api-for-etcd-in-preview/.
[3] Bbolt: An embedded key/value database for go. https://github.com/etcd-io/bbolt.
[4] Etcd website. https://etcd.io.
[5] K3s: Lightweight kubernetes. https://github.com/k3s-io/k3s.
[6] Kubeedge: Kubernetes native edge computing framework. https://kubeedge.io/en/.
[7] Kubernetes. https://kubernetes.io.
[8] Kv api guarantees made by etcd. https://etcd.io/docs/v3.4/learning/api_guarantees/.
[9] Why large organizations trust kubernetes. https://tanzu.vmware.com/content/blog/why-large-organizations-trust-kubernetes.
[10] Yjs garbage collection. https://docs.yjs.dev/api/y.doc#y.doc-api.
[11] Microsoft Azure. Azure cosmos db. https://azure.microsoft.com/en-gb/services/cosmos-db/.
[12] Kenneth Church, Albert Greenberg, and James Hamilton. On delivering embarrassingly distributed cloud services. 01 2008.
[13] The Git community. Git. https://git-scm.com/.
[14] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, apr 1988. https://doi.org/10.1145/42282.42283.
[15] Apache Foundation. Apache cassandra. https://cassandra.apache.org/_/index.html.
[16] GitHub. Coredns issues for etcd. https://github.com/coredns/coredns/issues?q=is%3Aissue+etcd+.
[17] GitHub. Kubernetes issues for etcd and scalability. https://github.com/kubernetes/kubernetes/issues?q=is%3Aissue+etcd+label%3Asig%2Fscalability.
[18] GitHub. M3 issues for etcd. https://github.com/m3db/m3/issues?q=is%3Aissue+etcd+.
[19] GitHub. Rook issues for etcd. https://github.com/rook/rook/issues?q=is%3Aissue+etcd+.
[20] Alex Good and Andrew Jeffery. Binary document format. https://alexjg.github.io/automerge-storage-docs/.
[21] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
[22] Andrew Jeffery, Heidi Howard, and Richard Mortier. Rearchitecting Kubernetes for the edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '21, page 7–12, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3434770.3459730.
[23] Chris Jensen, Heidi Howard, and Richard Mortier. Examining raft's behaviour during partial network failures. In *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*, HAOC '21, page 11–17, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3447851.3458739.
[24] Martin Kleppmann. Crdts: The hard parts. https://martin.kleppmann.com/2020/07/06/crdt-hard-parts-hydra.html.
[25] Martin Kleppmann and Heidi Howard. Byzantine eventual consistency and the fundamental limits of peer-to-peer databases, 2020.
[26] Michał Król, Spyridon Mastorakis, David Oran, and Dirk Kutscher. Compute first networking: Distributed computing meets icn. In *Proceedings of the 6th ACM Conference on Information-Centric Networking*, ICN '19, page 67–77, New York, NY, USA, 2019. Association for Computing Machinery.
[27] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.
[28] Mihai Letia, Nuno Preguiça, and Marc Shapiro. Consistency without concurrency control in large, dynamic systems. *ACM SIGOPS Operating Systems Review*, 44(2):29–34, April 2010.
[29] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, et al. Consistency, availability, and convergence. *University of Texas at Austin Tech Report*, 11:158, 2011.
[30] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.
[31] OpenAI. Scaling kubernetes to 7,500 nodes. https://openai.com/research/scaling-kubernetes-to-7500-nodes.
[32] Gang Peng. Cdn: Content distribution network, 2004.
[33] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, pages 386–400. Springer LNCS volume 6976, October 2011.
[34] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2741948.2741964.
[35] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)*, 49(1):1–34, 2016.
[36] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 33(2):344–358, 2021.