ChiaVe: A Flexible CRDT-Based Key-Value Store

Rafael Chaves
Cornell University
rvc29@cornell.edu

Abstract

ChiaVe[1] is a distributed key-value store built on top of conflict-free replicated data types (CRDTs). ChiaVe achieves high scalability and availability by replicating data through message-passing, both across nodes and across threads within a node.

The use of CRDTs yields desirable properties such as coordination-free strong eventual consistency and automatic conflict resolution. Though the use of these data structures can be potentially limiting in some applications, certain sophisticated data structures can still be constructed via the composition of smaller CRDTs. ChiaVe provides a rich interface for manipulating counters, sets, and register CRDTs.

Unlike other key-value stores, ChiaVe provides three different implementations of CRDTs: operation-based CRDTs, state-based CRDTs, and delta-state CRDTs (δ-CRDTs), unifying them under a simple, generic interface. ChiaVe builds Dotted Version Vectors [4] into its CRDT implementations in order to efficiently detect and resolve concurrent updates. ChiaVe is also one of the first open-source key-value stores that provides implementations of δ-CRDTs.

ChiaVe is written in Go, and its modular design provides a foundation for future implementers to swap out the storage medium, internal data structures, and the number of worker processes, automatically obtaining benchmarks related to latency, throughput, and convergence time. The project is open-source and can be found at https://github.com/rafaelvchaves/chiave.

1 Introduction

The CAP theorem states that in the event of a network partition in a distributed system, one must choose between availability and consistency. In most cases, availability is prioritized and a compromise is made for weaker consistency guarantees, commonly eventual consistency. However, even achieving eventual consistency is quite difficult to get correct, often requiring the implementation of complex consensus or rollback protocols to resolve conflicting updates [19]. This synchronization requirement also negatively impacts latency and scalability.

Conflict-free replicated data types (CRDTs) are data structures with desirable mathematical properties that allow them to be replicated across multiple servers and updated independently. They also allow for the automatic resolution of any conflicting updates that occur. They have been shown to provide strong eventual consistency (SEC) [20], which adds to eventual consistency the safety property that any two replicas that have received the same set of updates (in any order) will be in the same state. This results in CRDT-based systems being highly available, allowing for up to $n-1$ simultaneous failures over a group of n replicas.

There are two primary implementations of CRDTs: state-based CRDTs and operation-based CRDTs, which have been proven to be theoretically equal in their strong eventual consistency guarantees [20]. However, they can vary quite a bit in their practical implications. In recent years, delta state-based CRDTs (δ-CRDTs) have been explored as an optimized version of state-based CRDTs [1].

Given the differences between the CRDT types, the goal of this paper is to provide a flexible key-value store that can easily interchange between these three types to suit different application needs. A secondary result of the paper is a repository of CRDT implementations in Go, which are not nearly as common as C++ or JavaScript implementations.

The unique properties of CRDTs guide ChiaVe towards a different architecture than traditional key-value stores. Since conflict resolution can be done automatically across replicas, ChiaVe is not only able to distribute data across nodes, but also on multiple actor threads within a node. Rather than sharing memory, actors can operate independently, and periodically exchange any state changes that they encounter. This makes the system highly scalable and adaptable to a range of environments, from a single multi-core machine to a geographically distributed cluster of machines.

The structure of the paper is as follows. Section 2 discusses

[1] "chiave" is Italian for "key", and the V is stylized for "value".

related work in the area of key-value stores and CRDTs. Section 3 provides a more in-depth background on the three different CRDT types. Section 4 discusses the implementation of CRDTs in ChiaVe. Section 5 provides the architecture of the system. Section 6 evaluates system performance with respect to update latency, throughput, and time to reach convergence for various replication factors, workloads, and payload sizes.

2 Related Work

The idea of using CRDTs to implement large-scale distributed systems with strong eventual consistency goes back at least as far as 2011 [20]. Since then, CRDTs have been implemented in a number of applications, primarily in distributed databases [15,17,18], but also in text editors [6,23] and GPS navigation systems [21].

Other key-value stores such as DynamoDB [9], Cassandra [3], CouchDB [7], and DocumentDB [8] provide either limited or no support for CRDTs and do not partition the key space across cores in a single node; rather, they only partition across nodes. Memcached [12] and MongoDB [13] do partition across both nodes and cores, but do so through shared memory rather than message-passing.

Another popular key-value store, Redis [15], makes use of CRDTs to implement multi-master replication in its commercial version, Redis Enterprise [16]. However, it is not an open-source solution and only makes use of operation-based CRDTs. The open-source version of Redis does not support CRDTs, is not designed to support multi-threading on a single machine, and uses the master-slave replication model.

In terms of its architecture, ChiaVe borrows many ideas from Anna [22], another multi-mastered key-value store that also employs message-passing both between nodes and between threads. In Anna, these techniques were enabled by lattice-based composite data structures, which possess similar properties to state-based CRDTs. Anna also suffers performance hits for large payload sizes due to this approach, and ChiaVe looks at alternative approaches in operation-based and δ-CRDTs. ChiaVe's architecture also differs from Anna in a few areas: (1) ChiaVe places a leader process on each server to assist in re-routing client requests, detecting worker failures, and simplifying implementation; (2) ChiaVe employs CRDT-dependent mechanisms for propagating updates to other replicas, and does so over RPCs and Go's built-in buffered channels rather than network multicasts; (3) ChiaVe provides more efficient causality tracking with Anna by using Dotted Version Vectors rather than traditional version vectors. With ChiaVe, more emphasis is placed on providing a modular and extensible framework for future implementers to experiment with various CRDT implementations at scale under various environments.

Riak KV [17], a key-value store written in Erlang, provides state-based CRDT implementations for flags, registers, counters, sets, and others. It similar to Chiave and other distributed key-value stores in terms of utilization of the actor model to scale over multiple nodes. Although there have been proposals to implement δ-CRDTs in Riak in order to improve performance for large sets [5], they still do not exist in the open-source version.

3 Background

3.1 State-based CRDTS

This section describes in greater detail the difference between the three CRDT types: state-based, operation-based, and delta state-based.

Formally, a state-based CRDT is described by a tuple $(S, Q, M, merge)$, where S is the set of values that the data type can take on, Q is a set of query methods on the data type, M is a set of mutators, or operations, that can be performed on the data type, and $merge : S \times S \to S$ is a special operation matching the following properties [1]:

- **Commutativity:**
  $\forall s_1, s_2 \in S,\ merge(s_1, s_2) = merge(s_2, s_1)$
- **Associativity:**
  $\forall s_1, s_2, s_3 \in S,$
  $merge(s_1, merge(s_2, s_3)) = merge(merge(s_1, s_2), s_3)$
- **Idempotence:**
  $\forall s \in S,\ merge(s, s) = s$

With state-based CRDTs, replicas maintain their own copy of the data and process incoming mutation requests by invoking the corresponding mutator $m \in M$ on their local state. Periodically, state-based replicas exchange states, applying the *merge* operator to their local state and the incoming state. The properties of the *merge* operator allow for all replicas to eventually converge to the same state, regardless of messages being lost, reordered, or duplicated.

3.2 Operation-based CRDTS

An operation-based CRDT is a tuple (S, Q, U) where S and Q are the same as before. However, rather than updating state via the set of mutators M and the *merge* operator, operation-based CRDTs use a set of update methods U. Each update method in U is a pair $u = (p, e)$, where p is a side-effect-free "prepare-update" method and e is a mutating "effect-update" method [20]. The prepare-update method takes in the current CRDT state S and returns some payload π. The corresponding effect-update method takes in π and mutates the CRDT state. When a CRDT replica receives an update, it executes the two methods sequentially on its local state, and then delivers π to all other replicas, where they obtain the new state $X' := e(X, \pi)$. Operation-based CRDTs keep track of metadata to determine a causal history of updates, and must satisfy the property that concurrent calls to its effect-update method will result in the same state regardless of ordering. Additionally, the update methods do not need to be idempotent

or associative. These relaxed requirements make operation-based CRDTs more convenient to implement, but it becomes critical that a given update is delivered exactly once to each replica via a reliable broadcast channel.

Although state-based CRDTs require fewer assumptions about the communication system, they come with the disadvantage that a replica must ship the entirety of its state to others to perform a merge operation, which can become quite costly as the data structure increases in size. A recent development has been the introduction of δ-CRDTs, where replicas are able to send smaller payloads that can represent the changes that occurred in the state.

3.3 Delta CRDTS

Formally, a δ-CRDT is a tuple $(S, Q, M^{\delta}, merge)$, where S, Q, and *merge* are defined exactly as in state-based CRDTs [1]. M^{δ} is a set of *delta-mutators* $\{m_1^{\delta}, ..., m_n^{\delta}\}$, each of which is a function that corresponds to a specific update operation on the CRDT (for example, a counter δ-CRDT would have two delta-mutators, for the increment() and decrement() operations). A delta-mutator m_i^{δ}, given a current state $X \in S$, returns a "delta" state $\delta \in S$ that only encapsulates the changes that the update operation makes on the state. From this, a new state X' is obtained from the current state X by setting $X' := merge(X, \delta)$. The idea is then to ship only the smaller state δ to all replicas rather than the entire new state X' to reduce communication overhead. This introduces additional complexities associated with garbage-collecting deltas that have been observed by all replicas.

3.4 Dotted Version Vectors

In any key-value store, it is imperative to be able to efficiently and reliably detect when two updates on the same data have some causal relationship, or whether they are concurrent. Common approaches for this include vector clocks [11] and version vectors [14]; however, these come with some limitations, namely that they require maintaining an entry in the vector for each client of the key-value store. Since the number of clients can grow arbitrarily, systems that use client-ID-based version vectors suffer some performance costs for this. Systems that instead use server IDs for version vector entries can suffer from not being able to detect concurrent client updates, if for example they are both routed to the same server and processed sequentially.

A dotted version vector is a tuple $((r,n), v)$, where (r,n) is a "dot" denoting a globally unique event occurring at replica r, and v is a traditional vector clock that maps server IDs to sequence numbers, representing the causal past of the event. The addition of the dot provides information about possible events that a replica may have missed, as the dot may have an isolated sequence number that is outside the contiguous range of the version vector. Additionally, each dot acts as a unique event that occurred in the cluster. ChiaVe adapts DVVs to leverage the self-resolving properties of CRDTs when concurrent updates are detected.

4 CRDT Implementation

Unlike other key-value stores, ChiaVe provides op-based, state-based, and delta-based implementations for a few different data structures. The semantics and implementation of these data structures are described in this section.

ChiaVe currently supports the following data structures:

- **Counter:**
  A counter keeps track of a single signed integer, and supports increment() and decrement() operations for increasing or decreasing the value by 1, and a value() query for retrieving the counter's value.
- **Observed-Removed Set (ORSet):**
  An ORSet is a set of values, with a add() and remove() methods for adding and removing elements from the set, and value() method for retrieving the contents of a set. An ORSet possesses the semantics that if an element e is added to the set by a replica r_1, it can only be removed by a replica r_2 once r_2 observes this change made by r_1 (which occurs when r_1 exchanges state with r_2). This means that if the same element e is concurrently added and removed from the set, the addition will win, and e will eventually be present in all replicas.

These three implementations are combined into a single interface to easily integrate more data structures of any type in the future. The key to creating this interface was observing that each CRDT type involves some payload that encodes a set of updates that have been made to the state, and eventual consistency is achieved by communicating this payload between replicas in some way. Thus, ChiaVe packages these data structures into a general "Event" type, which contains information such as the ID of the source replica and the payload describing the update. Event types are defined using protocol buffers so that they can be efficiently serialized and sent over the network. Each CRDT specifies how to update itself with an event, and what events it needs to share with other replicas through the following methods:

```
type CRDT[F Flavor] interface {
    PrepareEvent() *pb.Event
    PersistEvent(*pb.Event)
}
```

For operation-based CRDTs, the PrepareEvent() method will return an event that contains the sequence of "effect updates" that the CRDT performed since the last time PrepareEvent() was called. In some cases, properties of this method can be exploited to reduce the size of the payload; for example, in a counter, the sequence of updates $\{+1, +1, -1, +1\}$ can be collapsed into a single event $\{+2\}$. When PersistEvent(e)

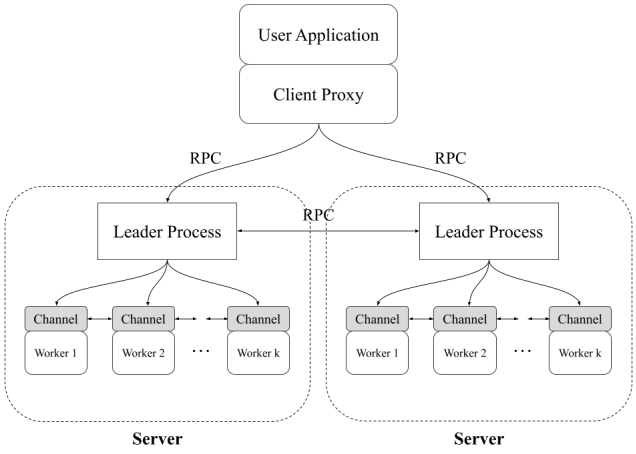

Figure 1: ChiaVe system architecture with 2 servers and k workers per server.

is called on an operation-based CRDT, the effect-update method is simply invoked on the current state with the argument $e.Payload$.

For state-based CRDTs, PrepareEvent() always returns a copy of the CRDTs entire state, and PersistEvent(e) invokes the CRDT's *merge* method on its local state and $e.Payload$, which the remote replica's state.

Like state-based CRDTs, δ-CRDTs invoke the *merge* operator on the their state and the remote state when PersistEvent(e) is called. The PrepareEvent() method returns the set of deltas that have been produced by all delta mutators invoked since the last time PrepareEvent() was called.

5 Architecture

The ChiaVe system layout is shown in Figure 1. A ChiaVe cluster consists of a set of servers that are accessed via one or more client proxies. Each server contains a collection of k worker threads, along with a single leader thread responsible for hosting the RPC server used by the proxy, as well as forwarding requests and update events to the correct worker. Given that the properties of CRDTs lend them to be updated independently by multiple replicas, ChiaVe employs a multi-primary approach to key replication, tunable by a user-provided replication factor r. In addition, rather than sharing memory, workers in a ChiaVe cluster exchange updates via Go channels for inter-server communication, and via remote procedure calls (RPCs) for intra-server communication.

5.1 Client Proxy

To use ChiaVe, a user creates an instance of the client proxy on their local machine. The client proxy currently provides the following interface for manipulating data types:

```
type CounterKey
type SetKey

type ClientProxy interface {
    Increment(key CounterKey)
    Decrement(key CounterKey)
    GetCounter(key CounterKey) int64
    AddSet(key SetKey, element string)
    RemoveSet(key SetKey, element string)
    GetSet(key SetKey) []string
}
```

ChiaVe uses consistent hashing to partition the key space across workers. Each worker is uniquely identified by a tuple (IP address, port number, worker ID) together called the "replica ID", which is the value that is hashed to find a worker's partition on the hash ring. Both the client proxy and the leaders must have knowledge of this hash ring so that they can forward user requests and events to the correct worker.

When a user makes a request on a specific key, the client proxy hashes the key and obtains a set O of r owners for the key via the hash ring, where r is the replication factor. It randomly shuffles this set to balance request load and then issues an RPC request to the corresponding owner. If the request fails, the client proxy proceeds to try the next owner in O. In the unlikely event that the client proxy exhausts this entire set without successfully completing an RPC, the proxy will notify the user that the request failed. Thus, a larger replication factor will result in a cluster with high availability, but will result in data taking longer to converge across all replicas.

5.2 Leader

When starting up a ChiaVe server, the CRDT type, replication factor, and number of workers is specified. A leader instance is then created, which spawns k worker threads, and then begins serving an RPC server at a fixed IP address and port number. From then, a leader services one of two requests: (1) an incoming *ProcessRequest* RPC sent by the client proxy; or (2) an incoming *ProcessEvent* RPC from another leader. To avoid creating a bottleneck in the system and to simplify implementation, the leader simply finds the correct worker to delegate the task to and puts the request or event on the corresponding channel.

5.3 Worker

Each ChiaVe worker maintains a map from a string to a CRDT of flavor $f \in \{\text{Delta}, \text{Op}, \text{State}\}$. In addition, each ChiaVe worker contains two buffered channels for intra-server communication: a request buffer and an event buffer. A request contains information about an operation a user wants to perform, while an event encapsulates an update that another worker has performed and has sent for other workers to apply.

Workers run in an asynchronous loop where they repeatedly consume client requests and events propagated by other workers. The workers constantly monitor their event channel for any events that the leader forwarded to it (this occurs when another worker sends the leader a *ProcessEvent* RPC). Each event describes the key to perform the update on. The worker retrieves this CRDT from their map and applies its *PersistEvent* method.

When a worker pops a client request from its request channel, it determines if it is a read or a write request. If it is a read request, the worker responds immediately to the leader with the data that currently exists in its map. If it is a write request, the worker applies the changes locally and adds the key to a "changeset", which is initialized to the empty set. The changeset is periodically propagated throughout this loop after some timeout called the "broadcast epoch". At the end of each epoch, a worker goes through each key in its changeset, and accesses the CRDT in its map that is associated with that key. It then calls the PrepareEvent() method on the CRDT, obtaining an event e. It uses the same consistent hashing algorithm as the client proxy to determine the set of owners O of the key, and for each owner $o \in O$, the worker sends a *ProcessEvent* RPC to o's leader, who places it on the correct worker's channel. If the request fails, the worker will perform a single retry. Note that this protocol is not currently sufficient to provide the exactly-once semantics that operation-based CRDTs require. This remains an active area of work in this project.

5.4 Configurability

One of ChiaVe's main design principles is to provide high customizability across multiple layers of the system. This allows users to easily swap in different components and run experiments to find a configuration that best suits their application's needs. Since all three CRDT types fall under the same central interface, it is easy to arbitrarily extend the supported data types, for example with graphs, registers, or text sequences. When starting up a ChiaVe cluster, the client proxy and leader processes read from the same configuration file, which specifies the replication factor, the number of nodes, the number of workers per node, the desired load factor on each key, and the type of CRDT.

ChiaVe also modularizes the underlying key-value store through the following interface:

```
type Store[F Flavor] interface {
    Get(string) (CRDT[F], bool)
    GetOrDefault(string, CRDT[F]) CRDT[F]
    Put(string, CRDT[F])
}
```

Currently, the only implementation provided is an in-memory hash table.

5.5 Causality Tracking

This section discusses how ChiaVe makes use of dotted version vectors to track causality across replica updates. Both client proxies and workers maintain dotted version vectors for each key in the key-value store. For the rest of this section, these dotted version vectors will be referred to as the "context". Every time a proxy makes a synchronous write request (such as an *AddSet* or *RemoveSet*), it sends the current state of its context for that key. The worker that handles the operation then performs an Update operation, which produces a new context u whose version vector is the same as the client's version vector, and whose dot is a pair $(i, n+1)$, where i is the worker's ID and n is the maximum sequence number of the worker prior to processing the request (sequence numbers are initialized to 0 on start-up). This context is then compared to the worker's previous context: if they are not detected to be concurrent, the worker simply sets its new context to be u. Otherwise, the worker invokes a Join operation that combines the histories of the two contexts. The worker will also obey any semantics imposed by the CRDT specification if the updates are conflicting; for example, if the client request was to remove an element e from an observed-removed set, and the element had been previously added in the set, it will only remove the element if it was contained in the causal history of the client (i.e. only if the element was observed).

Figure 2 shows a simple example, where two clients, c_1 and c_2, are performing updates that go to workers a and b. Initially, c_1's context is empty, and when replica a receives its request, it increments its sequence number and returns the updated context. c_1 then uses this new context to make another request, which this time goes to replica b. Replica b increments its sequence number and records that it has seen up to event 1 from replica a in the history vector. Concurrently, another client c_2 makes a request on the same key, which gets routed to replica b. At this point, replica b detects that this update is current with the dots $(a,1)$ and $(b,1)$, as they are not included in the context's history vector ($\{\}$). Replica b does any conflict resolution necessary, takes all of the events observed, including $(b,2)$, and records it in the causal history. At this point, client c_2 is now aware of all updates made by a up to event 1 and by b up to event 2.

6 Evaluation

In this section, we experimentally compare the three different CRDT types for sets. Each of the following experiments were performed on a single AWS EC2 instance in Ohio with 16 vCPUs and 32 GB of RAM, with 10 workers and a replication factor of 3.

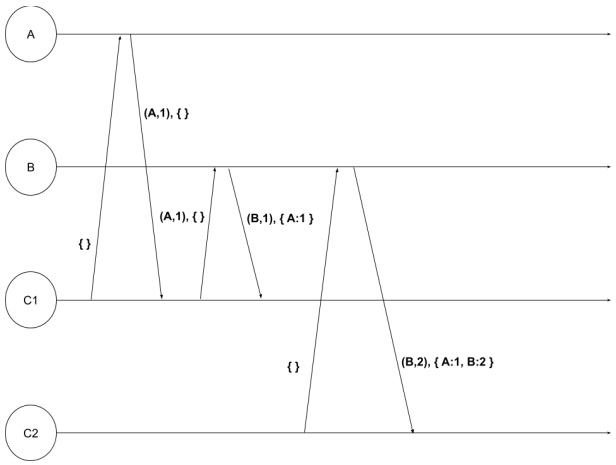

Figure 2: Context exchange between two clients and two replicas.

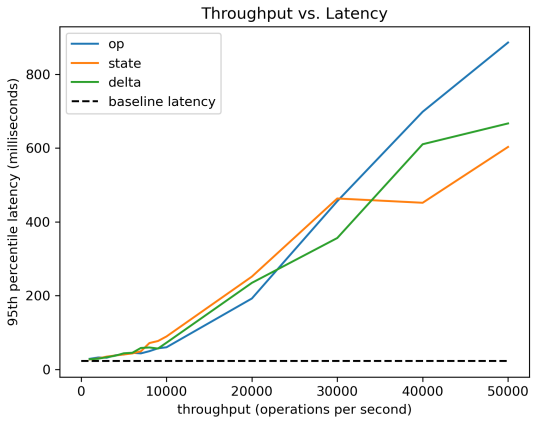

Figure 3: 95% latency vs. throughput for operation-based, state-based, and delta CRDTs for a workload consisting of 100% writes.

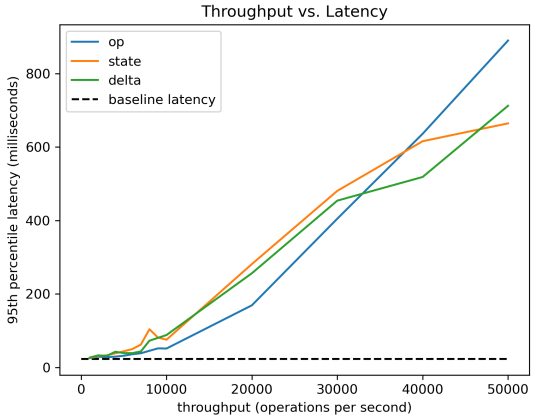

Figure 4: 95% latency vs. throughput for operation-based, state-based, and delta CRDTs for a workload consisting of 80% writes and 20% reads.

6.1 Latency vs. Throughput

First, we will test latency vs. throughput for the three different set implementations. To do this, we start up a client proxy and run a workload consisting of x add requests per second on a set of 1000 keys. Simultaneously, we spawn another proxy that measures the 95% percentile of multiple observations of operation latency. The workloads tested are as follows: a write-only workload (100% writes), a write-heavy workload (80% writes, 20% reads), and a read-only workload (0% writes, 100% reads). The results are summarized in figures 3, 4, and 5. Note that the baseline round-trip latency for a packet going from New York to Ohio is approximately 23ms, which is plotted in the figures for comparison.

For write-only workloads, state-based CRDTs yielded surprisingly good results relative to delta CRDTs. We believe that the cause of this was the broadcast epoch was too large for delta CRDTs (1 second), which resulted in relatively large state exchanges when calling *PrepareEvent*. However, this likely allowed delta and state CRDTs to have better latency, as workers tend to be less busy handling incoming events. For future work, we would like to explore mechanisms for dynamically adjusting the broadcast epoch as request load increases or decreases. We have also began using profiling tools to identify the bottlenecks in each of the three types and optimize accordingly.

Write-heavy workloads yielded similar results to the write-only workloads, particularly with operation-based CRDTs. This is likely because the difference in computational complexity for gets and puts are similar for operation-based sets. For state and delta-based CRDTs, the latency was marginally worse as throughput increases, but this is more likely due to noise in measurement.

For the read-only workloads, we noticed that all three CRDT implementations were quite similar in their latency throughout the experiment. We believe that this is the case because read-only workloads will put little pressure on the workers with regard to sending and receiving new events. Since the three implementations differ the most in the way they share updates and the payloads that they exchange, it makes sense that a read-only, or even a read-heavy workload would perform similarly across all three.

6.2 Convergence Time

A major difference between the three CRDT types is the time complexity of applying mutating operations to the state, as well as the size of the payload sent over the network. As a result, the amount of time for all replicas to converge to the same state may vary depending on the CRDT implementation, especially if the system is under high load. This section examines this empirically by performing the following experiment. We start up a client proxy and perform x operations per second for a period of time. During this, we add 10000 elements to

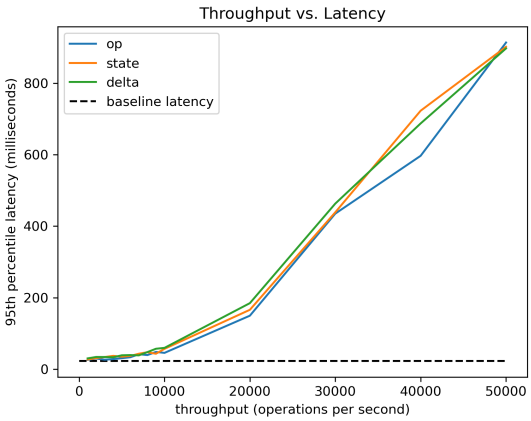

Figure 5: 95% latency vs. throughput for operation-based, state-based, and delta CRDTs for a workload consisting of 100% reads.

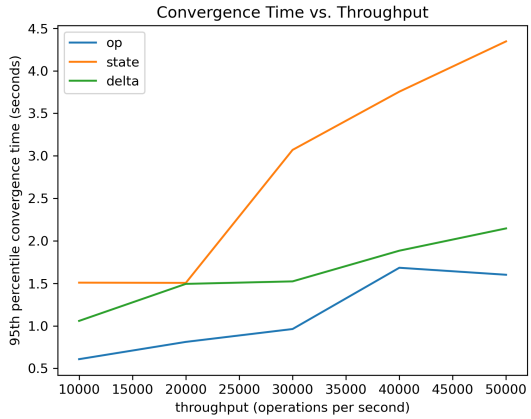

Figure 6: Time for all replicas to converge after varying numbers of set additions, for each CRDT implementation.

a set on a particular key. We then inspect the replica logs to measure how long each replica takes (after the first element is added) for its state to converge to the correct value on this key, and we take the 95th percentile convergence time of 20 replications. To make comparisons fair, we set the broadcast epoch to 100ms for each of the three experiments. The results are summarized in figure 6. State-based CRDTs begin to quickly degrade in performance when the size of the set got to about 30,000 operations. This is likely because each *PersistEvent* is merging an entire state containing an increasing number of elements each time. Perhaps more surprisingly, delta CRDTs yielded similar performance to operation-based CRDTs, with a 95% 2.14 second convergence time with 50,000 concurrent operations. This provides promising future directions in further optimizing the dissemination protocol for delta CRDTs to show their suitability for future CRDT-based key-value stores.

7 Future Work

ChiaVe is still in its early days, and there are many future development opportunities to be explored:

- **Client Proxy Features:** New features can be added to the client proxy to make the system more customizable. For example, Anna [22] employs proxy-side logic such as caching and buffering to provide varying consistency levels, such as *read uncommitted* and *item-cut isolation*.
- **Dynamic Membership Changes:** Currently, a ChiaVe cluster is manually started by loading in a configuration file, which pre-determines the IP addresses of each node, the number of workers per server, and the replication factor. Future mechanisms, such as the ones found in DynamoDB [9], will need to be implemented for dynamically adding and removing members from the cluster, as well as for reliably detecting the failure of a node.
- **Operation-Based CRDT Causal Broadcast:** The current approach for broadcasting update events is not sufficient to provide the exactly-once semantics that operation-based CRDTs require. A more robust solution will be needed by keeping persistent logs of events and implementing causal reliable broadcast.
- **Improving δ-CRDTs:** There has been a lot of recent work in making δ-CRDTs more efficient in terms of their storage and network costs. As δ-CRDTs provide a promising avenue in the future of CRDTs, implementing better synchronization mechanisms [10] or anti-entropy algorithms that perform periodic garbage collection [2] could greatly improve the performance of the system.

Conclusion

This paper presented ChiaVe, a highly flexible distributed key-value store that leverages conflict-free replicated data types to maintain high scalability and availability while still providing coordination-free strong eventual consistency. It differs from traditional key value stores in its multi-threaded actor model, which allows it to effectively scale on both multi-core machines and a larger, distributed setting. ChiaVe is unique in that it provides support for all three types of CRDTs, particularly δ-CRDTs, which haven't appeared to be adopted yet by modern key-value stores. It also uses dotted version vectors, a novel mechanism to detect causality between operations.

ChiaVe acts as an extensible platform for experimenting with these different data types and gaining empirical insights into the advantages and disadvantages of each. Early experiments on Chiave suggest that state-based and delta-based CRDTs can outperform operation-based CRDTs with respect to latency in write-only or write-heavy workloads, which is likely due to less strict demands about the dissemination of replica updates. Latency on read-only workloads was similar across all three implementations, likely due to the lack of reliance on the event exchanging system. All three implemen-

tations were able to exchange updates between replicas quite efficiently, although state-based CRDTs suffered fairly significant hits in this area as the throughput of the system went up to about 20,000 operations per second. Delta CRDTs were able to keep up with operation-based CRDTs up to about 50,000 operations per second. We hope to further optimize ChiaVe and perform further experiments to compare the performance of these three types.

ChiaVe is still a work in progress, with potential to add many new features such as client proxy customizability, dynamic membership changes, more supported data structures, dynamic broadcast epochs, and more domain-specific algorithms for CRDT propagation.

References

[1] P. Almeida, A. Shoker, and C. Baquero. Efficient State-based CRDTs by Delta-Mutation. *CoRR*, 2014.

[2] P. Almeida, A. Shoker, and C. Baquero. Delta state replicated data types. *CoRR*, abs/1603.01529, 2016.

[3] Apache Cassandra. https://cassandra.apache.org/.

[4] C. Baquero, V. Fonte, R. Gonçalves, P. Almeida, and N. Preguiça. Dotted Version Vectors: Efficient Causality Tracking for Distributed Key-Value Stores. 2012.

[5] R. Brown, Z. Lakhani, and P. Place. Big(ger) Sets: decomposed delta CRDT Sets in Riak. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*. ACM, apr 2016.

[6] Conclave: A private and secure real-time collaborative text editor. https://conclave-team.github.io/conclave-site/.

[7] CouchDB. https://couchdb.apache.org/.

[8] DocumentDB. https://aws.amazon.com/documentdb/.

[9] DynamoDB. https://aws.amazon.com/dynamodb/.

[10] V. Enes, P. Almeida, C. Baquero, and J. Leitão. Efficient synchronization of state-based crdts. *CoRR*, abs/1803.02750, 2018.

[11] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[12] Memcached. https://memcached.org/.

[13] MongoDB. https://www.mongodb.com/.

[14] D.S. Parker, G.J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, 1983.

[15] Redis. https://redis.io/.

[16] Redis Active-Active Geo-Distribution. https://redis.com/redis-enterprise/technology/active-active-geo-distribution/.

[17] Riak KV. https://riak.com/products/riak-kv/index.html.

[18] Roshi. https://github.com/soundcloud/roshi.

[19] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A Comprehensive Study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria - Centre Paris-Rocquencourt; INRIA, January 2011.

[20] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-Free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, page 386–400. Springer-Verlag, 2011.

[21] TomTom. https://www.tomtom.com/.

[22] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 401–412, 2018.

[23] Xi-Editor. https://abishov.com/xi-editor/.