RunTime-Assisted Convergence in Replicated Data Types

Gowtham Kaki
University of Colorado Boulder
Boulder, USA
gowtham.kaki@colorado.edu

Prasanth Prahladan
University of Colorado Boulder
Boulder, USA
prasanth.prahladan@colorado.edu

Nicholas V. Lewchenko
University of Colorado Boulder
Boulder, USA
nile1033@colorado.edu

Abstract

We propose a runtime-assisted approach to enforce convergence in distributed executions of replicated data types. The key distinguishing aspect of our approach is that it guarantees convergence *unconditionally* – without requiring data type operations to satisfy algebraic laws such as commutativity and idempotence. Consequently, programmers are no longer obligated to prove convergence on a per-type basis. Moreover, our approach lets sequential data types be reused in a distributed setting by extending their implementations rather than refactoring them. The novel component of our approach is a distributed runtime that orchestrates *well-formed* executions that are guaranteed to converge. Despite the utilization of a runtime, our approach comes at no additional cost of latency and availability. Instead, we introduce a novel tradeoff against a metric called *staleness*, which roughly corresponds to the time taken for replicas to converge. We implement our approach in a system called Quark and conduct a thorough evaluation of its tradeoffs.

CCS Concepts: • **Computing methodologies** → **Distributed programming languages**; • **Computer systems organization** → **Availability**; • **Software and its engineering** → *Formal software verification.*

Keywords: Replication, MRDT, CRDT, Runtime, Convergence, Concurrent Revisions, Causal Consistency, Decentralized Systems

ACM Reference Format:
Gowtham Kaki, Prasanth Prahladan, and Nicholas V. Lewchenko. 2022. RunTime-Assisted Convergence in Replicated Data Types. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3519939.3523724

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523724

1 Introduction

Large-scale web services and decentralized applications often rely on geo-distributed state replication to meet their latency and availability needs. An application's state is replicated asynchronously in such a setting, meaning that operations are independently executed at each replica, and updates are applied asynchronously at other replicas after a possible network delay. Asynchronous execution complicates programming and reasoning about distributed applications as it induces the possibility of conflicting updates leading to non-convergence and application integrity violations. Two basic approaches have been proposed to address this problem. The first approach is to selectively strengthen the system consistency to pre-empt the conflicting updates. The second is to redefine the application state in terms of *Replicated Data Types* (RDTs) that are specially engineered to handle conflicting updates. Strengthening system consistency necessarily entails inter-replica coordination, therefore applications prefer to utilize RDTs as much as possible, resorting to consistency strengthening only when it is necessary to maintain application integrity. The common design principle guiding the development of replicated data types is commutativity. The idea is that if the replicated state is only updated by commutative operations, then updates can be applied in any order and the replica states are still guaranteed to converge. Indeed, there exist common use cases in web applications that can be implemented using Commutative Replicated Data Types (CRDTs) [23]. For instance, a video view counter on a streaming application such as YouTube can be implemented as a Counter CRDT that supports commutative increments.

In general, however, commutativity is not a common occurrence among data types. Most common data type definitions come with at least a pair of operations that do not commute. For instance `add` and `remove` operations on a set do not commute if both are for the same element. Likewise two `insert` operations for the same position in a list do not commute. Such non-commutative operations translate to conflicting updates in a distributed setting, whose cumulative result cannot be determined by the sequential specification of the data type alone. To use a non-commutative data type in a replicated setting, creative re-engineering of its internal representation and algorithms is needed to turn it into a bonafide CRDT with a sensible distributed semantics. Indeed, most CRDTs have such carefully-crafted implementations that rely on non-trivial technical devices to keep track of causal dependencies and avoid or resolve conflicts. For instance, various CRDT variants of the set abstract data type rely on *vector clocks* to identify conflicting adds and removes [28, 29], Replicated Growable Array (RGA) – a CRDT for collaborative editing [20], and JSON CRDT – a CRDT variant of the JSON storage format, both use *Lamport timestamps* [15, 20]; and TreeDoc, another CRDT for collaborative editing, makes use of *dense linear orders* [19]. Such advanced implementations makes it hard to reason about basic correctness properties of CRDTs, such as convergence. Indeed, formal verification of *strong eventual consistency* (i.e., eventual convergence) for few of the aforementioned CRDTs required considerable effort on behalf of multiple authors who are experts in the field [11]. The extraordinary effort and the expertise required to build CRDTs is a deterrent towards using type-safe abstractions with strong guarantees in distributed applications. To overcome this deterrent, there is a need for an alternative approach that lets developers reuse their sequential abstractions in a distributed setting with little to no additional overhead of reasoning about their correctness properties such as convergence.

In this paper we describe an alternative approach to building convergent replicated data types that realizes the aforementioned virtues. Our approach is based on Mergeable Replicated Data Types (MRDTs) [14] – an alternative take on RDTs that is inspired by the Git version control system. MRDTs adopt a state-centric model of replication based on version-controlled *mergeable* states instead of an operation-centric model based on commutative operations. Unlike commutativity, mergeability does not require a data type definition to be refactored to suit distributed execution. Instead, the type definition only needs to be extended with a merge function to reconcile concurrent versions of the type in presence of their common ancestor version. However, mergeability itself doesn't guarantee convergence; there exist MRDTs with well-defined and intuitive merge semantics that nonetheless admit divergent executions. While constraining the data type semantics to conform to algebraic laws such as monotonicity might restore convergence, it would burden the developer with the task of enforcing such constraints, which we would like to avoid. We therefore propose an alternative approach to convergence – one that is based on runtime orchestration rather than static enforcement. In particular, we extend MRDTs with a distributed runtime that orchestrates only the *well-formed* executions which are guaranteed to converge. Notably, our approach guarantees convergence of MRDTs *regardless* of their merge semantics, thus obviating the need to constrain the type semantics or their implementations. The key insight behind our approach is the observation that a data type's merge is a complete function, and thefore can be safely invoked to merge concurrent versions of the type notwithstanding their individual operation histories. Thus, a replica's current version can always be made available for user operations with the guarantee that it can be safely merged with a concurrent version at a later point of time. The ability to arbitrarily delay merges gives us the opportunity to orchestrate them in a way that is *effectively* linear, which inturn guarantees convergence.

Note that it is easy to construct a trivial runtime that guarantees convergence of all executions. Such a runtime would make an extensive use of inter-replica coordination to synchronize the execution of every operation and thus induce a sequentially-consistent behavior. This would however defeat the purpose of state replication as the latency incurred for synchronized execution of an operation is prohibitively high, and the availability of the system during the execution is correspondingly low. It is therefore important for a distributed runtime of RDTs to *not* interfere with the execution of RDT operations. Orchestrating a well-formed convergent execution without impacting latency and availability is possible despite the the CAP theorem [9]. While CAP theorem does rule out linearizability, it does not prohibit enforcing weaker consistency constraints on distributed executions. Our approach exploits this observation by (i). Identifying a novel set of constraints on distributed executions that guarantee their convergence, and (ii). Localizing such constraints on (asynchronously executed) state merges such that per-operation latency and system-wide availability remain unaffected. The downside of interfering with merges, however, is that the replicas may now take longer to converge – a phenomenon we quantify using a metric called *staleness* (Sec. 6). Our approach thus brings to fore a fundamental tradeoff that has not been explored in the context of RDTs.

Contributions. The contributions of this paper are summarized below:

- We present a runtime-assisted approach to MRDT state replication that sufficiently constrains distributed executions so as to guarantee their convergence. This is in contrast to the existing models of replication, where each data type is obligated to prove its convergence by discharging algebraic proof obligations.
- We formalize the aforementioned constraints in the context of an abstract machine (named Quark) that manifests distributed executions as version history graphs similar to Git. We prove the *convergence* and *progress* properties of the Quark abstract machine that together guarantee its soundness. The formalization has also been mechanized in Ivy [18] and automatically verified with the help of Z3 [27]. To the best of our knowledge, this is the first time the meta-theory of a formal system is proven completely inside an SMT solver.
- We formalize Quark distributed machine that implements the semantics of the Quark abstract machine in the context of an asynchronous distributed system. The formalization addresses several practical concerns

of distribution and serves as a blueprint for an efficient implementation.

- We describe an implementation of Quark as a lightweight shim layer on top of Scylla – an off-the-shelf distributed key-value store [22]. We describe an extensive evaluation to quantify the staleness metric and novel tradeoffs in the context of case studies involving a data structure benchmark and a collaborative editing application.

While our focus in this paper is on RDTs, we believe our contributions are also meaningful in the context of the long line of work on programming models for shared-memory concurrency. In particular, this work generalizes concurrent revisions [6] beyond fork-join parallelism and server-client paradigm. Our approach demonstrates how causal coherency-based semantics can be implemented more generally in decentralized systems while guaranteeing the convergence of concurrent executions.

2 Motivation

In this section we motivate our runtime-assisted approach to convergent RDTs with help of a set data structure.

```
module Set: sig
  type t (* Abstract type of the set*)
  type elt (* Abstract type of set elements *)
  val add: elt -> unit (*Add an element *)
  val remove: elt -> unit (* Remove an element *)
end
```

Figure 1. An OCaml interface to `Set` abstract data type

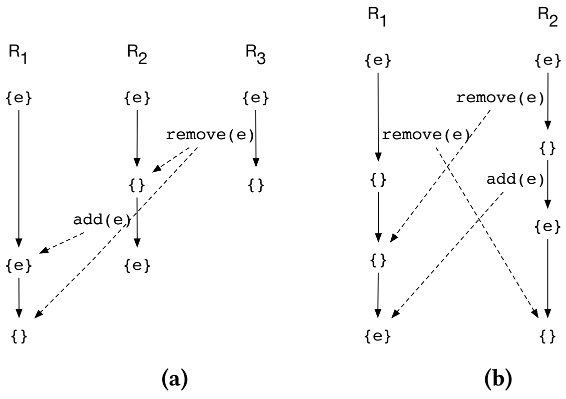


Figure 2. Anamolous executions resulting from the Asynchronous replication of `Set`. Dashed lines denote effect propagation.

From Set ADT to Set RDT. Fig. 1 shows a simplified interface of `Set` (abstract) data type in OCaml. The interface hides a reference to a set (`Set.t`), which can be updated in-place via `add` and `remove` operations. `Set` is an ordinary data type meant for sequential execution. Under a concurrent execution with an asynchronously replicated state, `Set` would exhibit anamolous behaviors such as those in Fig. 2.

Fig. 2a shows an anamolous execution with three replicas – R_1, R_2, and R_3, all of which start with a singleton set containing the element e. A client connects to the replica R_3 and executes a `remove(`e`)` operation, which is then asynchronously propagated to other replicas. Some time after applying R_3's `remove` at R_2, another client connects to R_2 and re-adds e by issuing an `add(`e`)` operation. Consequently, the state at R_2 is again the singleton set $\{e\}$. Replica R_3 however receives R_2's `add` ahead of R_1's `remove`, applies them in the same order, and ends up with an empty set. The execution thus results in divergent replica states.

Note that if `Set.add` and `Set.remove` commute, then executing them in different order at R_2 and R_3 would not have led to divergence. As such, `Set` is not a CRDT due to the admittance of non-commutative operations. Nonetheless, there exist approaches to transform `Set` into a CRDT by re-engineering its interface and operations [23, 28, 29]. For instance, the anomalous execution in Fig. 2a can be preempted by ensuring that updates are only ever applied in the causal order. This can be done by extending `Set` with vector clocks to keep track of the causal history of each operation. A set `add` (resp. `remove`) would now generate an `Add` (resp. `Remove`) *effect* tagged with the vector clock of the origin replica. Here, the vector clock simply records the sequence number of last operation from each replica whose effect has been received and applied at the current replica. When an effect is received at a replica, it is buffered until the time all its causally-preceding effects (as captured by the tagged vector clock) have already been received and applied. This strategy would preempt the execution in Fig. 2a by buffering R_2's `add` at R_1 until the causally-preceding `remove` of R_1 is received and applied. An interface for such a *causally-consistent* set RDT is shown in Fig. 3.

Add-Wins Set CRDT. Unfortunately, `Set` data type of Fig. 3 still admits divergent executions due to concurrent updates. Fig. 2b describes one such execution. Here, replicas R_1 and R_2 both start with a singleton set $\{e\}$. Two distinct clients connect to R_1 and R_2 respectively and issue two concurrent `remove(`e`)` operations. Later, another client connects to R_2 and issues an `add(`e`)` operation. The effects of these operations are asynchronously applied at remote replicas as shown in the figure, resulting in the divergent states at R_1 and R_2.

Note that, unlike the execution in Fig. 2a, the conflicting operations in Fig. 2b, namely R_1's `remove` and R_2's `add`, are *not* causally related, hence their relative order is not

```
module Set: sig
  type t       type elt
  type r_id (* Replica Id *)
  (* Vector Clock *)
  type v_clock = (r_id,int) HashTable.t
  (* Set RDT Effects *)
  type eff = Add of elt * v_clock
           | Remove of elt * v_clock
  (* This replica's vector clock *)
  val local_clock: v_clock
  val add: elt -> eff
  val remove: elt -> eff
  val buf: eff list (* pending effects *)
  (* Apply an effect to the local state *)
  val apply_eff: eff -> unit
end
```

Figure 3. An OCaml interface to Set replicated data type

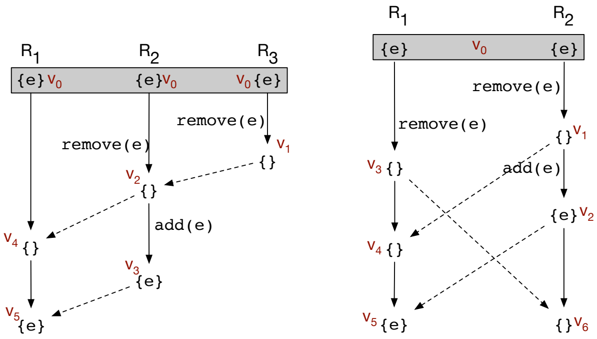
determined by the sequential specification of the data type. Forcing a causal relationship between them requires synchronization between adds and removes, which is expensive in an asynchronous distributed setting. It therefore becomes inevitable to ascribe semantics to concurrent executions to restore convergence. This is done by imposing an *arbitration order* among concurrent conflicting operations, which are otherwise unordered. For example, in Fig. 2b, we might let R_2's add override R_1's remove considering that add is re-adding an element after a previous remove. Ordering concurrent removes ahead of adds uniformly on all replicas leads to an implementation of Set RDT where concurrent (re-)adds consistently *win* over removes. Such an *add-wins* set is useful, for instance, to implement a shared shopping cart where two users can concurrently remove an item from the cart, but if one of them re-adds it then the item should be present in the final cart[1].

Extending the Set implementation from Fig. 3 with *add-wins* semantics is however not trivial. The suggested approach involves tracking element-wise causal dependencies between the removes and adds by maintaining a vector clock *for each element e* in the set [28]. The vector clock of *e* records the sequence number of the last add(e) operation from each replica whose effect has been received and applied at the current replica. The Add and Remove effects on *e* will now be tagged with *e*'s vector clock. When a Remove effect on *e* is received at a replica, it is applied only if the tagged vector clock is no less than *e*'s local vector clock, i.e., only if the arriving Remove has seen (i.e., causally succeeds) at least those add(e) operations the current replica is aware of. Otherwise the effect is simply a no-op. This strategy would result in convergent states despite the execution in Fig. 2b as R_2's remove effectively becomes a no-op at R_1 due to there being at least one add operation on *e* that it hasn't seen. The *add-wins* Set interface is similar to Fig. 3, except that it requires an additional data structure to track element-wise vector clocks:

```
  val e_clock: (elt, v_clock) HashTable.t
```

The resultant set RDT is assumed to be correct albeit a formal proof of convergence could not be found in the literature.

The problem with the CRDT approach. The above exercise demonstrates the considerable ingenuity and effort involved in deriving a convergent RDT out of such simple data type as Set. Such sophistication makes it quite hard for non-expert developers to build, or even customize existing replicated data types to suit the needs of their application. Some of the effort can be mitigated by strengthening the underlying system model insofar as it doesn't affect the latency and availability of the application. For instance, a system that always delivers messages in the causal order would automatically preempt the execution in Fig. 2a without the need for additional intervention on behalf of the developer. This is a particularly attractive proposition considering that causal consistency can be "bolted on" an existing implementation of an eventually consistent system without weakening its guarantees [3]. Unfortunately, such strengthening of the system model would deliver no benefits to the developer if they still have to reason about fine-grained causal dependencies to guarantee convergence, such as in the case of Fig. 2b. As we observed earlier, the execution in Fig. 2b seems inevitable unless every pair of add and remove operations are synchronized, which, regrettably, is not a practical option. The developer therefore seems to be stuck.



(a) **(b)**

Figure 4. Equivalent executions of Fig. 2 in state-centric replication model. Dashed lines now denote state merges.

The MRDT Approach. Mergeable Replicated Data Types (MRDTs) implement a state-centric model of replication that is inspired by the Git version control system. Like in Git, the state evolves in terms of versions, and concurrent versions of the state can be merged. The semantics of merge depends on

[1] This is in fact the semantics of Amazon's shopping cart [8]

the type of the state, so each MRDT is required to be equipped with a three-way `merge` function that merges concurrent versions of that type in presence of their (lowest) common ancestor version. In our running example, the state is a value of type `Set.t`, hence `Set.merge` function would have the type signature:

$$\texttt{Set.merge} : \texttt{Set.t} \rightarrow \texttt{Set.t} \rightarrow \texttt{Set.t} \rightarrow \texttt{Set.t}$$

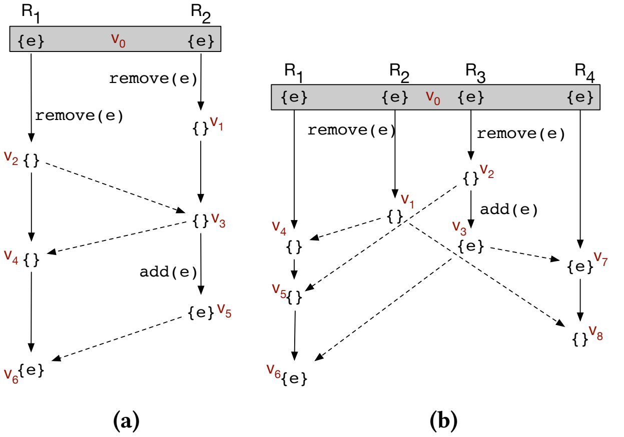
The three arguments of `merge` correspond to the lowest common ancestor (LCA) version and the two concurrent versions that independently evolved from the LCA version. The LCA is a causal ancestor of concurrent versions, hence causal consistency is built into the replication model. The result of set merge, intuitively, must contain the common elements in the two concurrent versions along with any newly added elements in either versions. Concretely:

```
let merge s(*lca*) s1 s2 =
          (s1 ∩ s2) ∪ (s1 - s) ∪ (s2 - s)
```

Extending `Set` ADT with the above merge function results in a `Set` MRDT.

Let us now reconsider the executions in Fig. 2, this time on `Set` MRDT. The equivalent executions are shown in Fig. 4. Due to causal consistency being built into the model, the divergent execution of Fig. 2a is preempted in favor of the convergent execution shown in Fig. 4a. The execution manifests as follows. The initial version (v_0) on all three replicas is the singleton set $\{e\}$. Applying operations to replicas creates new versions, e.g., v_1 on R_3, and v_2 and v_3 on R_2. Changes can be propagated by merging versions, e.g., version v_2 on R_2 is a result of merging v_1 into v_0 for which v_0 serves as the lowest common ancestor (LCA) [2]. Likewise v_4 on R_1 is created by merging v_2 into v_0 in presence of their LCA v_0. The next version v_5 on R_1 is the result of merging R_2's v_3 into R_1's v_4. The LCA for this merge is v_2. By the end of the execution, versions v_5 and v_3 on R_1 and R_2 (resp.) have witnessed the same set of operations, hence are in agreement.

Well-formed executions and the* Quark *runtime. Convergence however is not an inherent virtue of the state-centric replication model. Fig. 4b shows the state-centric analogue of the execution in Fig. 2b which diverges. Here R_1 and R_2 start with version $v_0 = \{e\}$. R_2 performs a `remove` and an `add` making versions v_1 and v_2 respectively. Simultaneously, R_1 performs a `remove` to make v_3. Replica R_1 now obtains R_2's changes by merging versions v_1 and v_2 to make new versions v_4 and v_5 respectively. LCAs for these merges are v_0 and v_1. Concurrently, R_2 obtains R_1's changes by merging v_3 with v_2 (LCA = v_0) to make v_6. Now R_1 and R_2 have same set of changes yet their final versions differ.



(a) **(b)**

Figure 5. Executions demonstrating the difference between linearized (former) and concurrent (latter) merges.

Note that the anamolous execution in Fig. 4b could have been avoided had the merges between R_1 and R_2 been linearized. Fig. 5a shows an execution that only slightly differs from the one in Fig. 4b. The difference is that the merges in Fig. 5a happen linearly: first R_1 is merged into R_2 (bringing R_1's `remove`), then R_2 into R_1 (bringing R_2's `remove`), followed again by R_2 into R_1 (bringing R_2's `add`). As a result of such linearization, the final versions on R_1 and R_2 converge to the singleton set $\{e\}$. Note that there exist other linearizations of merges; for instance, $R_1 \rightarrow R_2$ merge could be ordered between the two $R_2 \rightarrow R_1$ merges. However, all linearizations result in the same final state $\{e\}$. Another important point to note is that only the merges are linearized; not the entire execution. In Fig. 5a, both R_1 and R_2's `removes` remove the same element e, which is not possible in a linearized execution.

In the context of Fig. 5a, it is quite clear what linearization of merges means and how to enforce it via synchronization (for e.g., wrapping each merge within a global lock). In general however, the semantics of merge linearization isn't as cut and dried. For instance, consider the execution in Fig. 5b. The four replicas involved in the execution start with version $v_0 = \{e\}$. The replicas perform local operations as shown in the figure to make versions v_1 to v_3. Next they perform a series of merges to propagate local changes. The merges can be ordered in time as following: first $R_2 \rightarrow R_1$ (merging v_1), then $R_3 \rightarrow R_1$ twice (merging v_2 and v_3 resp.), then $R_3 \rightarrow R_4$ (merging v_3), and finally $R_2 \rightarrow R_4$ (merging v_1). These merges collectively propagate the effects of `add` and `remove` operations to R_1 and R_4. And despite being ordered in time, they nonetheless result in a divergent execution ($v_6 \neq v_8$). The problem here is that, although merges are executed linearly, the execution graph does not reflect this linearity; merges that end in R_1 in Fig. 5b are effectively concurrent with those that end in R_4. This shows that simply synchronizing the execution of merges does *not* result in convergence.

[2] Versions v_0 and v_1 are not concurrent as the former is an ancestor of the latter. Merging v_1 into v_0 is nonetheless possible as v_1 is ahead of v_0 in causal order. In Git parlance this is a *fast forward* merge.

Our key insight to overcome this impasse is a novel *well-formedness* condition on execution graphs that ensures convergence of final states. To understand well-formedness, let us contrast the bad executions in Figs. 4b and 5b against the good execution in Fig. 5a. Observe that in Fig. 5a, *every* pair of concurrent versions on R_1 and R_2 have a unique lowest common ancestor (LCA). For instance, LCA of (v_5, v_6) is v_5, (v_5, v_4) is v_3, (v_1, v_2) is v_0 and so on. By contrast in Fig. 4b, versions v_5 and v_6 have two LCAs, namely v_2 and v_3. Both these versions are common ancestors of v_5 and v_6, and both are *lowest* in the sense that there do not exist versions lower (in the execution graph) than v_2 and v_3 that are also common ancestors of v_5 and v_6. Likewise in Fig. 5b, versions v_6 and v_8 have two LCAs – v_1 and v_3. Multiple LCAs is an indication that there exist merges prior in the execution graph that are effectively concurrent. Considering that our approach to convergence in the state-centric model crucially relies on the linearity of merges, presence of multiple LCAs opens up the possibility of divergence. We therefore define well-formed execution graphs as those where LCAs for every pair of concurrent versions is unique. As we prove in Sec. 3, enforcing this structural well-formedness condition indeed guarantees the convergence of distributed executions. In Sec. 4, we formalize a distributed runtime called Quark whose primary purpose is to enforce the well-formedness condition on distributed executions. In Sec. 5, we describe its implementation. The runtime makes it possible to automatically enforce convergence, letting us promote ordinary sequential data types to convergent replicated types by simply equipping them with a merge function. Thus, extending the `Set` interface of Fig. 1 with the two-line `Set.merge` function shown above is sufficient to obtain a convergent replicated set data type.

3 Semantics of State-Centric Replication

In this section we present the formal semantics of our state-centric replication scheme with linearized merges. As the informal development from previous section suggests, our replication scheme is strongly inspired by version control systems (VCS) such as Git. We embrace this analogy in our formal development to manifest distributed executions over replicated state with help of an abstract "Git" machine building a well-formed version history graph. We show that the version history graph thus generated has several desirable properties including unique LCAs for every pair of versions, and convergence of versions that include the same set of *commits*. We are however not concerned about the practical aspects of the system yet; the subsequent sections gradually refine the abstract semantics described here into a practical distributed system we call Quark. For convenience, we refer to the abstract "Git" machine we describe here also as Quark.

Fig. 6 shows the operational semantics of the Quark abstract machine. The machine admits the usual version control operations, namely Commit, Fork, Merge, and FastFwd. Operation FastFwd is a special case of merge which simply *fast forwards* a branch to a later version. A *branch* is a linear sequence of versions, which intuitively denotes the progressive evolution of the state on a replica. There is thus a one-to-one mapping between replicas and branches. The latest version of the branch, called its *head*, denotes the current state of the corresponding replica. A new head version is created either by an externally-initiated *commit* (Commit) or by merging the current head with a concurrent or causally-succeeding version from a different branch (Merge and FastFwd respectively). A new branch can be created by *forking off* a version on an existing branch (Fork). Although a version control system admits more operations (e.g., "rebase"), we observe that these four basic actions are sufficient to capture the behavior of an asynchronously replicated multi-versioned state machine.

The state of Quark abstract machine is a tuple $\Delta = (G, N, C, H, L)$, where:

- $G = (V, E)$ is the version history DAG generated by the execution of the abstract machine. Vertices (V) of the graph are the set of all versions that ever existed during the execution. Edges (E) record the relationships between various versions. In particular, there are three kinds of edges corresponding to the three basic operations: commit $(\xrightarrow{c})$, merge $(\xrightarrow{m})$, fork $(\xrightarrow{f})$. Fast forward, being a special case of merge, is also denoted by a merge edge $(\xrightarrow{m})$. Existence of an edge $v_0 \rightarrow v_1$ denotes that versions v_0 and v_1 are related by some operation. For e.g., $v_0 \xrightarrow{f} v_1$ denotes that v_1 is a new version forked-off from the version v_0 using the Fork rule. As usual, *path* relation is the reflexive transitive closure of the edge relation, i.e, $\rightarrow^*$.
- $N : \texttt{Version} \rightarrow \texttt{Value}$ is a (partial) function (i.e., a map) that maps versions to their values. We distinguish versions from values as different versions on different branches may store the same value (e.g., the same string "hello"), yet need to be uniquely identified. The domain of values is left uninterpreted except for the requirement that it be *mergeable*, i.e., define a three-way `merge` function.
- $C : \texttt{Version} \rightarrow \mathcal{P}\,(\texttt{CommitId})$ maps a version v to the set of commit ids included in that version. Each commit event during the execution is uniquely identified by a commit id (analogous to the Git's commit hash). The set of commit ids $C(v)$ therefore denotes the commit events that contributed to the version v. Intuitively, this represents the set of user-initiated operations that affected the value of this version.

$$v \in \texttt{Versions/Vertices} \quad b \in \texttt{Branches} \quad c \in \texttt{CommitIds} \quad n \in \texttt{Values} \quad \xrightarrow{c}, \xrightarrow{f}, \xrightarrow{m} \in \texttt{Edges} \quad G \in (\texttt{Vertices}, \texttt{Edges})$$

$$N : \texttt{Version} \rightarrow \texttt{Value} \quad C : \texttt{Version} \rightarrow \mathcal{P}\,(\texttt{CommitId}) \quad H : \texttt{Branch} \rightarrow \texttt{Version} \quad L : \texttt{Branch} \times \texttt{Branch} \rightarrow \texttt{Version}$$

$$\boxed{(G, N, C, H, L) \longrightarrow (G', N', C', H', L')}$$

$$\frac{b \in \mathit{dom}(H) \quad v \notin V \quad i \notin \mathit{codom}(C)}{((V,E), N, C, H, L) \longrightarrow (V \cup \{v\},\ E \cup \{H(b) \xrightarrow{c} v\},\ N[v \mapsto n],\ C[v \mapsto \{i\} \cup C(H(b))],\ H[b \mapsto v], L)} \ [\text{Commit}]$$

$$\frac{b \in \mathit{dom}(H) \quad b' \notin \mathit{dom}(H) \quad v \notin V}{\begin{array}{r} ((V,E), N, C, H, L) \longrightarrow (V \cup \{v\},\ E \cup \{H(b) \xrightarrow{f} v\},\ N[v \mapsto N(H(b))],\ C[v \mapsto C(H(b))], \\ H[b' \mapsto v],\ L[(b', b) \mapsto H(b)][\{(b', b'') \mapsto L(b, b'') \mid b'' \neq b\}]) \end{array}} \ [\text{Fork}]$$

$$\frac{\begin{array}{c} b, b' \in \mathit{dom}(H) \quad C(H(b)) \supset C(L(b,b')) \quad C(H(b')) \supset C(L(b,b')) \\ \forall (b'' \in \mathit{dom}(H)).\ L(b,b'') \rightarrow^* L(b',b'') \ \vee\ L(b',b'') \rightarrow^* L(b,b'') \\ n = \mathsf{merge}(N(L(b,b')),\ N(H(b)),\ N(H(b'))) \quad v \notin V \end{array}}{\begin{array}{l} ((V,E), N, C, H, L) \longrightarrow (V \cup \{v\},\ E \cup \{H(b) \xrightarrow{m} v, H(b') \xrightarrow{m} v\}, \\ \qquad\qquad N[v \mapsto n],\ C[v \mapsto C(H(b)) \cup C(H(b'))],\ H[b \mapsto v], \\ \qquad\qquad L[(b,b') \mapsto H(b')][\{(b,b'') \mapsto L(b',b'') \mid L(b,b'') \rightarrow^* L(b',b'')\}]) \end{array}} \ [\text{Merge}]$$

$$\frac{\begin{array}{c} b, b' \in \mathit{dom}(H) \quad C(H(b)) = C(L(b,b')) \quad C(H(b')) \supset C(L(b,b')) \\ \forall (b'' \in \mathit{dom}(H)).\ L(b,b'') \rightarrow^* L(b',b'') \ \vee\ L(b',b'') \rightarrow^* L(b,b'') \quad v \notin V \end{array}}{\begin{array}{l} ((V,E), N, C, H, L) \longrightarrow (V \cup \{v\},\ E \cup \{H(b) \xrightarrow{m} v, H(b') \xrightarrow{m} v\}, \\ \qquad\qquad N[v \mapsto N(H(b'))],\ C[v \mapsto C(H(b'))],\ H[b \mapsto v], \\ \qquad\qquad L[(b,b') \mapsto H(b')][\{(b,b'') \mapsto L(b',b'') \mid L(b,b'') \rightarrow^* L(b',b'')\}]) \end{array}} \ [\text{FastFwd}]$$

Figure 6. The semantics of Quark abstract machine inspired by the Git version control system

- $H : \texttt{Branch} \rightarrow \texttt{Version}$ is an injective (partial) function that maps each branch to its head version.
- $L : \texttt{Branch} \times \texttt{Branch} \rightarrow \texttt{Version}$ maps a pair of branches to the lowest common ancestor (LCA) version of their heads. We later prove the unique LCA property, so L is indeed a (partial) function. Note that LCA of a pair of branches b_1 and b_2 need not necessarily lie on b_1 or b_2; it could also be a version on a different branch b. This happens, for e.g., if b_1 and b_2 alternatively merged the same version from b.

Notation. We let $v_\odot$ denote the initial version ("root"), and $b_\odot$ denote the initial branch ("master"). The execution of the abstract machine progresses by adding to the sets V and E, and updating the maps N, C, H, and L. We adopt the usual update notation, e.g., $H[b \mapsto v]$ is a map H' such that $H'(b) = v$, and for all $b' \neq b$, $H'(b') = H(b')$. Multiple updates to a map are parsed left-associatively. Map L is assumed to be commutative, so $L[(b,b') \mapsto v]$ is equal to $L[(b,b') \mapsto v][(b',b) \mapsto v]$; the former is used as a succinct replacement of the latter. To update multiple bindings in L, we use the set comprehension notation: given a branch b, $L[\{(b,b') \mapsto v \mid \phi(b,b')\}]$ updates all bindings (b,b') in L to v, where b' is any branch such that $\phi(b,b')$ is true. Domain and co-domain of a map M is denoted $\mathit{dom}(M)$ and $\mathit{codom}(M)$ respectively.

Definition 3.1 (Initial State and Version History Graph)**.** The graph $G_\odot = (\{v_\odot\}, \emptyset)$ is the initial version graph. The state $\Delta_\odot = (G_\odot,\ [v_\odot \mapsto b_\odot],\ [b_\odot \mapsto v_\odot],\ \emptyset)$ is the initial state.

All executions of the abstract machine are assumed to start from the initial state.

Definition 3.2 (Ancestor)**.** In version history DAG $G = (V,E)$, version $v_0 \in V$ is said to be a (causal) ancestor of $v_1 \in V$ iff $v_0 \rightarrow^* v_1$. Versions $v_0, v_1 \in V$ are causally related iff either $v_0 \rightarrow^* v_1$ or $v_1 \rightarrow^* v_0$.

Definition 3.3 (Lowest Common Ancestor (LCA))**.** In version history DAG $G = (V,E)$, a version $v \in$ is a lowest common ancestor of versions $v_1 \in V$ and $v_2 \in V$ iff:

- v is a common ancestor of v_1 and v_2, i.e., $v \rightarrow^* v_1$ and $v \rightarrow^* v_2$, and
- There does not exist a $v' \in V$ such that v' is a common ancestor of v_1 and v_2, and $v \rightarrow^* v'$.

LCA of a pair of branches is defined as the LCA of their heads.

Rules. The rule Commit (Fig. 6) describes committing a new version v onto the branch b updating its head. The value n for the new version is assumed to have been provided by whoever has invoked the commit, e.g., the user. Intuitively, user invokes an RDT operation (e.g., $\mathtt{add}(e)$) on the value of the current version to create the new value n, and thus the new version v. A unique commit id i is assigned to this commit event and added to $C(v)$. The set $C(v)$ also contains all the commit ids from the previous version $H(b)$ since the new value n is assumed to have been derived from the previous value $N(H(b))$. The edge $H(b) \xrightarrow{c} v$ records this dependency and also documents the progression of branch b. The LCA map L does not change as the only new edge is between the versions of the same branch b.

Fork describes the semantics of forking a new branch b' from the head of an existing branch b. The head of b' is a new version v that shares the same commit set ($C(v)$) and value ($N(v)$) as its predecessor ($H(b)$). The lowest common ancestor (LCA) of b' and its parent b is clearly the head of the parent $H(b)$ as there does not exist a version v lower than $H(b)$ that is an ancestor of both $H(b)$ and $H(b')$. For every other branch b'', $L(b', b'')$ is same as $L(b, b'')$. Fork operation could model, for e.g., creating a new replica by forking off the current state of an existing replica.

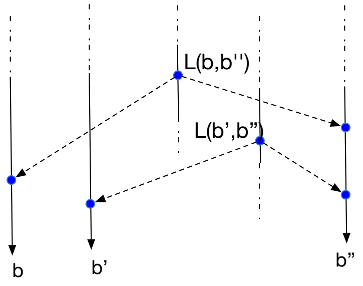


Figure 7. Explanation for the premise $L(b, b'') \longrightarrow^* L(b', b'') \lor L(b', b'') \longrightarrow^* L(b, b'')$ on the Merge rule.

Merge describes the semantics of merging the head of a branch b' into b resulting in an new version v on b. Intuitively, Merge models the information exchange between replicas. The two pre-conditions specified using the strict superset relation ($\supset$) require each of the merging versions, $H(b)$ and $H(b')$, to include at least one commit not present in their common ancestor $L(b, b')$. These conditions ensure that the merge is not trivial (trivial merge is handled by FastFwd). The next pre-condition is key to ensuring the uniqueness of LCAs and the linearity of merges. It requires that, for every branch b'' in the system, the LCA of b'' with merging branches b and b' be causally related, i.e, $L(b, b'') \longrightarrow^* L(b', b'') \lor L(b', b'') \longrightarrow^* L(b, b'')$. Fig. 7 helps visualize this condition in the most general case when (i). Branches b, b', and b'' are distinct, and (ii). Their LCAs $L(b, b'')$ and $L(b', b'')$ lie on a distinct pair of branches not equal to b, b', and b''. In the figure, once you merge b' into b, every version v that is an ancestor of $L(b', b'')$, i.e., $v \longrightarrow^* L(b', b'')$, will be a common ancestor of $H(b)$ and $H(b'')$. Clearly, $L(b', b'')$ is the lowest among such common ancestors. But the current lowest common ancestor of b and b'' is $L(b, b'')$. We therefore end up with two lowest common ancestors – $L(b', b'')$ and $L(b, b'')$, *unless* both are ancestrally related. Thus the precondition $L(b, b'') \longrightarrow^* L(b', b'') \lor L(b', b'') \longrightarrow^* L(b, b'')$. If $L(b', b'') \longrightarrow^* L(b, b'')$, then $L(b, b'')$, the current LCA of b and b'', is still the LCA after the merge. On the other hand, if $L(b, b'') \longrightarrow^* L(b', b'')$, then $L(b', b'')$ becomes the lowest common ancestor of b and b'' after the merge. Thus $L(b, b'')$ needs to be updated if and only if $L(b, b'') \longrightarrow^* L(b', b'')$. The conclusion of the Merge captures this update using the set comprehension notation. It also updates the LCA of the merging branches $L(b, b')$ to $H(b')$ since the head of b' is merged into b. Updates to the other components of the state (e.g., C) follow the same rationale as previous rules. Two edges are added to E as the new version v is a descendant of the two merging versions.

Another notable aspect of the Merge rule is the invocation of the `merge` function on the values of the merging versions and their common ancestor to derive the result of the merge. As explained above, our development is parameterized on the domain of values for which a `merge` function is defined. No further constraints are imposed on `merge`. We use an uncurried version of `merge` to avoid clutter.

FastFwd generalizes merge to the case when the merging version $H(b')$ is a descendant of the version $H(b)$. In this case $H'(b)$, the new head of b, needs to have the same value and same set of commits as $H(b')$. Other premises and conclusions are similar to the Merge rule. Note that we don't need a separate rule for fast forward merge if `merge` satisfies the invariant that $\forall n, n'. \mathtt{merge}(n, n, n') = \mathtt{merge}(n, n', n) = n$. Having a separate rule lets us elide this constraint.

Properties. We now formalize the notable properties of the abstract machine and its executions[3].

Lemma 3.4 (Uniqueness of LCA)**.** *In every reachable state $\Delta = (G, N, C, H, L)$ of the abstract machine, every pair of branches $b_1, b_2 \in dom(H)$ has a unique LCA given by $L(b_1, b_2)$.*

The intuition behind the proof is succinctly captured by Fig. 7, which is explained above.

Lemma 3.5 (Commit sets grow monotonically)**.** *In every reachable state $\Delta = ((V, E), N, C, H, L)$ of the abstract machine: For all $v_1, v_2 \in V$, if $v_1 \longrightarrow^* v_2$ then $C(v_1) \subseteq C(v_2)$.*

Lemma 3.5 guarantees that merges never lose a commit.

Corollary 3.6 (Commit sets modulo LCA are disjoint)**.** *In every reachable state $\Delta = ((V, E), N, C, H, L)$ of the abstract machine: For all distinct $b_1, b_2 \in dom(H)$, and $v_0, v_1, v_2 \in V$ s.t. $v_1 = H(b_1)$ and $v_2 = H(b_2)$ and $v_0 = L(b_1, b_2)$, the following is true: $(C(v_2) - C(v_0)) \cap (C(v_1) - C(v_0)) = \emptyset$.*

Corollary 3.6 follows from Lemmas 3.4 and 3.5.

[3] Proofs can be found in the Appendix

Theorem 3.7 (Convergence). *In every reachable state* $\Delta = ((V, E), N, C, H, L)$ *of the abstract machine: For all distinct* $b_1, b_2 \in dom(H)$*, and* $v_1, v_2 \in V$ *such that* $v_1 = H(b_1)$ *and* $v_2 = H(b_2)$*, the following is true:* $C(v_1) = C(v_2) \Rightarrow N(v_1) = N(v_2)$.

Theorem 3.7 is the key result of this section. It asserts that any two branches that witnessed the same set of commits have the same value. Intuitively, this means that any two replicas that witnessed the same set of user actions arrive at the same final state *regardless* of the order in which they are witnessed.

Convergence is vacuously true if the abstract machine never lets any merges to happen, i.e., if the premises of the Merge rule are too strong to be never true. We prove that this is not the case with help of the following theorem:

Theorem 3.8 (Progress). *Every reachable state* $\Delta = ((V, E), N, C, H, L)$ *of the abstract machine is either:*

- *A "quiescent" state, where:* $\forall b_1, b_2 \in dom(H).\ C(H(b_1)) = C(H(b_2))$*, Or*
- *An "unstuck" state, where there exist* $b_1, b_2 \in dom(H)$ *that satisfy the pre-conditions of* Merge *or* FastFwd *rules, i.e.,* b_1 *and* b_2 *are mergeable.*

Convergence and Progress together ensure the soundness of the Quark abstract machine.

Ivy formalization. In addition to the manual proofs, we also formalized the Quark abstract machine in Ivy [18] and verified its properties automatically with help of Z3 [27]. Ivy is a multi-modal verification tool, whose primary purpose is to verify the design and implementations of distributed protocols. The key differentiating aspect of Ivy is its onus on predictable automation and decidable reasoning. Automated proofs in Ivy are restricted to logical fragments for which the tool is a decision procedure. Consequently, the creative aspect of proving a theorem in Ivy lies in discovering decidable abstractions that characterize the system being reasoned about, and crafting the inductive invariants required to discharge the automated proof. This is in contrast to theorem provers such as Coq and Isabelle, and their extensions such as Sledgehammer [4], where automation is used on a best-effort basis to aid interactive theorem proving.

Our formalization of the Quark abstract machine in Ivy follows the formal development in Fig. 6 but uses slightly different abstractions to obtain decidable reasoning. Maps C, N, H, and L are modeled as relations instead of functions since reasoning about quantifier-bound function application is in general undecidable. V and E are relations by definition. Their initial values are as specified by $\Delta_{\odot}$ in Def. 3.1. The system admits four *actions* corresponding to the four transition rules in Fig. 6. Each action "updates" the system state $\Delta = ((V, E), N, C, H, L)$ as defined by the corresponding rule in Fig. 6. Commit, version, and branch identifiers are modeled as totally ordered sets of uninterpreted values from which new values can be chosen sequentially. This lets us capture these identifiers as monotonically increasing integers while avoiding undecidable reasoning with quantifier-bound integer arithmetic. Lemmas/Theorems 3.4 to 3.8 are asserted as inductive properties of the Quark abstract machine. Ivy succesfully verifies each of these properties with help of Z3, thus certifying the soundness of Quark's meta-theory.

4 Concrete Semantics

The Quark abstract machine of the previous section deliberately ignores system-level concerns to focus on the semantics of version-controlled state replication. In this section we present the Quark *distributed* machine that addresses the key system-level concerns and provides a blueprint for a practical version-controlled replicated state machine. We first reify the one-to-one correspondence between replicas and branches we assumed informally in the previous section. This requires us to relax the assumption of the state Δ being shared synchronously across all replicas. Next we present an efficient method to track the version history with the help of vector clocks. Finally, we outline an algorithm for garbage collecting older versions so that the version history need not grow unboundedly.

Fig. 8 shows the operational semantics of the distributed machine. The key difference from the abstract semantics of Fig. 6 is the presence of `ReplicaId` indexing the components of the system state. We thus admit the possibility of different replicas having different conceptions of the state. Another major difference is the use of version vectors [16] as identifiers and placeholders for versions. A version vector of a version v records the sequence number of the last version from each branch that causally precedes v (as per Def. 3.2). Concretely, version vector is a map from branches to natural numbers:

$$\texttt{VersionVector} = \texttt{Branch} \longrightarrow \mathbb{N}$$

The state of the distributed machine is the triple $\delta = (B, N, H)$. Components B and H are indexed by `ReplicaId` to let us denote their replica-local copies. For instance, replica i's copy of the *head map* H is given by $(H\ i)$, which we abbreviate to H_i for notational convenience. Like H in Fig. 6, H_i maps each branch to the version vector of its head. Note that two replicas may be out of sync w.r.t the information in H and B. For instance, $H_i(b)$ may not be equal to $H_j(b)$ if replicas i and j are out of sync. On a replica i, *branch map* B_i gives the version vector corresponding to a particular sequence number on a branch. For instance, the version vector of the first version on branch b is given by $B_i(b, 1)$ on replica i. The value map N maps version vectors to their values. We assume a single copy of N to simplify the presentation. Generalizing Fig. 8 to allow for replica-local copies of N is straightforward.

Conspicuous by its absence in Fig. 8 is the LCA map L, which we previously used to track the LCA version for every

$$i, j \in \texttt{ReplicaIds} \quad b_i, b_j \in \texttt{Branches} \quad t \in \texttt{VersionVectors} \quad n \in \texttt{Values} \quad N : \texttt{VersionVector} \rightarrow \texttt{Value}$$
$$H : \texttt{ReplicaId} \rightarrow \texttt{Branch} \rightarrow \texttt{VersionVector} \quad B : \texttt{ReplicaId} \rightarrow (\texttt{Branch} \times \mathbb{N}) \rightarrow \texttt{VersionVector}$$

$$\boxed{(B, N, H) \twoheadrightarrow (B', N', H')}$$

$$\frac{b_i \in dom(H_i) \quad t = H_i(b_i) \quad t' = t[b_i \mapsto t(b_i) + 1]}{(B, N, H) \twoheadrightarrow (B_i[(b_i, t'(b_i)) \mapsto t'],\ N[t' \mapsto n],\ H_i[b_i \mapsto t'])} \quad [\textsc{Commit}]$$

$$\frac{\begin{array}{c} b_i, b_j \in dom(H_i) \quad \forall(k \in dom(H)).\ b_k \in dom(H_i) \ \wedge \ H_i(b_k) = H_k(b_k) \\ t_i = H_i(b_i) \quad t_j = H_i(b_j) \quad t_l = t_i \sqcap t_j \quad t_l < t_i \quad t_l < t_j \quad t' = t_j \sqcup t_i[b_i \mapsto t_i(b_i) + 1] \\ \forall(k \in dom(H)).\ (t_i \sqcap H_i(b_k)) \lesseqgtr (t_j \sqcap H_i(b_k)) \quad n = \mathsf{merge}(N(t_l),\ N(H_i(b_i)),\ N(H_i(b_j))) \end{array}}{(B, N, H) \twoheadrightarrow (B_i[(b_i, t'(b_i)) \mapsto t'], N[t' \mapsto n],\ H_i[b_i \mapsto t'])} \quad [\textsc{Merge}]$$

$$\frac{b_k \in dom(H_i) \quad b_k \in dom(H_j) \quad t = H_i(b_k) \quad t' = H_j(b_k) \quad t' > t}{(B, N, H) \twoheadrightarrow (B_i[(b_k, t(b_k) + n) \mapsto B_j(b_k, t(b_k) + n) \mid n \in \{1, \ldots, t'(b_k) - t(b_k)\}], N,\ H_i[b_k \mapsto t'])} \quad [\textsc{Sync}]$$

Figure 8. The semantics of Quark distributed machine

pair of branches. The use of version vectors, coupled with the unique LCA guarantee, obviates the need for an LCA map. We can instead identify the LCA of a pair of versions v_1 and v_2 by computing the *greatest lower bound* (GLB) of their version vectors t_1 and t_2. Concretely:

$$t_l \ = \ t_1 \sqcap t_2$$

Where t_l is the version vector of the LCA and $\sqcap$ is the GLB operator. Conversely, the version vector identifying the result of a merge can be computed as the *least upper bound* (LUB) of the two version vectors involved in the merge. Concretely:

$$t_m \ = \ t_1 \sqcup t_2$$

Where t_1 and t_2, and t_m are the version vectors of the merging versions and the result of the merge, respectively. For technical reasons, however, we increment the component of t_m corresponding to the current branch b to signify that this is a new version on b. So the actual version vector of the merge is $t'_m = t_m[b \mapsto t_m(b) + 1]$. The GLB and LUB operations on version vectors are standard [16] – GLB is computed by taking the component-wise minimum, and LUB by taking their maximum. The comparison of version vectors is also standard – $t_1 < t_2$ iff $\forall b.\ t_1(b) < t_2(b)$. Clearly, not all version vectors are comparable. We write $t_1 \lesseqgtr t_2$ if vectors t_1 and t_2 are comparable.

Notation and Conventions. We enforce a one-to-one mapping between replicas and branches by adopting the convention that branch b_i always corresponds to replica i. Concretely this means that replica i only ever creates new versions on branch b_i. Also, replica i can only update its local copies of H and B, i.e., H_i and B_i. For e.g., $H_i[b_i \mapsto t]$ updates the head of branch b_i to t on replica i. Since H_i is simply an abbreviation of $H\ i$ in the formalism, $H_i[b_i \mapsto t]$ actually expands to $H_i[i \mapsto b_i \mapsto t]$. We exploit this notation in Fig. 8. We let $\delta_\odot$ denote the initial concrete state which is defined on the similar lines as the initial abstract state (Def. 3.1).

Rules. The transition relation $\twoheadrightarrow$ of the Quark distributed machine is defined in Fig. 8. Every transition rule of the abstract machine (Fig. 6) has a corresponding rule for the abstract machine. Fig. 8 however elides Fork and FastFwd in the interest of space. The rule Commit describes replica i committing a new version on the branch b_i with the given value n. The vector t' for the new version is obtained from that of the previous version t by incrementing the component b_i. The sequence number of the new version on b_i is $t'(b_i)$ and the branch map B_i is updated to reflect that. Note that for all $j \neq i$, H_j and B_j remain unchanged indicating that other replicas are not (yet) aware of this commit. Fork forks off a new branch b_j from b_i. A new replica j is assumed to take over b_j. The version vector t' of the branch head now has a new component b_j mapped to 1 to denote this is the first version on b_j. Map B_j is updated accordingly. Value $N(t')$ is same as its parent $N(t)$.

Merge describes the semantics of replica i merging a concurrent version from a (remote) replica j. The merging versions are heads of their respective branches b_i and b_j. Like its counterpart in Fig. 6, Merge insists that the LCAs of every other branch b_k with b_i and b_j be causally related. This condition is however expressed in terms of version vectors with help of the GLB ($\sqcap$) and comparison ($\lesseqgtr$) operators. For the LCA determination to be sound, the merging replica i needs to have an accurate conception of the current version history. Furthermore, there cannot be a concurrent merge happening elsewhere that undermines the judgment of replica i (reg. the safety of $i \leftarrow j$ merge). These requirements are enforced by the premise $\forall(k \in dom(H)).\ b_k \in dom(H_i) \ \wedge \ H_i(b_k) = H_k(b_k)$, which insists that replica i's knowledge of every other branch k be current. This condition effectively linearizes merges by requiring a merge to either *see* or *be seen* by every other merge. In practice this is achieved through global coordination (Sec. 5). Note that Merge only preempts a concurrent

Merge, not a concurrent Commit. A remote replica k is allowed to keep committing new versions on to b_k even as it remains unaware of the merge on replica i. Such leniency is imperative if the system were to retain the performance benefits of asynchronous replication.

The rule Sync captures asynchronous communication between a pair of replicas (i and j) to get one of them (i) up-to-date with the other (j). Unlike the other rules, Sync does not extend the version history graph, and therefore has no counterpart in Fig. 6. It merely updates replica i's knowledge of branch b_k if replica j happens to have later updates from k, i.e., $H_j(b_k)$ happens to be ahead of $H_i(b_k)$. The new versions on b_k known to j but not i are then simply replicated at i.

Refinement. The relationship between the Quark abstract (Sec. 3) and distributed (Fig. 8) machines is stated thus:

Theorem 4.1 (Refinement). *There exists a refinement relation R relating abstract and concrete states such that:*

- *R relates the initial abstract and concrete states, i.e., $R(\Delta_\odot, \delta_\odot)$.*
- *Forall δ, δ', Δ, if $R(\delta, \Delta)$ and $\delta \longrightarrow\!\!\!\!\!\rightarrow \delta'$, then there exists a Δ' such that $\Delta \longrightarrow \Delta'$ and $R(\delta', \Delta')$.*

The proof is done in Ivy. The refinement relation R essentially relates the version vectors of concrete semantics to the version histories of abstract semantics. A *happens-before* relation has been defined as a ghost state in concrete semantics to help us establish the refinement relation in Ivy.

Garbage Collection. One downside of the distributed semantics in Fig. 8 is that maps B and N that together track the version history grow monotonically as the execution progresses and new versions are created. Fortunately, this is easy to address as the execution only needs finite version history to compute LCAs. Since LCA version vectors monotonically increase, older versions with vectors less than the least known LCA vector can simply be garbage-collected. Moreover, a replica can make this decision locally without having to synchronize with its peers. In practice, applications may prefer to flush out older version history to a stable storage from where it can be re-created as necessary.

5 Implementation

We realize a prototype of Quark runtime for MRDTs as a lightweight shim layer on top of Scylla – an off-the-shelf distributed data store [22]. We rely on Scylla for inter-replica communication, data replication, persistence, and fault tolerance. Quark translates the high-level MRDT implementations in OCaml to their low-level representations in the backing store and orchestrates their well-formed distributed executions. Fig. 9 illustrates the overall architecture.

The implementation of Quark largely follows the design of Quark distributed machine described in Sec. 4. We manifest each component of the state, namely the branch map B, the value map N, and the head map H, as a column family (i.e., a table) in Scylla. The synchronization needed to linearize merges is implemented with help of Scylla's support for conditional updates (CAS operations) and expiring columns. The total order among merges is enforced with help of `Quorum` reads and writes. Each user process is assigned its own branch containing a replica of the MRDT. Version vectors are realized as associative lists and stored in Scylla as blobs.

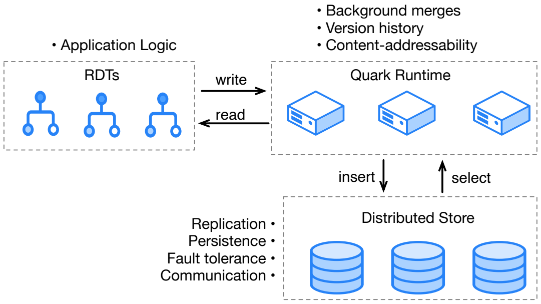


Figure 9. Quark implementation architecture

The implementation however differs from the formalization in two significant ways. First is in the treatment of MRDT values. Formalization assumes values to be atomic with no sharing in between them. In practice, however, an MRDT could be a linked data structure such as a binary tree, and two such values could share a significant amount of internal structure. Consequently, the size of the *diff* between two consecutive versions of a value could be asymptotically less then the size of the data structure itself, in which case it unreasonable to transfer the entire data structure over the network. To facilitate the efficient computation of diffs between versions of data structures, we implement a *content-addressable store* as a key-value table in Scylla where key is simply the SHA256 hash of the value. A linked data structure is stored as a collection of nodes, where each node links to the other by referring to its hash. The diff between two consecutive versions of a data structure would simply manifest as new entries in the content-addressable store reachable from the root of the new version. The new entries, being new data, are automatically replicated by Scylla, thus letting us reconstruct the new version at a remote location. The root hash of the new version is obtained from the value map N, which now maps version vectors to the *hashes* of values.

The second significant difference between the implementation and the formalization is the timing of merges. Operational Semantics in Sec. 3 and Sec. 4 interleave commits and merges such that only one of them executes at any given time. Since commits are initiated by the user in practice, interleaving them with strongly consistent merges increases their latency as perceived by the user. For the user-perceived latency to remain unaffected, it is important that a replica be always available to execute user requests. Quark ensures this by handling user requests in a foreground thread that is always allowed to commit to the local branch. Merges, on the other hand, are relegated to the background. A background thread constantly scans the remote branches for new versions, and if there are any, merges them into the latest version on the local branch after checking the necessary preconditions (Sec. 3).

Quark's background merges however pose a new problem as they create new versions on the local branch in the background while a user operation is manipulating an older version in the foreground. When the user attempts to write their version to the store, simply committing it would effectively override the concurrent updates from other users obtained via background merges. The solution, fortunately, is straightforward: Quark *merges* the user-submitted version with the latest version on the local branch to create a new version that includes the updates from either direction. Since this merge is fully confined to the local replica, it is guaranteed to not affect the LCA of the local branch in relation to any remote branch. To the external world, it appears as if the local branch has simply committed a new version that was derived from the older version. Since the commit operation now entails a merge, the merged version has to be returned to the user as the result of the commit. Consequently the `write` operation exposed by Quark is a function of type `Value` → `Value`, where `Value` is any MRDT.

6 Evaluation

In this section we present an empirical evaluation of Quark on two case studies. First is a collaborative document editing application – a common usecase addressed by several CRDT proposals [19, 20, 23]. Second is a replicated key-value store implemented using a mergeable Red-Black tree data structure.

6.1 Collaborative Editing

Quark's MRDT approach obviates the need to build a dedicated replicated data type for collaboratively-edited documents; an ordinary document format extended with a merge operation would suffice. While many data structures exist to represent text documents (e.g., ropes [5]), we decided to adopt the simplest representation of a document as a list of characters.

```
type doc = char list
```

While being simple, the advantage of this presentation is that we can simply reuse the three-way `List.merge` function of the list data type to merge documents. `List.merge` is a simple implementation of list merge algorithm (in 60 lines of OCaml) inspired by the GNU `diff3` algorithm [10]. We thus adopt a straightforward approach to building a collaborative document editor with the intention to keep the development effort low enough to be easily replicated. The convergence guarantee of Quark ensures that the simplicity of our implementation doesn't come at the expense of correctness. The aim of the experimental evaluation is to quantify the impact of Quark on the performance.

Our experiment setup consists of multiple collaborators simultaneously editing a 10000+ line document obtained from the Canterbury Corpus [7]. Each user holds a replica of the document and is assumed to be editing the document at the speed of 240 characters per minute or 1 character every 0.25s. At 6 characters per word, this amounts to 40 words per minute, which is the average typing speed of humans. Each edit is immediately persisted to the disk by creating a new version in the backing store. Thus there are at least as many versions of the document as there are edits. Such extensive versioning may be considered excessive in practice and could be disabled.

Latency. To measure the impact of Quark runtime on user writes, we measure the latency of the `write` operation, which includes the time spent merging the user version with the current version, and persisting the resultant version to the store. We conduct the experiments on a three-node cluster of `i3.large` machines located in Amazon `us-west2` data center. Each user connects to one of the machines, forks a new branch, and performs 1000 edits in succession, saving the document after each edit. We progressively increase the number of concurrent users editing the document from 3 to 60 and measure the impact of the increased concurrency on write latency. Fig. 10 shows the 10th, 50th (median), and 90th percentile latency values. The median and 10th percentile latencies remain more-or-less constant with a slight increase between 3 and 60 concurrent users. This is expected considering that Quark does not constrain the execution of user operations. The slight increase, we confirmed, is due to the increased latency of concurrent database writes. The 90th percentile latencies, however, show an initial increase before adopting a pattern similar to median latencies. We attribute this behavior to an idiosyncrasy of our multi-threading implementation, which is more likely to affect measurements at the extreme. In particular, we use OCaml v4.12 extended with Lightweight Threads (LWT) library, which offers concurrency but not parallelism. At lower number of replicas, the merge thread's lock requests succeed more often, resulting in the process spending more time merging than editing. We expect this pattern to smoothen out in OCaml v5.0, which introduces true multi-core parallelism. Notwithstanding this idiosyncrasy, the maximum value of 90th percentile latency measured (32ms) remains well below the the time between consecutive edits (0.25s), making it hard to perceive by a human user.

For comparison against a baseline, we have implemented an "SC" approach which achieves convergence by synchronizing each operation, i.e., executing it under strong consistency (SC). The SC implementation shares most of its

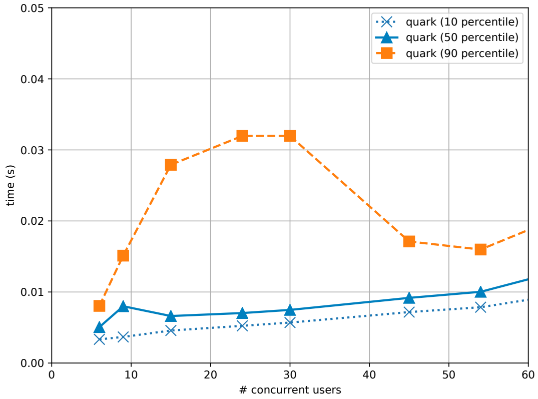


Figure 10. Latency of writes to a shared document under Quark.

code with Quark with the only change being that it wraps commits instead of merges inside a lock. As with Quark, we measured the write latency of the SC implementation while increasing the number of concurrent users (n). The median SC write latency increases super-linearly from 10ms for $n = 3$ to 63s (i.e., >1m) for $n = 60$, which is considerably more than the inter-edit latency of 0.25s.

Staleness. Our implementation of Quark relies on Scylla to replicate the contents of each branch across all the replicas as fast as the network allows. However, for a user A to see the changes made by the other user B, the changes have to be reflected in A's local version, which can only happen through a merge operation. Since Quark synchronizes merge operations globally, it induces additional delay before A can see B's changes. We call this additional delay *staleness* as with the progression of time, B's version known to A becomes increasingly stale. At the system-level, an increase in staleness effectively delays the convergence (but doesn't preempt it, as proved by Theorem 3.8).

To understand the effect of Quark on staleness, we quantify and measure it along with latency in the experiment setup described above. Staleness is defined as the time taken for a version committed on one replica to be merged into a concurrent version on a remote replica. To measure staleness, we annotate every version v with the timestamp t of the wall clock time of its creation. When v is is merged into a remote branch b at a later time t', the difference $t' - t$ denotes the staleness of v w.r.t the new version on b. One such staleness measurement is recorded for every merge that ever happens during the experiment. We compute 10th, 50th, and 90th percentiles of staleness values thus obtained. Fig. 11 shows the results. As evident, staleness increases steadily with the increasing number of concurrent replicas, which is expected considering that merges are synchronized, and increasing the number of replicas reduces the number of opportunities for a replica to merge. The increase is roughly linear in the number of replicas as our implementation passes the lock around in a round-robin fashion. Despite the steady increase, the 90th percentile staleness values remain low – less than 1s with the number of replicas under 30, and less than 2s for number of replicas under 60. While further optimizations might reduce staleness, a non-trivial staleness overhead is inevitable in Quark due to our use of synchronized merges to guarantee convergence.

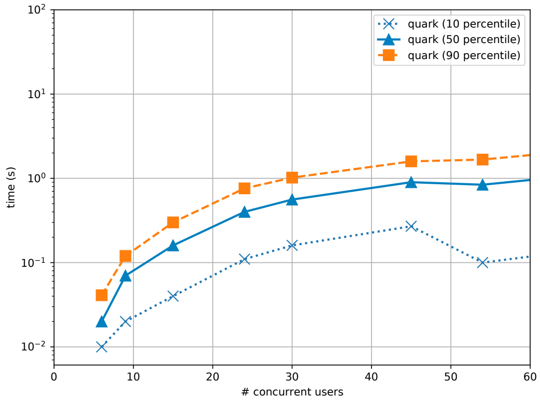


Figure 11. Staleness increases as the number of concurrent editors increase.

6.2 Key-Value Store

Our second case study involves a mergeable key-value store, which, unlike a text editor, is expected to work in the background supporting applications such as e-Commerce. The store is mergeable in the sense that it can merge conflicting values assigned to equal keys using value-specific merge functions. For instance, an e-Commerce application might store shopping carts as values, in which case the merge function on shopping carts is used to merge concurrent conflicting updates to a user's shopping cart. The key-value store is implemented using a Red-Black tree MRDT [14], which is a standard Red-Black tree implementation extended with a merge function. The merge function implements set merge semantics (Sec. 2) for concurrent insertions and deletions of keys, and defers to the value merge function for concurrent updates against a key. Concretely, `RBTree` is an OCaml functor parameterized on the `Key` and `Value` modules, where the latter is required to provide a `merge` function. We use integer keys and mergeable counter values in our experiments.

We use the same experimental setup as before – an increasing number of concurrent processes perform a random operation on the local version of the tree while merging with the remote versions in the background. Each process forks off a branch with an initial version of an `RBTree` of size 1000, performs an insertion, updatation, or deletion with probabilities of 0.5, 0.25, and 0.25. A new version of the tree is

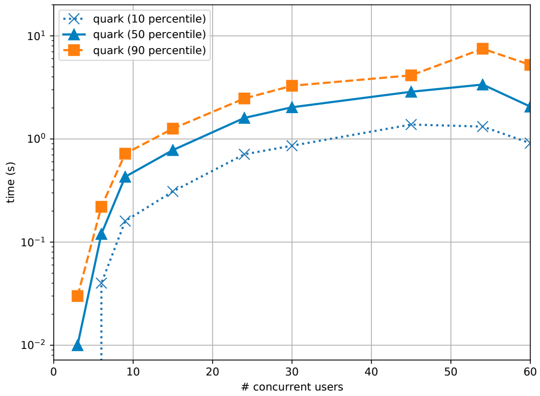


Figure 12. Staleness of key-value store merge operations.

committed after each operation. Each process performs 1000 operations with an inter-operation delay of 100ms. We measure the latency of each tree operation and the staleness of each merge. Latency measurements follow the same pattern as those for collaborative editing (Fig. 10), and have been elided. The median latency remains around 50ms for most of the experiment. Staleness measurements are plotted in Fig. 12. As before, staleness increases steadily in proportion to the increasing number of concurrent processes. The 90th percentile staleness exceeds 1s for 15 concurrent processes, and 7s for 54 concurrent processes. Unlike the collaborative applications, however, the number of replicas in a conventional web application tends to be low, which could limit staleness to a tolerable range. For instance, in the e-Commerce application described above, 3 replicas of the shopping cart can be kept in sync with each other with a 90th percentile lag (staleness) of 30ms.

Our experiments bring to the fore an inherent tradeoff among the competing concerns of Convergent RDTs, namely (i). The ease of programming, (ii) Latency, and (iii). Staleness. While CRDTs try to optimize for latency and staleness, they require a significant amount of development and verification effort to be expended to ensure convergence. In contrast, QUARK lets developers derive convergent-by-construction MRDTs from ordinary data types that are optimized for latency, but incur a non-trivial staleness overhead that delays the time to convergence.

7 Related Work

Mergeable Replicated Data Types (MRDTs) were introduced in [14], where authors also demonstrate an approach for deriving merge functions from first principles. Despite resulting in sensible merge semantics for several data types, their approach was never shown to guarantee the convergence of the resultant MRDTs. Indeed, as we demonstrate in Sec. 2, convergence of unrestricted MRDT executions cannot be guaranteed due to the presence of anomalies such as Fig. 4b and Fig. 5b. QUARK fixes this problem by extending MRDTs with a runtime that limits their executions to those that are guaranteed to converge.

State-centric replication was also explored in the context of CRDTs [23]. However, such state-based CRDTs require the replicated state to be organized as a lattice with the merge function acting as as a least upper bound operator. We eliminate this restriction in our setting with help of the QUARK runtime. A thorough comparison of our approach with the operation-based CRDTs can be found in Secs. 1 and 2.

Several verification techniques, program analyses, and tools have been proposed to reason about the program behavior in a weakly-consistent distributed setting [1, 2, 12, 17, 24]. These techniques treat replicated storage as a black box with a fixed pre-defined consistency models. The focus is on assigning appropriate consistency levels to operations so as to preserve application integrity. Such an approach results in assigning sequential consistency (SC) to all operations since the next weaker consistency model – causal consistency, is insufficient to guarantee convergence (as Sec. 2 demonstrates). Conversely, QUARK cannot reason about application-level invariants, such as `balance` ≥ 0 in a banking application. Thus both approaches confer complimentary benefits on application developers.

The meta-theory of QUARK abstract machine (Sec. 3 bears resemblance to the type safety proofs carried out in Wright and Felleisen's style [26]. Theorem 3.8 (Progress) is analogous to the Progress lemma of type safety, and Theorem 3.7 (Convergence) is analogous to Preservation. However, unlike the type-based approaches, safety in QUARK is enforced through run-time monitoring. Static enforcement of convergence in MRDTs using, for e.g., Session Types [13], is an interesting direction for future work.

QUARK's programming model is inspired by distributed version control systems in general, and Git in particular. To the best of our knowledge, operational semantics of Git has never been formalized. This is not a serious concern considering that humans are heavily involved in Git workflows, and the complexity of version histories manifest by Git is bound by the limits of human cognition. Nonetheless, it is possible to construct Git version histories that result in non-convergent and counter-intuitive final states [21, 25]. Such anomalies are more likely to occur if Git version histories are manifest commensurately with distributed executions of computer programs. QUARK's run-time monitoring preempts this possibility in our case.

Finally, the implementation of QUARK bears resemblance to Quelea [24] as both systems offload low-level concerns to an underlying data store. While Quelea enforces high-level invariants on applications built with CRDTs that are assumed to be convergent, QUARK enforces convergence of data types that are otherwise not guaranteed to converge.

Acknowledgments

We thank the anonymous reviewers and our shepherd Doug Lea for their thoughtful comments. First author has benefited from discussions with Anmol Sahoo and Suresh Jagannathan. Anmol has also developed and maintained the OCaml-Scylla library, which is a key piece of infrastructure underlying Quark.

A Proofs

Remark The following algebraic properties over sets are referred in the proofs:

- **Law1**: Set difference distributes over union. $\forall(A, B, C).\ A - (B \cup C) \ = \ (A - B) \cap (A - C)$
- **Law2**: $\forall(A, B, C, D).\ (A - C) \ \cap \ (B - C) = \emptyset \ \wedge \ C \subseteq D \ \Rightarrow \ (A - D) \ \cap \ (B - D) = \emptyset$
- **Law3**: Set union distributes over difference. $\forall(A, B, C).\ (A \cup B) - C \ = \ (A - C) \cup (B - C)$
- **Law4**: Set union distributes over intersection. $\forall(A, B, C).\ (A \cup B) \cap C \ = \ (A \cap C) \cup (B \cap C)$
- **Law5**: $\forall(A, B, C, D).\ (A - C) \ \cap \ (B - C) = \emptyset \ \wedge \ A \subseteq B \ \Rightarrow \ A \subseteq C$

Proof of Lemma 3.4. By induction on Δ. The initial state $\Delta_\odot$ vacuously satisfies the unique LCA condition. The inductive step has three cases:

- Case Commit: LCA function L is not updated in this case. Neither are any new branches created. Hence the proof follows from the induction hypothesis (IH).
- Case Fork: We fork off a new branch b' from b in this case. The LCA of b and b' in this case is clearly $H(b)$ – the version from which b' is forked. Indeed, this is the updated value of $L(b, b')$ in the conclusion.
  For every other branch b'', the LCA of b' and b' (i.e., $L(b', b'')$) is same as the LCA of b and b'' (i.e., $L(b, b'')$), which IH gurantees to be unique. This is because every ancestor of $H(b')$ is either equal of $H(b)$ or is an ancestor of $H(b)$, the hence lowest common ancestor of $H(b'')$ and $H(b)$ should also be the lowest common ancestor of $H(b'')$ and $H(b')$, which is indeed the updated value of $L(b, b'')$ in the conclusion.
- Case Merge: The proof follows from the premise of Merge, which requires that, for every branch b'' in the system, the LCA of b'' with merging branches b and b' be causally related, i.e, $L(b, b'') \longrightarrow^* L(b', b'') \vee L(b', b'') \longrightarrow^* L(b, b'')$. Fig. 7 helps visualize this condition in the most general case when (i). Branches b, b', and b'' are distinct, and (ii). Their LCAs $L(b, b'')$ and $L(b', b'')$ lie on a distinct pair of branches not equal to b, b', and b''. In the figure, once you merge b' into b, every version v that is an ancestor of $L(b', b'')$, i.e., $v \longrightarrow^* L(b', b'')$, will be a common ancestor of $H(b)$ and $H(b'')$. Clearly, $L(b', b'')$ is the lowest among such common ancestors. But the current lowest common ancestor of b and b'' is $L(b, b'')$. This leads to the condition where there are two lowest common ancestors – $L(b', b'')$ and $L(b, b'')$, *unless* both are ancestrally related. The premise $L(b, b'') \longrightarrow^* L(b', b'') \vee L(b', b'') \longrightarrow^* L(b, b'')$ guarantees that they are causally-related, ensuring the uniqueness of LCA.
- Case FastFwd: Similar to Merge

□

Proof of Lemma 3.5. Straightforward induction on Δ. □

Proof of Lemma 3.6. Proof by induction on Δ. We consider the cases below starting with then Merge case.

- Case Merge: In all the cases where $b_1 \neq b$ and $b_2 \neq b$, proof follows from IH because nothing changes for these branches in Δ'. We shall therefore focus on the case when one of b_1 or b_2 is b. Without loss of generality, let's assume $b_1 = b$. Two cases now depending on what b_2 is:
  - Case $b_2 = b'$: From the Merge rule, it's clear that $v_1 = H'(b) = v$ and $v_2 = H'(b') = H(b')$ and $v_0 = L'(b, b') = H(b') = v_2$. Since $v_2 = v_0$, $C(v_2) - C(v_0) = \emptyset$, hence the proof.
  - Case $b_2 \neq b'$. Let $b_2 = b'' \neq b'$. Hence $v_1 = H'(b)$, $v_2 = H'(b'') = H(b'')$, and $v_0 = L'(b, b'')$, which would either be $L(b, b'')$ or $L(b', b'')$ (see the premise of Merge). For notational convenience let's use v' to denote $H'(b') = H(b')$, v'' to denote $H'(b'') = H(b'')$, and v_{old} to denote $H(b)$. The hypotheses are listed below:

$$\begin{array}{ll}
C(v_{old}) - C(L(b, b'')) \ \cap \ C(v'') - C(L(b, b'')) = \emptyset & IH1 \\
C(v') - C(L(b', b'')) \ \cap \ C(v'') - C(L(b', b'')) = \emptyset & IH2 \\
L(b, b'') \longrightarrow^* L(b', b'') \ \vee \ L(b', b'') \longrightarrow^* L(b, b'') & H1 \\
C(H(b')) \supset C(L(b, b')) & H2 \\
C(H(b)) \supset C(L(b, b')) & H3
\end{array}$$

We will now destruct $H1$ and consider each disjunct separately.

- Case $L(b, b'') \longrightarrow^* L(b', b'')$: From Lemma 3.5 we know:

$$C(L(b, b'')) \subseteq C(L(b', b'')) \quad H4$$

The Goal is to prove the following:

$$C'(v) - C'(L'(b, b'')) \ \cap \ C'(v'') - C'(L'(b, b'')) \ = \ \emptyset$$

We shall now rewrite the goal through a series of equalities. First, since $C'(v) = C(v_{old}) \cup C(v')$, and $L'(b, b'') = L(b', b'')$ (since $L(b, b'') \longrightarrow^* L(b', b'')$ as per our assumption in this case), and $C'(v'') = C(v'')$:

$$(C(v_{old}) \cup C(v')) - C(L(b', b'')) \ \cap \ C(v'') - C(L(b', b'')) \ = \ \emptyset$$

Rewrite using Algebraic Law 3:

$$\begin{array}{c}(C(v_{old}) - C(L(b', b'')) \ \cup \ C(v') - C(L(b', b''))) \\ \cap \, C(v'') - C(L(b', b'')) \ = \ \emptyset\end{array}$$

Rewrite using Algebraic Law 4:

$$\begin{array}{c}C(v_{old}) - C(L(b', b'')) \ \cap \ C(v'') - C(L(b', b''))) \\ \cup \, C(v') - C(L(b', b'')) \ \cap \ C(v'') - C(L(b', b'')) \ = \ \emptyset\end{array}$$

Rewrite using IH2:

$$C(v_{old}) - C(L(b',b'')) \ \cap\ C(v'') - C(L(b',b'')) \ = \ \emptyset$$

Applying Algebraic Law2 with $C := L(b,b'')$:

$$\begin{aligned} &C(v_{old}) - C(L(b,b'')) \ \cap\ C(v'') - C(L(b,b'')) \ = \ \emptyset \\ &\qquad \wedge L(b,b'') \subseteq L(b',b'') \end{aligned}$$

Now both conjuncts of the goal follow from hypotheses $IH1$ and $H4$, respectively.

∗ Case $L(b',b'') \longrightarrow^* L(b,b'')$: From Lemma 3.5 we know:

$$C(L(b',b'')) \subseteq C(L(b,b'')) \quad H5$$

The Goal is to prove the following:

$$C'(v) - C'(L'(b,b'')) \ \cap\ C'(v'') - C'(L'(b,b'')) \ = \ \emptyset$$

We shall now rewrite the goal through a series of equalities. First, since $C'(v) = C(v_{old}) \cup C(v')$, and $L'(b,b'') = L(b,b'')$ (since $L(b',b'') \longrightarrow^* L(b,b'')$ as per our assumption in this case), and $C'(v'') = C(v'')$:

$$\begin{aligned} &(C(v_{old}) \cup C(v')) - C(L(b,b'')) \\ &\quad \cap C(v'') - C(L(b,b'')) \ = \ \emptyset \end{aligned}$$

Rewrite using Algebraic Law 3:

$$\begin{aligned} &(C(v_{old}) - C(L(b,b'')) \ \cup\ C(v') - C(L(b,b''))) \\ &\quad \cap C(v'') - C(L(b,b'')) \ = \ \emptyset \end{aligned}$$

Rewrite using Algebraic Law 4:

$$\begin{aligned} &C(v_{old}) - C(L(b,b'')) \ \cap\ C(v'') - C(L(b,b'')) \\ &\cup C(v') - C(L(b,b'')) \ \cap\ C(v'') - C(L(b,b'')) \ = \ \emptyset \end{aligned}$$

Rewrite using IH1:

$$C(v') - C(L(b,b'')) \ \cap\ C(v'') - C(L(b,b'')) \ = \ \emptyset$$

Apply Algebraic Law2 with $C := L(b',b'')$:

$$\begin{aligned} &C(v') - C(L(b',b'')) \ \cap\ C(v'') - C(L(b',b'')) \ = \ \emptyset \\ &\qquad \wedge L(b',b'') \subseteq L(b,b'') \end{aligned}$$

Now both conjuncts of the goal follow from hypotheses $IH2$ and $H5$, respectively.

- Case FastFwd: Fast forwarding is a special case of merge, hence the proof is going to be very similar to the Merge case. To reduce the FastFwd case to Merge, we make the following observations about the diferences between both the rules:
  - The subsumption relation $C(H(b)) \supset C(L(b,b'))$ in the premise of Merge is replaced with equality in FastFwd, but this is irrelevant is the subsumption premise is never used in the proof of Merge case.
  - $N'(v) = N(H(b'))$ in FastFwd whereas it is computed using merge in Merge. However, the current theorem says nothing about N, and the rest of Δ is independent of N, hence this is irrelevant.
  - $C'(H'(b)) = C(H(b'))$ in the conclusion of FastFwd whereas it is equal to $C(H(b)) \cup C(H(b'))$ in Merge. However since $C(H(b)) = C(L(b,b')) \subset C(H(b'))$ in the premise of FastFwd, $C'(H'(b))$ in the conclusion is indeed equal to $C(H(b)) \cup C(H(b'))$ like in Merge.

  Consequently, we can reuse the proof of Merge as a proof of FastFwd.
- Case Fork: Branch b' is forked from branch b. In all the cases where $b_1 \neq b'$ and $b_2 \neq b'$, proof follows from IH because nothing changes for these branches in Δ'. We shall therefore focus on the case when one of b_1 or b_2 is b'. Without loss of generality, let's assume $b_2 = b$. We now have two cases depending on what values b_1 can take:
  - Case $b_1 = b$: In this case $L'(b_1,b_2) = L'(b,b') = H(b) = H'(b)$. Hence, $C'(H'(b_1)) - L'(b_1,b_2)$ is equal to $C'(H'(b)) - C'(H'(b)) = \emptyset$. Proof follows.
  - Case $b_1 = b'' \neq b$: In this case, the proof follows straightforwardly from the inductive hypothesis that asserts the lemma for b and b'' in Δ because $L'(b',b'') = L(b,b'')$ and $C(H(b')) = C(H(b))$.
- Case Commit: Proof is trivial because c is a fresh commit id that does not exist in Δ.

□

Proof of Theorem 3.7. Proof by induction on Δ. Base case trivially satisfies as there are no two branches in $\Delta_{\odot}$. Inductive proof has four cases corresponding to the four rules in Fig. 6:

- Case Commit: Branch b is extended with a fresh commit c, hence there does not exist another branch whose head has same commit set as $H'(b)$.
- Case Fork: In all the cases where $b_1 \neq b'$ and $b_2 \neq b'$, proof follows from IH because nothing changes for these branches in Δ'. We shall therefore focus on the case when one of b_1 or b_2 is b'. Without loss of generality, let's assume $b_2 = b$. Two cases now depending on what b_1 is:
  - Case $b_1 = b$: Proof by definition as Fork defines $C'(H'(b')) = C(H(b)) = C'(H'(b))$ and $N'(H'(b')) = N(H(b)) = N'(H'(b))$.
  - Case $b_1 = b'' \neq b$: Proof follows from IH relating b_1 and b in Δ because $C'(H'(b')) = C(H(b))$ and $N'(H'(b')) = N(H(b))$.
- Case FastFwd: The proof is similar to the Fork case since $C'(H'(b)) = C(H(b')) = C'(H'(b'))$ and $N'(H'(b)) = N(H(b')) = N'(H'(b'))$.
- Case Merge: In all the cases where $b_1 \neq b$ and $b_2 \neq b$, proof follows from IH because nothing changes for these branches in Δ'. We shall therefore focus on the case when one of b_1 or b_2 is b. Without loss of generality, let's assume $b_1 = b$. Two cases now depending on what b_2 is:
  - Case $b_2 = b'$: The relevant premise of Merge is reproduced below:

$$C(H(b)) \supset C(L(b,b')) \quad H0$$

We shall prove that $C'(H'(b_1))$ cannot be equal to $C'(H'(b_2))$ by first assuming that they are equal and then deriving a contradiction.

$$C'(H'(b_1)) = C'(H'(b_2)) \quad H1$$

Since $b_1 = b$, and $b_2 = b'$, and $C'(H'(b')) = C(H(b'))$:

$$C'(H'(b)) = C(H(b'))$$

Since $C'(H'(b')) = C(H(b)) \cup C(H(b'))$:

$$C(H(b)) \cup C(H(b')) = C(H(b')) \quad H2$$

From Lemma 3.6 on Δ, we know the following:

$$\begin{aligned} &C(H(b)) - C(L(b,b')) \\ &\qquad \cap\, C(H(b')) - C(L(b,b')) \;=\; \emptyset \quad H3 \end{aligned}$$

Rewriting H3 using H2:

$$\begin{aligned} &C(H(b)) - C(L(b,b')) \\ &\quad \cap\, (C(H(b)) \cup C(H(b')) - C(L(b,b')) \;=\; \emptyset \end{aligned}$$

Rewriting using Algebraic Law3:

$$\begin{aligned} &C(H(b)) - C(L(b,b')) \,\cap \\ &(C(H(b)) - C(L(b,b')) \;\cup\; C(H(b')) - C(L(b,b'))) \;=\; \emptyset \end{aligned}$$

Rewriting using Algebraic Law4:

$$\begin{aligned} &(C(H(b)) - C(L(b,b')) \;\cap\; C(H(b)) - C(L(b,b'))) \\ &\cup\, (C(H(b)) - C(L(b,b')) \;\cap\; C(H(b')) - C(L(b,b'))) \;=\; \emptyset \end{aligned}$$

Simplifying, and Rewriting using $H3$:

$$C(H(b)) - C(L(b,b')) \;=\; \emptyset$$

Which is possible if and only if:

$$C(H(b)) \subseteq C(L(b,b')) \quad H4$$

$H0$ and $H4$ now result in a contradiction.

– Case $b_2 = b'' \neq b'$: Premises of Merge:

$$\begin{aligned} &C(H(b)) \supset C(L(b,b')) && H1 \\ &C(H(b')) \supset C(L(b,b')) && H2 \\ &L(b,b'') \rightarrow^* L(b',b'') \;\vee\; L(b',b'') \rightarrow^* L(b,b'') && H3 \end{aligned}$$

From the conclusion of Merge:

$$\begin{aligned} &C'(H'(b)) = C(H(b)) \cup C(H(b')) && H4 \\ &C'(H'(b'')) = C(H(b'')) && H5 \end{aligned}$$

And the premise of the theorem:

$$C'(H'(b)) = C'(H'(b''))$$

Due to $H5$, this is equivalent to:

$$C'(H'(b)) = C(H(b'')) \quad H6$$

From Lemma 3.6 on $H'(b')$ and $H'(b'') = H(b'')$:

$$\begin{aligned} &C'(H'(b)) - C'(L'(b,b'')) \,\cap \\ &\qquad C(H(b'')) - C'(L'(b,b'')) \;=\; \emptyset \end{aligned}$$

Rewriting using $H6$:

$$C'(H'(b)) - C'(L'(b,b'')) \;=\; \emptyset$$

Hence:

$$C'(H'(b)) \subseteq C'(L'(b,b''))$$

Rewriting using $H4$:

$$C(H(b)) \cup C(H(b')) \;\subseteq\; C'(L'(b,b'')) \quad H7$$

Let us distruct $H3$ and consider the two disjuncts separately:

∗ Case $L(b,b'') \rightarrow^* L(b',b'')$: From the conclusion of Merge:

$$L'(b,b'') = L(b',b'') \quad H8$$

Rewriting $H7$ using $H8$:

$$C(H(b)) \cup C(H(b')) \;\subseteq\; C(L(b',b'')) \quad H7$$

Which tells us:

$$C(H(b)) \subseteq C(L(b',b'')) \quad H9$$

Note that $L(b',b'')$ is LCA of $H(b')$ and $H(b'')$, hence $L(b',b'') \rightarrow^* H(b')$. From Lemma 3.5, we get:

$$C(L(b',b'')) \subseteq C(H(b')) \quad H10$$

From $H9$ and $H10$:

$$C(H(b)) \subseteq C(H(b')) \quad H11$$

Lemma 3.6 on $H(b)$ and $H(b')$ tells us:

$$\begin{aligned} &C(H(b)) - C(L(b,b')) \,\cap \\ &\qquad C(H(b')) - C(L(b,b')) \;=\; \emptyset \quad H12 \end{aligned}$$

Applying Algebraic Law 5 on $H11$ and $H12$:

$$C(H(b)) \subseteq C(L(b,b'))$$

Which contradicts $H1$. Hence $C'(H'(b))$ cannot be equal to $C'(H'(b''))$.

∗ Case $L(b',b'') \rightarrow^* L(b,b'')$: Very similar to the above case. Using the similar approach as above, we derive $C(H(b')) \subseteq C(H(b))$ (the opposite of $H11$), which leads us to $C(H(b')) \subseteq C(L(b,b'))$, thus contradicting $H2$.

□

References

[1] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings.* 249–260.

[2] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196. https://doi.org/10.14778/2735508.2735509

[3] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. ACM, New York, NY, USA, 761–772. https://doi.org/10.1145/2463676.2465279

[4] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. 2011. Extending Sledgehammer with SMT Solvers. In *Automated Deduction – CADE-23*, Nikolaj Bjørner and Viorica Sofronie-Stokkermans (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 116–130.

[5] Hans-J. Boehm, Russ Atkinson, and Michael Plass. 1995. Ropes: An Alternative to Strings. *Softw. Pract. Exper.* 25, 12 (Dec. 1995), 1315–1330. https://doi.org/10.1002/spe.4380251203

[6] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. 2010. Concurrent Programming with Revisions and Isolation Types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. ACM, New York, NY, USA, 691–707. https://doi.org/10.1145/1869459.1869515

[7] Canterbury 2021. The Canterbury Corpus. https://corpus.canterbury.ac.nz/descriptions/ Accessed: 2021-11-18 13:21:00.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. ACM, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[9] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59. https://doi.org/10.1145/564585.564601

[10] GNU Diffutils 2021. GNU Diffutils. https://www.gnu.org/software/diffutils/ Accessed: 2021-11-18 13:21:00.

[11] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying Strong Eventual Consistency in Distributed Systems. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 109 (oct 2017), 28 pages. https://doi.org/10.1145/3133933

[12] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL 2016)*. ACM, New York, NY, USA, 371–384. https://doi.org/10.1145/2837614.2837625

[13] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1, Article 9 (mar 2016), 67 pages. https://doi.org/10.1145/2827695

[14] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable Replicated Data Types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 154 (oct 2019), 29 pages. https://doi.org/10.1145/3360580

[15] Martin Kleppmann and Alastair R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (Oct 2017), 2733–2746. https://doi.org/10.1109/tpds.2017.2697382

[16] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[17] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906

[18] Kenneth L. McMillan and Oded Padon. 2020. Ivy: A Multi-modal Verification Tool for Distributed Algorithms. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 190–202.

[19] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. 2009. A Commutative Replicated Data Type for Cooperative Editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems (ICDCS '09)*. IEEE Computer Society, Washington, DC, USA, 395–403. https://doi.org/10.1109/ICDCS.2009.20

[20] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *J. Parallel Distrib. Comput.* 71, 3 (mar 2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006

[21] Russell O'Connor's Blog 2011. Git is Inconsistent. http://r6.ca/blog/20110416T204742Z.html Accessed: 2022-03-19 12:21:00.

[22] Scylla 2021. A Real-Time Big Data Database. https://www.scylladb.com/ Accessed: 2021-11-18 13:21:00.

[23] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (Grenoble, France) *(SSS'11)*. Springer-Verlag, Berlin, Heidelberg, 386–400. http://dl.acm.org/citation.cfm?id=2050613.2050642

[24] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI 2015)*. ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

[25] Tycon Blog 2019. Vagaries of Git Merge. http://tycon.github.io/git-inconsistencies.html Accessed: 2022-03-03 12:21:00.

[26] A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (nov 1994), 38–94. https://doi.org/10.1006/inco.1994.1093

[27] Z3 2022. The Z3 Theorem Prover. https://github.com/Z3Prover/z3 Accessed: 2022-03-20 13:21:00.

[28] Marek Zawirski. 2015. *Dependable Eventual Consistency with Replicated Data Types.* Ph.D. Dissertation.

[29] Yuqi Zhang, Yu Huang, Hengfeng Wei, and Jian Lu. 2019. Remove-Win: a Design Framework for Conflict-free Replicated Data Collections. arXiv:1905.01403 [cs.DC]