Selling Tickets Efficiently and Safely: CRDTs with Leases

Eugen Lucchiari Hartz, Samuel Horáček, Paul Hübner

14.03.2024

1 Introduction

In distributed systems, a significant challenge is to manage resources across multiple servers while ensuring as much consistency, availability and partition tolerance as possible. CRDTs (Conflict-Free Replicated Data Types) provide a way to achieve all three by utilizing a weak form of consistency: eventual strong consistency. However, these do not support monotonic operations. **This report discusses to what extent it is possible to augment CRDTs with non-monotonic operations in the form of stock limitations.**

In this report, we discuss the design, implementation, and testing of a hybrid-property CRDT system that introduces the notion of "leases" to CRDTs: individual replicas obtain permission to consume units of stock. This is done in a coordinated manner such that stock limitations are never exceeded. We implement a counter and set CRDT to illustrate this proof of concept.

To exemplify the practical application of this, we consider the example of an online ticket retailer that has a specific number of tickets/seats (stock). Ticket sale nodes are geographically distributed to provide the best user experience for the buyer. Each replica is a CRDT that keeps track of how many tickets are sold. Using the lease system, it can be ensured that no more tickets than seats available are sold. Selling a ticket "uses up" a lease, whereas refunding one replenishes a lease. Consequently, we mostly retain the performance benefits of a highly available and partitioned system while also providing domain-specific guarantees.

This report is structured as follows: first, we highlight our assumptions and reveal our design with their trade-offs in Section 2. Then, we showcase our implementation in Section 3. We evaluate our system with respect to correctness and real-time performance benchmarking in Section 4. We provide insights into further work and improvements in Section 5. Lastly, we summarize our findings in Section 6. Our codebase can be found on GitHub: `https://github.com/Arraying/LCRDT`.

2 Design

Our proposed system is conceptually made up of CRDT replicas that, when needed, coordinate to ensure they do not violate limitations reflected in the real world. In our case, this limitation is represented by an item stock. When clients add items to a cart, we aim to ensure their shopping carts do not contain more instances of the item than its actual stock. This coordination is achieved using a lease management service at every CRDT.

In this section, we describe both components in depth. We provide an insight into our assumptions and logic, focusing on possible alternatives and inherent trade-offs.

2.1 Overview

Our system will run in one of two modes of operation at all times, depending on whether coordination is required. Regardless of mode, our utmost priority at all times is ensuring the number of allocated leases does not exceed stock.

When all CRDTs have sufficient leases In this case, CRDTs will serve their local state, synchronizing periodically. Our system is eventually (strongly) consistent, highly available, and partition tolerant.

When lease coordination is required We cannot make progress during partitions and consequently may sacrifice availability under some conditions in order to guarantee atomic consistency. We cannot guarantee all 3 as per the Brewer's theorem [2].

2.2 Assumptions

It is important to formalize the set of assumptions under which the system is considered and operational.

- The system operates in an asynchronous environment, which implies we make no timing assumptions regarding bounds on computation time, communication delays, or bounds on physical clock drift rate.
- An inherent requirement of both CRDTs and our lease management is support for crash-recovery failures. We will therefore assume the crash-recovery model, one in which persistent storage is reliable and fault tolerant.
- We assume arbitrary network partitions will arise infrequently and will be short-lived. In other words, they will not violate consistency during lease manipulations. This scenario is revisited in Section 2.4.
- The communication between CRDT replicas is asynchronous and it is assumed FIFO perfect links are available to us.
- Our broadcast is a (FIFO-)reliable one. This is required for coordination and gives an increased rate of convergence to consistency at very little cost, even though it is not a requirement of CRDTs.
- Events emulating method calls within a process or between processes in a single node do not fail and eventually return their outcome.
- The network structure is static, and every process can directly communicate with every other process.
- The total number of stock is static and will not change throughout the lifetime of the program.
- Leases are revoked on a best-effort basis. A node is under no obligation to revoke leases when requested to do so if it cannot, such as when it does not have sufficient leftover leases.

2.3 System Description

This section introduces in depth both sub-systems of every node, a CRDT supporting monotonic operations and a lease management (LM) algorithm introducing support for non-monotonic ones. An assigned lease entitles a CRDT to utilize one unit of stock. What this entails depends on the CRDT used. We assume a lease is assigned indefinitely (i.e. it does not expire), though it may be attempted to be revoke it manually if unused. This lease management is done to avoid coordination at every step.

2.3.1 CRDTs

CRDTs represent a relatively novel approach to optimistic replication, formally defined in 2011 [6], that has appealing properties for systems looking to offer availability and partition tolerance, without the need for coordination, relying purely on synchronization. This approach, providing strong eventual consistency, implies individual replicas might temporarily out of sync, but will eventually converge to the same state. This consistency is often sufficient enough for many applications.

Not relying on coordination implies the ability to overcome netsplits, to safely handle concurrent updates, and better resiliency, as even a single process can make progress.

There are two theoretically equivalent [6] approaches to CRDTs – operation-based (CmRDTs) and state-based (CvRDTS). Our approach utilizes CvRDTs as they are easier to implement, offering easy-to-achieve flexibility. This is at the cost of having to keep track of potentially large configurations, that need to be synchronized across replicas. This is a trade-off we deem acceptable for the scope of this project.

Our synchronization is timed-based, which allows replicas to be synchronized periodically. The lower the timeout, the more consistent replicas are with one another (as they synchronize more frequently). This synchronization is conceptually simple, as as CvRDT state merges do not cause conflicts across replicas [4].

Implementations Our system was designed with the possible support of different CRDT implementations in mind. We demonstrate this by directly supporting/implementing two different CRDTs:

PN-Counter Positive-Negative Counter effectively combines two Grown-Only Counters, in order to support both increments and decrements. Every process has a tuple containing the number of increments and decrements, and every CRDT has the tuple of every process. The value of the counter is the difference between the sum of increments and the sum of decrements. Merging is done by taking the maximum for both entries in the tuple, for every process-tuple.

In this implementation, using a lease corresponds to incrementing a counter. On the other hand, decrementing the counter results in reclaiming of a lease, to be used again. Thus, one lease increases the counter's value by one. This implies the counter's value can be at most equal to the total stock. We also impose a minimum value of 0.

OR-Set An Observed-Remove Set keeps track of two sets (add and remove) on a per-entry basis. Both of these sets are made up of unique values. Adding is done by adding a unique element to the add set and removing consists of assigning to the remove set the union of add and remove sets. An element is contained in the OR-Set if the set difference of the add and remove sets is non-empty. Merging is done by making unions of corresponding add and remove sets.

We treat an element to be a tuple $U \times I$ where U is unique IDs (users in our shopping cart scenario) and I is items (products in this scenario). Then, owning n leases corresponds to the ability to, for each item $i \in I$, add at most n tuples where the second element is i. Unlike the PN-Counter, this lease system is domain-specific to facilitate a shopping cart scenario, and highlights how leases can be used to define custom behavior.

2.3.2 Lease Management

Lease management, as per its specification below, supports the allocation and deallocation of leases to CRDTs. Each of these operation is considered a transaction, bearing its own transaction ID (identifier). As a protocol representing a form of strict coordination, it was implemented as a variant of the Two-Phase Commit (2PC). This strict form of coordination provides certain consistency and makes it easier for us to reason about it, as there is always at most one round of coordination at any time.

2PC is, as the name suggests, structured into two phases. A prepare (sometimes called proposal) and commit (commit-or-abort) phases. In the prepare phase, a dedicated leader process (coordinator) contacts all followers to commit a value. They all respond with their OK or Abort votes. If all agree to commit, the coordinator issues a Commit. Otherwise, an Abort is issued.

Let a transaction txn be denoted as $\langle \texttt{allocate}, n, p_i, \rangle$ and $\langle \texttt{deallocate}, n, p_i \rangle$ to allocate to and deallocate n leases from CRDT p_i, respectively. The lease management has the following interface:

- **Request**: $\langle \text{LM}, \text{start}|\text{txn} \rangle$
- **Indications**:
  - $\langle \text{LM}, \text{prepare}|\text{txn} \rangle$
  - $\langle \text{LM}, \text{commit}|\text{txn} \rangle$
  - $\langle \text{LM}, \text{abort}|\text{txn} \rangle$

Similarities to conventional 2PC What is common for both is that coordination is blocking, as we do not have timeouts leading to aborts. In our case, using a timeout to abort or retry would lead to encountering the same problem again, but the achieved progress would be lost. If arbitrary messages could be dropped, thanks to the assumptions of FIFO perfect links and temporary network partitions, all participants eventually receive all messages in the correct order, once the connection is re-established. Thus, there is no need for timeouts commonly used to handle this scenario.

What is also common is unanimous agreement. In our case, in comparison to other coordination approaches, we can not rely on a majority (quorum) for agreement, as we require all nodes to reflect the same state. This implies also the need to contact the node whose leases are being manipulated, which would not be guaranteed by a quorum.

We decided to use 2PC as opposed to Three-Phase Commit (3PC) to ultimately reduce complexity and scope, and make it simpler to reason about our design. 3PC as a benefit is discussed in Section 5.

Differences to conventional 2PC One of the biggest differences between our coordination to conventional 2PC is that the coordinator process fulfills the roles of both the coordinator and follower, and is thus required to vote. This simplifies the implementation, as the coordinator keeps track of the same state as a follower would. That being

said, this adds the problem that a single failure can now lead to the entire protocol being blocked.

Another significant difference is our optimistic approach to committing. Followers optimistically apply every change once they receive a Propose. Only once an Abort is received, is this change rolled back. This is an optimization as we assume commits are more frequent than aborts, as the lease change concerns only a single node's leases.

The last noticeable difference (to some variants of 2PC) is that the coordinator does not collect acknowledgments from each follower whether the decided commit action, executed in the commit phase, was successful. These acknowledgments could be used for logging but were not considered needed, as there is no simple way of acting upon potential errors.

2.4 Behaviour

This section describes how our design functions in optimal conditions, as well as in possible scenarios that would test the consistency of our proposed lease management protocol. Finally, we describe the limitations of our coordination and possible problems.

2.4.1 Nominal Operation

Successful lease allocation/revocation The coordinator issues a Prepare request to all nodes asking if a transaction – in our case, a lease manipulation – can be executed. All nodes write this change to their write-ahead log, which is stored persistently, they update their lease state and respond with OK. Once these votes are received by the coordinator, it issues a Commit message. Once received by followers, they update their write-ahead log to reflect the success of this transaction, and the coordination ends.

Unsuccessful lease allocation/revocation There are two main sources of aborts – if we try to deallocate more leases from a node than it has, or if we try to allocate more leases than we have stock. The former scenario, in comparison to the latter, can be detected only by the replica, the one whose leases are being manipulated, as it is the only one knowing at all times their actual amount (as per eventual strong consistency, the others may not know in real-time). The latter scenario is simpler, all nodes know at all times how many leases each node has (due to our consistency model for leases), and can determine if total stock limits are being violated. An Abort message is issued by at least one of the followers. Once the coordinator learns this, it sends an Abort to all followers, they each revert this change due to information stored in the write-ahead log.

2.4.2 Crashes

In these scenarios, we look at how crashes in coordination are handled. It should be reiterated that we assume communication between CRDTs and their corresponding lease management process does not fail.

Coordinator crashes before Prepare phase In this scenario, 2PC is correct, as the coordinator fails before the algorithm starts. If the coordinator is offline, this message will eventually be delivered to them due to the assumption of the perfect link.

Coordinator crashes after receiving some/all votes This scenario is handled by contacting all followers who have not yet voted (OK/Abort), if any. Using a recovery signal, the coordinator makes each of these followers resend their votes, in order to

continue to the protocol's next phase. Since we are using FIFO links, we can be sure that this message will be delivered to the follower after the initial Prepare, if applicable.

If all followers have voted, the coordinator knows the outcome of the transaction and re-sends the outcome to all followers. These act upon this message only if they have not received it already (for this transaction ID).

It has to be said that while the coordinator has failed, the followers are left in an uncertain state. This means no progress in lease allocation can be made.

Follower crashes before receiving a Prepare message If a follower crashes before receiving a Prepare message, it enters recovery mode by contacting the coordinator. The coordinator re-sends its Prepare request if there is one. This scenario also covers the cases where a follower crashes before 2PC is even started, or before the Prepare message is even sent by the coordinator. In these cases, no cooperation is needed to recover.

Follower crashes after receiving a Prepare message If the coordinator has not received the response, we cannot be sure whether the follower crashed before or after applying the change locally. Thus, to ensure consistency we are pessimistic and abort.

Follower crashes after Commit has been decided If a follower crashes after Commit has been decided, because of our optimistic approach of realizing the operation during the Prepare phase, consistency is never violated. If applicable, the outcome message is re-sent. Nonetheless, the Commit outcome is logged to the log. That way, if the outcome is replayed, the follower knows whether or not it has handled it.

Follower crashes after Abort has been decided On the other hand, if a follower crashes after Abort has been decided, and before it is able to revert this change, it will do so once it recovers. The coordinator re-transmits the outcome. Using the information in the write-ahead log, it reverts the operation in question and logs an Abort.

2.4.3 Network Partitions

Network partitions represent scenarios where nodes form groups that are not able to communicate with one another. This implies network partitions can lead to the network dropping an arbitrary amount of messages in transit [2]. However, in our case, since we assume that these are temporary and our links are FIFO perfect, messages will eventually get delivered.

Traditionally, if arbitrary network partitions were to arise during the agreed-upon commit phase of 2PC or even 3PC, coordination consistency is not guaranteed. In fact, one could "split" the commit phase into an arbitrary number of sub-phases, but it is not possible to guarantee a network partition will not arise during the final commit sub-phase. This is similar to the Byzantine general's problem.

However, thanks to the combination of FIFO perfect links (in the form of reliable broadcast) and optimistic committing, we ensure consistent and correct lease behavior. Due to our optimistic approach, only dropped Abort messages could impact lease assignment and thereby allocate more leases than stock. However, we show in Section 4.1 that this is never the case in our system.

2.4.4 Shortcoming

Even though our design covers the scenarios described above, there is still one major shortcoming.

The coordinator and at least one follower crash during commit phase In 2PC, it is impossible, in case of failures, to know with absolute certainty if all followers have successfully executed the agreed action. For example, if after instructing the followers to Commit, both the coordinator and one of the followers that has committed crash, it is impossible in our model to recover from this scenario into a consistent state. The coordinator will consider this a successful transaction and will not abort, even though the Commit message might have not reached all followers.

This is an inherent limitation of 2PC. It can mostly be resolved by using the 3PC algorithm, which introduces another communication step [1].

3 Implementation

In order to implement this system, we decided to use Elixir. Elixir runs atop the Erlang/OTP ecosystem, which provides many benefits, such as a message-driven asynchronous model, source-FIFO message delivery, and many quality-of-life abstractions. Furthermore, Elixir's concise syntax and Mix ecosystem allowed us to create a highly concurrent and testable system with ease.

Our communication assumptions are met with this implementation. We use message passing to simulate an asynchronous environment. Erlang provides FIFO links out of the box, and best-effort broadcast was satisfied due to the system running on a single Erlang node.

For persistent storage like write-ahead logs or state snapshots, we used Erlang's `:erlang.term_to_binary` and vice versa. We assume that files on disk persist forever (and consider disk replication to be out of the project's scope).

3.1 Highlights

Having implemented the system with the previously described properties, in this section, we present some of the highlights that made this possible.

Supervision and system structure One of the highlights is the system structure that we used. By creating the supervision tree as shown in Figure 1, we were able to achieve crash-recovery very easily. We coupled the CRDTs with their respective lease management nodes such that if one failed, both would restart. This ensures that both remain in the same consistent state. It should be noted that if the CRDT crashes (e.g. due to too high load), it could mean other nodes' lease allocations are denied. However, we do not consider this a grave issue.

GenServer By using `GenServer`, we were able to create a system that worked off of asynchronous message passing. The underlying library provided us with a strong foundation to build upon, and is thoroughly tested. We were able to effectively build upon this platform.

Testing We have a comprehensive suite of integration and end-to-end tests. We were able to test the scenarios presented in Section 2.4, and emulate how the system would behave in production in the real world. By defining so-called "crashpoints" we were able to cause crashes to get into the desired state for the test. In many places of the code, we have 'if'-expressions checking if we should crash. This effective testing is also enabled by Elixir being a functional programming language.

Reusable abstractions While the software engineering principles are not a primary concern to us, the reusability of our `CRDT` behavior allowed us to create shared logic between counter and OR-set implementations, such that any further implementation of CRDTs just has to write the business-logic.

Configurability The selected CRDT gets correctly selected at runtime. This and many other configuration parameters can be specified through environment variables. This enabled us to significantly improve testing.

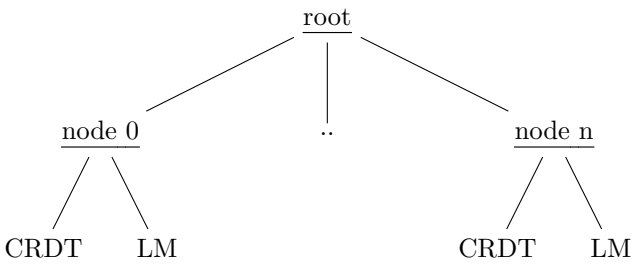

Figure 1: The supervision tree of our system. Underlined nodes are supervisors, non-underlined nodes are `GenServer`s.

3.2 Challenges

There were many challenges that we faced throughout the implementation. Some of the more interesting ones are listed.

Concurrent control flows Although our application is inherently asynchronous, we still have to provide scheduling in some form or another. For example: when a new lease for p is being decided, and q requires a new lease too, we cannot perform this concurrently. We need to queue up q's request. Scheduling with a waiting queue and complex control flow logic. This was all difficult to conceptualize and to implement, in particular, because `GenServer` did not provide a suitable abstraction for this.

Crash recovery Crashing is straightforward, recovering less so. In particular coordination, in the form of lease management, required a number of states to be considered. It was of great aid that we had a very thorough understanding of scenarios that can arise during execution. Nonetheless, the intricacies of the many possible states a process can find itself in, when crash recovering, and the difficulty of ensuring that the recovered state is consistent both challenged us.

Testing Although we mention testing as a highlight, it was also a major challenge. Especially due to concurrency, testing and bugfixing was very thought-provoking and took up a large portion of the actual implementation time. Fortunately, the Elixir/Mix ecosystem provides a very pleasant testing experience that made testing as easy as possible.

4 Evaluation

When evaluating the implementation of our system, our focus is twofold – first ensuring correctness, secondly its performance. As is the theme of this report, this is done

separately for each sub-system.

4.1 Conceptual Correctness

4.1.1 Lease Management

Using Bernstein, Hadzilacos, and Goodman [1], we established the following correctness properties of the lease management process as follows.

- *AC1:* All processes that reach a decision reach the same one.
- *AC2:* A process cannot reverse its decision after it has reached one.
- *AC3:* The Commit decision can only be reached if *all* processes voted Commit.
- *AC4:* If there are no failures and all processes voted Commit, then the decision will be to Commit.
- *AC5:* Consider any execution containing only failures that the algorithm is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.

Properties *AC3* and *AC4* are straightforward and are satisfied as per the Nominal scenario in Section 2.4. We ensure individual decisions cannot be reversed (*AC2*) by storing them on persistent storage using a write-ahead log. *AC1* is guaranteed thanks to the aforementioned reasons and by ensuring only one transaction is being decided at a time.

Bernstein, Hadzilacos, and Goodman [1] argue that properties *AC1*-*AC4* are satisfied inherently by 2PC, but *AC5* is not and lists two reasons. Firstly, arbitrary messages may not arrive. Secondly, if a process fails and recovers, it must recover its previous state from stable storage and contact the coordinator in order to get up to date.

The first reason is not applicable due to our assumptions in Section 2.2, as we assume reliable broadcast (with FIFO perfect links) and temporary network partitions. Secondly, due to the persistently-stored log and the recovery mode mentioned in 2.4, we always make sure a recovered process has an equivalent state to the one it crashed with and it reaches a consistent state with the other followers, as previously described. Thus, we consider the implementation of our lease management process to be correct.

Proof of lease correctness We have formalized the correctness of our lease management protocol, but we must still prove no CRDT will allocate more leases than it has (i.e. is correct), given this lease management system. Proof 4.1.1 proves this for unpartitioned nodes, and Proof 4.1.1 for partitioned nodes.

Lemma 1. *Given that there are no network partitions or node crashes, the CRDTs will remain consistent.*

Proof. If there are no network partitions, and no processes have crashed, we are able to complete the 2PC protocol eventually. Therefore, either everyone commits or aborts, and thus the view of leases that all CRDTs have is the same. □

Lemma 2. *Given that there are network partitions, and no processes have crashed, the CRDTs will remain consistent.*

Proof. Assume that our nodes are bi-partitioned. Consider one partition, P, which contains a single node, and the other partition R all other nodes including the coordinator. The number of nodes in P is irrelevant, but a single node helps illustrate more coherently. We divide into the following cases.

Case 1 P does not receive a Prepare. In the case of an allocation, P's lease allocations may be at worst more conservative than R (it believes less leases are given than in reality). The same happens with a deallocation request for another node. If it is a deallocation of leases from P, we may choose to reject it later on if in the meantime we used up more leases than we have available (as is in-line with our best-effort deallocation assumption). Hypothetically, in a later transaction, P could then erroneously commit to an over-allocation of leases. However, since we have source-FIFO links, any later Prepare will arrive after this missed Prepare, thus we will be in a consistent state.

Case 2 P does not receive a Commit. Since we commit, all nodes must have unanimously agreed, and the changes are already optimistically reflected in each of the write-ahead logs. Thus, we will still be in a consistent state.

Case 3 P does not receive an Abort to an allocation. Allocation decisions are unanimous in our system as all nodes know all leases and these leases are all kept consistent. In this case, we can be certain P has aborted this transaction as well, and the change has not been applied. Thus, there is no consistency violation.

Case 4 P does not receive an Abort to revocation of leases of another process than itself. In this case, as with Case 1, we could allocate more leases in the next transaction. However, due to FIFO delivery, we can be sure that the abort is reflected beforehand, and we retain a consistent state.

Case 5 P does not receive an Abort to revocation of leases of itself. In this case, P's leases are more conservative than in reality, and we cannot over-allocate leases, thereby retaining lease consistency. We will again correct this before the next Prepare due to FIFO delivery.

We have shown that under all possible scenarios, P remains in a consistent state. R will be consistent, as this partition contains the coordinator and all channels are correct. Therefore, our lease management remains consistent. □

Please note that we do not consider node failures in these proofs. As demonstrated by the scenarios in Section 2.4, single-follower or single-coordinator failures can recover. Hence, these proofs could be augmented to cover these situations.

4.1.2 CRDT

We define the properties of a CRDT to be as follows:

1. *CRDT1:* Updates can be initiated at any replica.
2. *CRDT2:* If two correct processes receive the exact same set of updates, then their state is equivalent.
3. *CRDT3:* Any two updates are distributed events. These can be:
   (a) Causally dependent updates, in which case one is encapsulated in the state of another due to the monotonicity of CRDTs.
   (b) Concurrent updates, in which case there will be a commutative join-semilattice.

By our distributed design, *CRDT1* and *CRDT2* hold true. This is further verified by test cases. Furthermore, since our CRDTs are monotonic and grow-only, we fulfill *CRDT3*. Consequently, we exhibit strong eventual consistency.

4.2 Implementation Correctness

As mentioned in Section 3, we employ many end-to-end and integration tests (both good-weather and bad-weather) in order to verify correctness. Elixir tests the Erlang process scheduler with random seeds, so we can be confident that the most prominent race conditions are detected. We employ a total of **46** test cases (most of which based on the scenarios discussed in Section 2.4), which are summarized as follows:

1. Config tests
2. PN-Counter logic, such as incrementing, decrementing, summing
3. PN-Counter synchronization
4. PN-Counter responses with manual and automatic lease management
5. OR-Set logic, such as adding, removing and existance checks
6. OR-Set synchronization
7. OR-Set responses with manual and automatic lease management
8. TPC commits and aborts with lease allocations
9. TPC commits and aborts with lease deallocations
10. TPC follower crashes during preparation phase
11. TPC follower crashes after commit/abort decisions
12. TPC follower crashes and immediate next coordination
13. Coordinator crashes during preparation phase
14. Coordinator crashes during commit/abort decisions

4.3 Performance

Regular CRDTs have the advantage that they require no coordination, and can therefore be heavily parallelized. This is strongly advantageous in load-balancing situations. Introducing frequent coordination diminishes such benefits, and this Section explores the real-world impact of lease allocation in scenarios where no nodes crash or partition.

4.3.1 Parameters

In these tests, we consider systems with 3, 5, and 10 CRDT nodes. For the benchmark, we chose the counter to emulate ticket sales. Each CRDT has 100 "client" processes that concurrently perform operations on it. Each client performs 10 sequential operations (in the counter example, this is an increment operation, i.e. a ticket sale). We consider the initial allocation and automatic re-allocation of leases to be $n \in \{1, 10, 100, 500, 1000\}$. It should be noted that $n = 1$ is equivalent to performing coordination for every CRDT operation, i.e. a fully coordinated system. We consider the number of stock to be sufficiently high such that we do not run out of stock throughout the benchmark.

4.3.2 Procedure

To benchmark, we run every trial in isolation. Before each trial, the system starts up and sets the correct parameters. That way, the system is in a ready state by the time we initiate trials.

A trial starts by creating a timer process, which saves the start time. Then, all the clients are spawned and the trial commences. After each client finishes its requests, it notifies the timer. Alongside this notification, it reports the number of lease violation errors it encounters. Once the timer has received notifications from all clients, it saves the end time. The total time taken is the difference between these times in millisecond granularity. Therefore, even with clock drift, we can still achieve millisecond precision.

4.3.3 Experimental Setup

These benchmarks were performed on a single machine (Apple MacBook Pro 2023, M2 Pro aarch64 CPU, 16GB RAM). Since this is not a distributed environment, we emulate latency in the coordination phase with a random arbitrary delay of $x \in [1, 10]$ ms to compensate for this. We ensure to run tests with the same seed, such that this delay is deterministic across trials. We repeated every trial thrice to account for inaccuracies.

4.3.4 Results and Discussion

The results can be seen in Figure 2 (note the logarithmic Y axis) and Table 1. As expected, more coordination causes more contention which affects the parallelism of the CRDTs. Consequently, the runtime increases. As lease increments increase, CRDTs are able to spend more time running in parallel and coordinate less, such that we can harvest parallelization benefits.

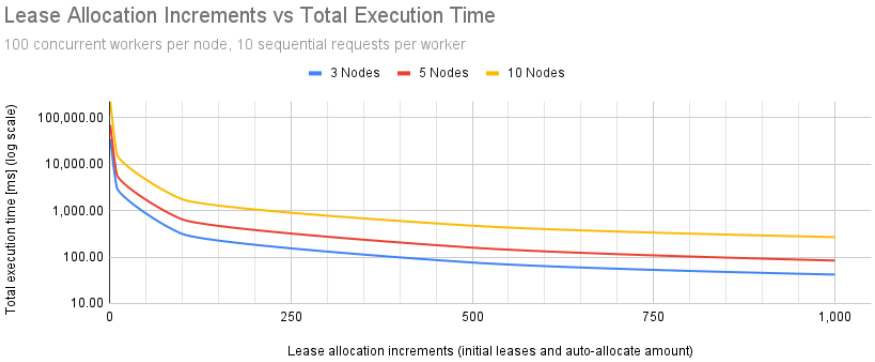

Figure 2: Benchmarking graph for 100 clients at 10 iterations each.

# Leases	3 Nodes	5 Nodes	10 Nodes
1	$34,684.16 \pm 1076.23$	$69,805.96 \pm 75.07$	$220,542.16 \pm 8725.20$
10	$3,096.09 \pm 19.90$	$5,893.12 \pm 56.36$	$16,264.91 \pm 116.57$
100	314.69 ± 22.84	646.35 ± 4.67	$1,750.54 \pm 56.04$
500	76.16 ± 1.07	159.17 ± 9.04	472.07 ± 21.38
1000	41.92 ± 8.01	83.96 ± 5.09	268.25 ± 9.66

Table 1: Benchmarking results in milliseconds for 100 clients at 10 iterations each.

It should be noted that with small lease increments, requests can and do come in faster than coordination can happen. Therefore, assuming a constant stream of requests, the system will eventually be unable to keep up, and the times between request invocation and request response at the CRDT level will perpetually increase.

In Figure 3, we highlight the scalability explicitly. With more nodes, communication time increases. We require more messages to be sent, we must wait for more responses, and coordination happens more frequently (as the absolute number of coordination phases increases). This is linear, and the trend is exemplified in the figure.

It should be noted that this scalability is not entirely accurate. Processes need to run on a thread, and we cannot attain true parallelism of nodes $\times$ clients $\times$ 100 threads as we are bound by the hardware requirements of a single machine (in our case, 12 cores and

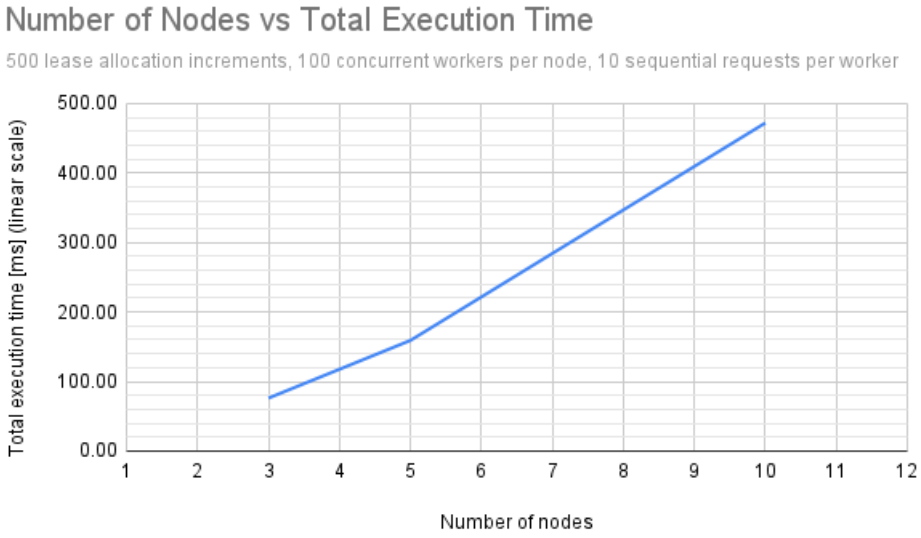

Figure 3: Effect of a different number of nodes on total runtime for 100 clients at 10 iterations each, using 500 lease allocation increments

12 threads). Despite this, we believe that these result in a sufficiently accurate picture of how our system scales.

Lastly, it should be mentioned that throughout benchmarking we did not encounter any lease violation errors. This exemplifies the system's correctness in scheduling coordination phases such that every CRDT request can be served with a positive response (assuming sufficient stock and correct domain behaviour).

5 Future Work

In this section, we discuss potential improvements and areas of future work on the project. Initially, in the established context of an asynchronous system model, but we do not limit ourselves to it and explore possibilities beyond it, namely by assuming a partially synchronous system.

Distribution The current implementation concerns a single-node system for now. This does not represent a "real" distributed system per se, but a local one providing us implicitly with FIFO perfect links and reliable broadcast. So, the first step of improvement is to distribute the system properly. This then simulates geographical distribution, but changes are required to guarantee FIFO perfect links and FIFO reliable broadcast.

Benchmarking with failures and partitions Currently, we benchmark only under the assumption of no node crashes or partitions. Building upon the improvement of distributing this system, it would be interesting to explore how the presence of these faults impacts performance, in particular timings.

Lease Management: 3PC We identify the following improvements to coordination. Firstly, as described in Section 2.4.4, there is a potential scenario (both the coordinator

and at least follower crash) that could introduce inconsistency across replicas. We propose to resolve this by changing our underlying coordination from 2PC to 3PC.

Lease Management: Abort retries Currently, there are scenarios where aborts are issue in the case of crashes. These are not differentiated by the application (CRDT) from aborts due to constraint violations, so they are not retried. For a production-ready system, a retry mechanism should be implemented.

Lease Management: Better effort lease revocations When a node is asked to deallocate leases and it does not have the capacity to do so, it Aborts the request. It could be investigated if there are smarter approaches to "undoing" some lease allocations. For instance by deallocating the same sum of leases, but across multiple nodes rather than a single one. Or in the case of an online retailer, cancelling certain orders in order to satisfy such a request.

Lease Management: Domain-specific lease allocation Coordination is expensive. In production-grade systems, the lease management and allocation should be fine-tuned for its specific use-case. For instance, in a scenario where individual replicas are situated geographically around the world, we can increase performance by auto-allocating more leases to regions with a large amount of traffic, whereas those in e.g. early morning hours with less traffic are expected to use significantly fewer leases. This is of course dependent on the problem that is being solved. Nonetheless, it offers a compromise between less coordination times and lease over-allocation. Arguably, allocating an optimal amount of leases to a replica could neglect the negative effects of (partial) failures, such as network partitions.

Lease Management: Domain-specific lease revocation To ensure that leases are being allocated as effectively as possible, systems could revoke leases guided by domain-specific heuristics. For example, in line with the previous improvement, during night-time in Japan, de-allocate many left over leases to allocate them to the awaking Americas.

CRDT: Extending the OR-Set with quantity Currently, we only store if an item is in the shopping cart or not. In the future, we could extend the functionality to keep track of the number of times each item is in the shopping cart, by extending the item of our OR-Set element tuple. Decreasing item's quantity would be then represented by two consecutive CRDT operations – remove and add, adding initial quantity – the amount decreased by.

CRDT: Stock and leases per item in OR-Set Currently, all items (I) in the OR-set have the same stock and get given the same leases. This is not a realistic assumption for an online retailer, and thus an area of further improvement.

CRDT: Computational efficiency in the OR-Set Currently, lease availability checking algorithm runs in $\mathcal{O}(n)$. This could be optimized to run in constant time.

CRDT: Network efficiency Efficiency of synchronization between CRDT replicas could be increased, by using operation-based CRDTs (CmRDTs) instead of state-based ones, that keep track of increasingly large configuration space. CmRDTs would optimize this, but at the cost of having to keep track of causality, using for instance vector clocks.

5.1 Assuming a Partially Synchronous System

Up until this point, we have considered a purely asynchronous system model. This limits us in certain ways, as it is for instance impossible to implement failure detection [3]. Thus, this section will briefly consider possible improvements, if we assume a partially synchronous system model. That is a system that becomes synchronous in periods of time, long enough for the algorithm to terminate its execution [3]. This assumption brings new possibilities and directions for improvement in terms of performance. Such as a possibility of eventual failure detection.

Assuming a coordinator crashes, this could be partially solved by letting a follower complete the ongoing transaction (fail-over), as coordinator state can be replicated to every follower. This would however not allow a start of another transaction until the coordinator recovers (which is does not have a significent effect if we pre-allocate sufficient leases). Ideally, we would like to elect a new coordinator if the designated one has failed, but as we require the coordinator to vote as well, the previous coordinator will still be required to come back to make progress.

Lastly, an area of improvement could be re-configuration. This can be done by using protocols such as OmniPaxos [5] (that rely on the partially synchronous model). We have assumed the number of CRDT replicas to be static, but it would be of great practical use to be able to add or remove them, in order to scale up and down respectively. In other words, to ensure elasticity.

6 Summary

We presented a system for the asynchronous message-driven crash-recovery model that in a hybrid manner combines PN-Counter and OR-Set CvRDTs with coordinated lease allocation. This adds support to CRDTs for non-monotonic operations, such as limited stock in our case. Under sufficient pre-allocated leases, this system is eventually strongly consistent, highly available, and partition tolerant. When coordination is required, we sacrifice availability at some CRDTs to guarantee atomic consistency. We utilize Two-Phase Commit and show that this design is resilient to node crashes (and remains consistent), unless during lease coordination, both a coordinator and follower crash.

We implemented this system in the Elixir programming language on top of the Erlang/OTP ecosystem, an ideal choice considering our environment and assumptions, while showing its correctness both formally and by employing tests in our code. Benchmarking this system in a real-world scenario, we show that we attain substantial performance benefits by pre-allocating replicas leases as opposed to coordinating every step of the way, while still not exceeding stock limitations.

Finally, we present possible areas of improvements. These include porting our code to distributed nodes, transitioning from 2PC to 3PC to be more failure-resistant, improving computational efficiency, and using domain-specific heuristics to guide lease allocations to optimize for the perfect performance while avoiding over-allocation ratio.

7 References

References

[1] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. "Concurrency Control and Recovery in Database Systems". In: 1987. URL: https://api.semanticscholar.org/CorpusID:59855432.

[2] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

[3] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer Science & Business Media, May 2006, pp. 43–47. ISBN: 9783540288466.

[4] Andreas Henriksson and Svante Bennhage. "Trading performance for precision in a CRDT-based rate-limiting system". English. MA thesis. Chalmers University of Technology, 2021. URL: https://hdl.handle.net/20.500.12380/304211.

[5] Harald Ng, Seif Haridi, and Paris Carbone. "Omni-Paxos: Breaking the Barriers of Partial Connectivity". In: *Eighteenth European Conference on Computer Systems (EuroSys '23)*. Rome, Italy: ACM, May 2023, p. 17. DOI: 10.1145/3552326.3587441. URL: https://doi.org/10.1145/3552326.3587441.

[6] Marc Shapiro et al. "Conflict-free replicated data types". In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, pp. 386–400. ISBN: 9783642245497.