JEPSEN

Replicant 0.0.10

Kyle Kingsbury

2019-10-25

Replicant is a set of client libraries and an HTTP server for building causally consistent, totally-available distributed applications. Like Bayou and Eventually Serializable Data Systems, Replicant augments this causal order with eventual convergence to a total serial order of transactions. We discuss Replicant's safety properties, as described by design documents and associated documentation. We have not evaluated Replicant experimentally. This report is intended for Replicant internal use, rather than publication. This work was funded by Rocicorp, LLC (the makers of Replicant).

1 Background

Interactive web and mobile applications confront engineers with a classic challenge in distributed systems: synchronizing state despite high network latency. Mobile applications, in particular, may experience effective network latencies on the order of minutes to weeks, if, for example, a user enters a tunnel, visits a festival, or travels to a country without service. Even when service is available, congestion or noise may result in latencies which would frustrate interactive users.

To reduce frustration, responsive web and mobile applications often queue and retry updates, and while waiting for server responses, perform some sort of *state prediction*. For instance, when a user types a message in a chat application, the application may queue and retry the message several times—but while waiting, show the message in the conversation view. Once the server acknowledges the message, the client can stop retrying. Some applications perform buffering and prediction on an ad-hoc basis for specific actions. So-called "offline-first" applications take this approach for *every* action.

Performing updates asynchronously introduces new challenges. In particular, certain consistency models, like sequential consistency or serializability, are impossible to provide offline. This implies that web and mobile applications must choose between consistency and offline availability.

2 Replicant

Replicant is a framework for writing offline-first applications. It provides a transactional key-value store shared between clients and servers. Transactions are deterministic JavaScript functions. Any client, and any server, can execute any transaction at any time. Transactions on clients take effect immediately, resulting in a new predicted state, and are asynchronously replicated to servers. Servers use a strict-serializable database to construct an authoritative, total order of transactions, which is consistent with the causal relationships between transactions. As clients learn this authoritative transaction order, they unwind and replay transactions in the authoritative order.

This approach has several advantages. The state at any replica is always the product of a total order of transactions. This implies that if every transaction in the system individually preserves some invariant I, and I holds for the initial state, then I holds for every state at every replica. Furthermore, the final serialization order will be consistent with the causal order of individual transactions. This prevents numerous anomalies: for instance, a user's reply to a question in a chat will always appear after the question itself.

However, this unwinding-and-replying behavior brings unique challenges. The effects of "committed" transactions are temporarily unstable: a transaction might go through several reorderings before being finalized. For instance, a user who registers an account with a particular username might observe their transaction succeed, and go on to use the service—only to discover later that their account-creation transaction had actually *failed*, because some other person registered that username instead.

While Replicant's local transactions execute against this speculative, reorder-able state, the consistent prefix is also discoverable through a listener API. Users can ignore the results of locally executed transactions, and instead rely on a listener API to discover their final results.

2.1 Conflicts

A classic challenge in causal consistency is the problem of *convergence*: while causal ensures that every replica executes operations in an order that respects causality, it does not require that those replicas converge to the same state. *Causal+*, which also ensures convergence, can be achived by deterministically resolving updates to each key individually, as in COPS. However, deterministic resolution on a per-key basis can introduce application-level inconsistencies by mixing updates from different transactions together. For instance, if one user sets two different records to A, and another user sets those records both to B, the resulting state could be A for the first record, and B for the second. Replicant avoids this problem by applying transactions atomically.

Some causal systems expose write conflicts to the user, possibly with causal information about the structure of the divergence, and require that the user manually (or automatically) resolve the conflict. This approach has its own challenges: manual resolution may be time-consuming and confusing for users, and automatic resolution requires that users define an associative and commutative merge function. This approach is similar to CRDTs, except that idempotence is provided by the database system. Replicant does not expose conflicts to users: conflict resolution is implicit in transaction reordering.

The Replicant design document states that "developers rarely need to think about conflicts." We are not convinced this is correct: conflicts are inherent to the problem domain, and must be reasoned about.

2.2 When is a Transaction Safe?

If one considers a transaction in Replicant committed only when it has been acknowledged by the server as a part of the stable prefix, then Replicant could be a strict-serializable system—where transaction latencies may become arbitrarily long during network outages. Of course, this requires that users diligently ignore the return values from the client's `exec` function, and instead use the `onChange` listener to observe the final results of submitted transactions. Since the Replicant client keeps a copy of the last known authoritative database state, it can offer serializable read-only transactions even offline.

What if one uses Replicant as intended, and trusts the values returned by `exec`? When is this safe? We know that strict serializability is out of the question, since locally queued transactions cannot possibly be visible to other clients. Could it be serializable?

Serializability requires that the effects of transactions be equivalent to some total order. It would suffice to show equivalence of the results of locally executed transactions to Replicant's final transaction order.

By definition, the final transaction order will be consistent with the currently known stable prefix. To preserve causal ordering, it will also be consistent with the local order of pending transactions. We need not concern ourselves with *every* possible reordering. Instead, we need only show that our chosen transaction f commutes with every concurrent and future transaction. By "commutes", we mean that for every concurrent or future transaction g, $f \circ g = g \circ f$. To show this equality in general, we must reason not only about the "current" state of the database, but over *every possible state* of the database, since at any time, another client might choose to submit a transaction which performs arbitrary write.

As an example, take an insert of a new row with a predetermined globally unique key. Since the key is globally unique, this transaction commutes with every possible other transaction.

Similarly, adding and subtracting to a counter is safe, so long as we guarantee that no future transaction performs a non-commutative operation on that counter. Resetting the counter to zero would violate commutativity, of course. Surprisingly, so would any *read* of the counter, because the return value of the read transaction depends on precisely which increments and decrements have taken place prior. In particular, consider two clients which concurrently add 1 and 2 to a counter initialized to 0. Local reads on each client could return 1 and 2 respectively, even though no serial order of transactions could produce both of those values.

Of course, serializability violations don't mean the system is *semantically* unsound. It's just that serializability is one of the most powerful tools we have to reason about arbitrary invariants. There are many applications for which serializability violations aren't particularly meaningful, and for these applications, developers need not prove commutativity of their transactions.

In our counter example, we might say that users are not affected by transient reads of "impossible" counter values, so long as *eventually*, the system converges on the sum of all operations.

Many kinds of time-ordered logs, like chats, forum threads, and bank account ledgers accept temporary reorderings, so long as the history converges on some stable prefix after a reasonable time. Transactions which append elements to a log in Replicant can model these systems perfectly well, so long as users do not expect the order of recent log operations to be stable.

Similarly, Replicant's transaction atomicity allows users to safely transfer funds back and forth between simulated bank accounts, by incrementing one account and decrementing the other in a single transaction. The total value of funds in the system will remain constant *regardless* of reorderings. However, we disagree with Replicant's design document: performing any kind of *non-monotonic* computation, like enforcing a minimum balance, can result in anomalies. In particular, a user with $100 in their account could spend $80, then $60, on two devices concurrently, and observe both transactions succeed. At some future point, Replicant will finalize the order of those transactions, one of their minimum balance checks will fail, and the transaction will retroactively abort. Users must a.) be aware of this possibility, b.) listen for changes in past transactions appropriately, and c.) defer or undo any side effects.

2.3 Equivalence to CRDTs

Replicant's design document spends a good deal of time establishing that Replicant's approach is superior to CRDTs. However, readers may have noticed that the examples we have just discussed correspond to the behavior of common CRDTs. In fact, Replicant's speculative transaction behavior *is* a CRDT.

Replicant's database is a key-value store, which we can represent as a single value: a map. Transactions are simply functions over that map. To provide Replicant's transactional atomicity guarantees, we can store a set of transactions, each including the function to apply to the database, the arguments to that function, a uniqueness token (e.g. a Flake ID, and some kind of timestamp. To merge two copies of the database together, we merge the transaction maps with set union. To derive the "current state" of the database, we start with an initial state (say, an empty map), sort the transactions (say, by timestamp, then uniqueness token), and apply each one in order. This approach is very similar to MochiMedia's Statebox.

To provide causal consistency, we can choose transaction timestamps such that they preserve causality. For instance, we could use version vectors, which provide causal information. This ensures that the sorted order is consistent with the causal order. So long as timestamps monotonically advance and nodes periodically communicate, the transaction order will also trend towards a stable prefix.

Moreover, any CRDT can be encoded in Replicant by storing the state of the CRDT in some object, and for updates, calling the CRDT merge function in a transaction, to combine the current state of the CRDT with the state resulting from one's update. Replicant's speculative transactions are therefore equivalent to CRDTs.

2.4 But With Order

While Replicant's speculative transactions are equivalent to a CRDT, Replicant also establishes a serializable total order—and this order allows Replicant to do several things which CRDTs can't.

In particular, operation-based CRDTs have a garbage problem: they must keep track of every operation forever. Moreover, some operations, like resetting a counter, or performing a compare-and-set on a register, are fundamentally unsafe in a CRDT without coordination. As Shapiro et al note, this coordination requires atomic, unanimous agreement between all replicas. However, this may be impractical when clients are transient or frequently disconnected. Leția et al's approach solve this problem by establishing a group of *core* nodes, whose membership is expected to be (relatively) stable. Core nodes (e.g. servers) execute the agreement protocol for garbage collection, and other nodes (e.g. clients) passively learn of the servers' decisions via a monotonically increasing *epoch* system.

Similarly, Replicant servers establish a total transaction order via their backing database. This order allows Replicant to perform garbage collection safely: nodes can use the stable prefix of the transaction order to compute a snapshot of the database state, and discard the transaction log prior to that point—although transaction identifiers must be retained in order for clients to learn which of their pending transactions have committed. Likewise, clients can take advantage of the total order to execute non-commutative operations safely.

Replicant offers two chief advantages over state-based CRDTs. First, users need not provide a merge function which is associative, commutative, and idempotent. Idempotence is taken care of by unique transaction identifiers. Convergence is ensured via transaction determinism (which Replicant can experimentally verify) and the ever-growing stable prefix of the transaction log.

Of course, convergence is not *safety*. Last-writer-wins is a trivial CRDT that can be applied to any datatype, and it converges—but it exhibits lost updates. Similarly, speculative Replicant transactions may violate safety guarantees unless users reason carefully about equivalence of their transactions under reordering. However, when users are *not* confident in the reordering-safety of some transaction, they can ignore the return value of `exec`, and instead treat a transaction as committed only when they observe its final ordering.

This allows users to build applications which are a transparent hybrid of always-available, commutative transactions, mixed with indefinitely-blocking, strict-serializable transactions. The Replicant API doesn't yet represent these two classes of transactions as first-class objects, but we believe it could with minor changes.

3 Related Work

Bayou, described in this 1995 paper from Xerox PARC, used rollbacks and redos to force the eventual convergence of operations according to a global serialized order. Like Replicant, it was designed for use in mobile applications, where nodes could perform local reads and writes without the need for coordination. Also like Replicant, Bayou updates had two states: *tentative* and *committed*, based on whether or not their conflicts had been stably resolved by the total order. However, Bayou considered conflict resolution intrinsic to the problem of offline computation, and users were expected to define merge functions based on their application semantics. Replicant's conflict resolution is implicitly encoded by transaction structure.

Building on Bayou's legacy, IceCube described an approach to reconciling divergent operation logs between remote replicas, based on the observation that reconciling based on some arbitrary order, like timestamps, could result in more transaction conflicts than were strictly necessary. IceCube treats log ordering as a constrained optimization problem, and attempts to find log orders which satisfy ordering constraints between transactions (e.g. causal relationships) as well as user-specified constraints on transactions, like pre- and post-conditions. Replicant's total order respects causal constraints, but it does not appear to minimize conflicts in this way.

In 1992, Ladin et al published Lazy Replication, which allowed clients to perform operations locally, and lazily replicate them between nodes. Like Replicant, the state at any node is always a result of some order of atomic operations. Users may indicate causal dependencies on prior transactions, which will be respected when applying transactions at every replica. They also described two types of stronger transactions: *forced* operations, which are totally ordered with respect to other forced operations, and *immediate* operations, which are totally ordered with respect to all operations. However, this approach involved complex multi-part timestamps, could only express read-only or write-only operations, and required user-specified dependencies between non-commutative transactions to prevent replica divergence.

Eventually Serializable Data Services extended Lazy Replication's ideas by forcing an eventual total order for operations, like Replicant and Bayou. In this model, most operations are required to tolerate reorderings, but users may execute *strict* operations which are totally ordered before commit. Like Replicant and Bayou, Eventually- Serializable Data Services (user-specified) causal orders between transactions. We believe all three systems are quite similar in terms of safety properties.

In the early 2010s, the need for quickly-responsive web applications led to client+server frameworks like Meteor, which provided the illusion that a single database was available both locally and remotely. Meteor performed client writes asynchronously, but applied those operations to the local copy of the database immediately to provide state prediction, much like Replicant. Similarly, authoritative data from the server could overwrite client's speculative state at any time.