From Causality to Stability: Understanding and Reducing Meta-Data in CRDTs

Jim Bauwens
Software Languages Lab
Vrije Universiteit Brussel
Belgium
jim.bauwens@vub.be

Elisa Gonzalez Boix
Software Languages Lab
Vrije Universiteit Brussel
Belgium
egonzale@vub.be

ABSTRACT

Modern distributed applications increasingly replicate data to guarantee both high availability of systems and optimal user experience. Conflict-Free Replicated Data Types (CRDTs) are a family of data types specially designed for highly available systems that guarantee some form of eventual consistency. To ensure state convergence between replicas, CRDT implementations need to keep track of additional meta-data. This is not a scalable strategy, as a growing amount of meta-data has to be kept.

In this paper, we show that existing solutions for this problem miss optimisation opportunities and may lead to less reactive CRDTs. For this, we analyse the relation between meta-data and the causality of operations in operation-based CRDTs. We explore a new optimisation strategy for pure operation-based CRDTs and show how it reduces memory overhead. Our approach takes advantage of the communication layer providing reliable delivery to determine causal stability, and as a result, meta-data can be removed sooner. We furthermore propose a solution for improving the reactivity of CRDTs built on a reliable causal broadcasting layer.

We apply our strategy to pure-operation based CRDTs and validate our approach by measuring its impact on several different set-ups. The results show how our approach can lead to significant improvements in meta-data cleanup when compared to state-of-the-art techniques.

CCS CONCEPTS

• **Software and its engineering** → **Garbage collection**; **Synchronization**; **Consistency**; **Distributed architectures**.

KEYWORDS

Replication, CRDTs, Memory management

ACM Reference Format:
Jim Bauwens and Elisa Gonzalez Boix. 2020. From Causality to Stability: Understanding and Reducing Meta-Data in CRDTs. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes (MPLR '20), November 4–6, 2020, Virtual, UK*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3426182.3426183

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MPLR '20, November 4–6, 2020, Virtual, UK

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8853-5/20/11…$15.00
https://doi.org/10.1145/3426182.3426183

1 INTRODUCTION

To ease the development of geo-distributed applications, much research has studied the concept of replicated data types which expose to programmers an interface akin to that of a sequential data type while implementing mechanisms to keep data consistent across replicas [7, 12, 16]. Conflict-Free Replicated Data Types [14–16] (CRDTs) are replicated data structures that can be concurrently updated without requiring synchronization among replicas to deal with conflicts. In particular, CRDTs are said to be strong eventually consistent (SEC), which denotes that if two components of a system have received the same updates, they will be in the same state. This implies that assuming no new updates happen to a set of replicas, they will eventually converge to the same state without conflicts.

Designing new data types that adhere to this property is a complex task. In the most simple form, one can construct a CRDT where all operations are commutative. This means that regardless of the ordering in which a set of operations is applied, the resulting state will be equivalent. Another common approach is to use causal ordering for non-concurrent operations and only have concurrent operations be commutative [2, 11, 14].

To handle concurrent operations and ensure commutativity, a CRDT implementation typically keeps some meta-data. For example, some set CRDTs might use tombstones — placeholders for removed entries — to ensure that removal operations are commutative [16]. If a replica receives a removal operation for an item before it received the actual operation that added it, then tombstone will ensure that the removal will still be processed after the add.

Alternatively, many CRDT implementations rely on Reliable Causal Broadcasting [6] (RCB), used to ensure both causal ordering and reliable delivery [3, 16]. In fact, RCB is commonly understood to be a requirement for operation-based CRDTs [2, 3, 14, 16].

Regardless of the approach, some causality meta-data will always be stored in some form. For many CRDT types, this meta-data grows unboundedly, leading to bad performance and scalability.

Pure operation-based CRDTs [3], a framework for building operation-based CRDTs, aims to tackle this problem by removing meta-data for causally stable operations (i.e. operations for which no new concurrent operations can occur). The framework relies on a messaging middleware utilizing RCB for keeping a partially ordered log of operations, according to the happened-before causality relation[13]. By tracking the causality information between replicas it is possible to determine if operations are causally stable and if, consequently, their meta-data can be removed.

While this approach ensures causal ordering and allows meta-data to be removed, it is not a complete solution and suffers from some issues. First of all to determine causal stability from causal

information alone all replicas need to regularly push out updates. This means that if a single replica in a system does not perform any operation, no meta-data can be removed. Secondly, causal ordering is not always desirable, as it may hamper the reactivity of an operation-based CRDT. Since the happened-before relation does not always imply an actual dependency between operations, operations may be buffered by the RCB middleware needlessly. The result of this is a less responsive CRDT, where replicas might have to wait for updates of multiple other replicas before they can apply already received updates.

In this paper, we explore techniques for overcoming the described limitations and detail our strategy for addressing them in the pure operations-based framework. We first analyse the relationship between meta-data and the causality of operations for operation-based CRDTs and look into the meta-data removal approach used by the pure operation-based CRDT framework. We then explore a new optimisation strategy for the framework that takes advantage of the communication layer providing reliable delivery and show how meta-data can be removed sooner, improving on the existing approach. We furthermore propose a solution for improving the reactivity of CRDTs built on a reliable causal broadcasting layer, by reifying part of the RCBs message buffer.

We apply our new optimisation strategy to pure-operation based CRDTs and validate our approach by measuring the log size of set CRDTs in different set-ups. The results show how our approach can lead to significant improvements on meta-data cleanup when compared to state-of-the-art techniques while providing developers with methods to optimize the tradeoff between operation processing and meta-data cleanup.

2 META-DATA IN CRDTS AND ITS RELATION TO CAUSALITY

This section delves into the details of the described limitations and analyses them in the context of current implementation strategies. We furthermore provide some background information on CRDTs, meta-data usage and causal stability.

2.1 Background

The core idea of CRDTs is to provide a replicated data type which exhibits an API similar to that of a sequential data type while guaranteeing *eventual state convergence* under concurrent operations. This means that eventually the state of two replicas should become equivalent when they have received the same operations, regardless of the order in which they arrived.

There are two main and complementary CRDT designs: state-based and operation-based. State-based approaches propagate local updates by transferring their entire state (or deltas of the state) to other replicas that will merge them with their state. This approach may incur a large network overhead and requires complex state design and merge functionality. Operation-based designs, on the other hand, propagate updates at the operation level and do not need special merging logic. This allows for a simpler implementation and state design while limiting network overhead. Operation-based designs do however require reliable delivery[6, 17] of all updates, which is not required for state-based designs. In this paper, we focus on operation-based approaches and their means of implementation.

2.2 Current Implementation Strategies

Operation-based CRDTs can be implemented in various ways using different techniques. In this section, we detail the general idea used by state-of-the-art implementations and what methods they employ for limiting meta-data built-up.

The simplest strategy for implementing an operation-based CRDT is to make all possible operations commutative, ensuring that, regardless of the order of operations, an equal state will eventually be reached. However, the operations that can be applied to most common data types mostly do not commute. For example, a set data type traditionally supports `add` and `remove` operations. These operations do not commute, as can be demonstrated by simply comparing the two different possible orderings of an add and remove applied to an empty set:

(1) `set <- add(X)`
(2) `set <- remove(X)`

This will result in a set without the item X. The following will result in a set with item X:

(1) `set <- remove(X)`
(2) `set <- add(X)`

To keep the sequential data type API for a data structure, a CRDT implementation will typically modify the semantics of the data type so that it can commute. In our set example, the Observed-Removed Set (OR-Set) CRDT[16] solves this by generating a unique identifier per newly added item. From the user's point of view, the client code interacts with the OR-Set using add and remove operations. But, at the implementation level, an item in an OR-Set can have multiple unique identifiers associated with it, stored along as meta-data. When an item is removed, replicas will instruct other replicas to only remove the items with the identifiers that they have observed before. If the replica receiving the remove has not yet observed a certain identifier, it will keep track of the removed identifier as a *tombstone* and effectively delay the operation until after the add is received. This ensures that in the case of a concurrent add and remove operations, the add will always be ordered before the remove, resulting in a commutative data type.

The unique identifiers in OR-Sets have essentially two purposes: 1) encoding the *happened-before* relation of operations[13]; e.g. remove operations that do not include a certain identifier must have happened before adds with that identifier, and 2) providing *add-wins* semantics for concurrent operations, i.e. that concurrent adds will win over concurrent removes. Similarly, many other CRDTs (e.g. MV-Registers, U-Sets, RGAs, ...) use unique identifiers and meta-data to guarantee that operations commute[5, 11, 14, 16]. However, this approach leads to complicated implementations as the code responsible for operation logic handles many different concerns[2]. The meta-data also leads to a memory management problem as information — such as tombstones — has to be kept to ensure correct concurrent behaviour[3, 5, 15].

To simplify some of this complexity, some CRDT implementations rely on Reliable Causal Broadcasting [6] (RCB), which will ensure causal ordering for non-concurrent operations (along with reliable delivery)[2, 16]. Baquero et al. extend on this and rely on existing causality information stored within the RCB middleware instead of manually encoding causality information as meta-data to operations. To this end, they introduced the Pure Operation-Based

framework embodying such an implementation strategy [3]. The framework employs a partially ordered log of operations (shortened to PO-Log) constructed with the causality information of the underlying RCB communication layer. The state of the data structure can be computed by observing this log, and with simple rules, the log can be compacted whenever operations become redundant.

While the framework simplifies CRDT design, it does not tackle the growing meta-data problem. For this, an extension was proposed that uses causality information from the RCB layer to determine what operations are *causally stable*. Operations are causally stable if they have been applied by all replicas, i.e. no concurrent operation can occur anymore. The implication of this is that causal meta-data can be stripped away from log entries when an operation becomes causally stable and that the remaining operations can be stored in a sequential data structure. This reduces memory overhead and limits the computational complexity as a smaller log means that less work is needed to construct the state of a replica.

2.3 Problem Statement

There are, however, some limitations to pure op-based CRDT due to the approach used to determine causal stability and its reliance on reliable causal broadcasting. In this section, we further analyse meta-data and its relation to causality and identify two problems which motivate our work.

2.3.1 Too Conservative Approach to Causal Stability. As mentioned previously, pure operation-based CRDTs will strip causal information from operations whenever the framework determines that they are causally stable. Baquero et al. utilize an underlying Reliable Causal Broadcast (RCB) middleware to track the causality of operations that are propagated through a system. They define causal stability as follows:

> A timestamp τ, and a corresponding message, is causally stable at node i when all messages subsequently delivered at i will have timestamp $t > \tau$. (Baquero et al., 2017; Note: *node* is equivalent to *replica* in our text)

This means that if a replica has received an operation o with a (logical) timestamp t for source replica n, and subsequently it receives operations from all other replicas where the timestamp for replica n is larger than t, o is said to be causally stable as every other replica must have observed it.

Consequently, with this approach, it is only possible to determine causal stability for an operation if and *only if* every other replica in the system sends new updates following the operation, to collect enough causal information. In other words, if one single node does not issue any updates for some time, no causal stability can be determined at any replica during that period. In the evaluation section of this paper, we perform several tests to further illustrate this problem, clearly showing that the approach is not eager enough as it may be problematic in terms of memory usage.

2.3.2 Lowered Reactivity Through Causal Buffering. The second issue we identify with current approaches is rooted in the reliance on an RCB middleware for communication. An RCB layer orders causally related messages according to Lamport's happened-before

Table 1: A sequence of operations applied to several replicas implementing an add-wins pure operation-based set CRDT.

SET A	SET B	SET C	Operation
{}	{}	{}	SET C :: Add (X) SET C :: Add (Y)
{X,Y}	{X,Y}	{X,Y}	SET B :: Add(Z)
{X,Y}	{X,Y,Z}	{X,Y,Z}	SET C :: Remove(X)
{X,Y}	{Y,Z}	{Y,Z}	

Table 2: A sequence of operations applied to several replicas implementing a classic OR-Set CRDT.

SET A	SET B	SET C	Operation
{}	{}	{}	SET C :: Add (X) SET C :: Add (Y)
{X,Y}	{X,Y}	{X,Y}	SET B :: Add(Z)
{X,Y}	{X,Y,Z}	{X,Y,Z}	SET C :: Remove(X)
{Y}	{Y,Z}	{Y,Z}	

relation. This means it may buffer a message if some causal predecessors have not yet been received. However, that buffering will delay operations that may not be dependent on each other, resulting in a less reactive CRDT. This means that CRDTs may not immediately reflect the updates that they have received, even if there is no valid reason for doing so. In the case of pure op-based CRDTs, this may also impact the removal of redundant log entries, leading to higher memory consumption.

To demonstrate the implications of a less reactive CRDT, consider a sequence of operations that are applied to three replicas A, B, and C hosting a set CRDT. Figure 1 visualises the connectivity between the replicas. All replicas are connected, except for the link between A and B which has a temporal failure, i.e. updates are not propagated between replicas A and B.

We employ two different implementations of a set CRDT with add-wins semantics: table 1 shows the operations applied on a pure op-based Add-Wins set, while table 2 shows the operations applied on an OR-Set CRDT. Contrary to the pure op-based Add-Wins set, the OR-Set CRDT does require an RCB middleware. At first two operations are applied to replica C. This update is propagated to all other sets and their state will be updated. Following this, an additional update is applied to set B. This update is only sent to set C, as there is a disconnection between set A and B. Now set C applies a remove of item X, which will be observed by set A and B as set C is connected to both other replicas. In the case of the OR-Set, the item will be immediately removed on both set A and B. This differs however for the Add-Wins set, where the operation will not be applied. The reasoning behind this is simple: the Add-Wins set relies on the RCB middleware, and that layer will buffer the operation as it can observe in the causality information that it received along with the operation (from set C) that A has not yet received one or more operations from B. In practice, this means that only after the connectivity issue between A and B is resolved, and set A receives and applies the `Add(Z)` operation from B, the remove operation from set C will be applied.

Not only was the OR-Set able to apply the operation immediately because there was no buffer holding it back, but also because it only encodes causal dependencies for operations that can effectively be dependant on each other, e.g. operations on the same set items.

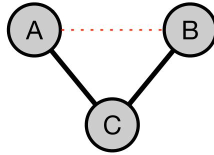

Figure 1: Network connectivity between set replicas

2.3.3 Summary. In short, we identified two issues applicable to all operation-based approaches that rely on RCB and comparison of clocks for the determination of causal stability:

(1) a single replica not performing updates prevents the removal of meta-data since no causal stability can be determined. (2) the reliable causal broadcasting layer will buffer operations while it waits for their causal predecessors, which may lead to less reactive CRDTs and higher memory overhead.

3 EAGER STABILITY DETERMINATION

In this section, we propose an extension for pure operation-based CRDTs that improves on its meta-data removal capabilities by taking advantage of reliable delivery. Classic pure-operation based CRDTs rely on the causality information provided by the RCB middleware to determine causal stability. However, an RCB middleware does not only ensure causal delivery but also reliable delivery, i.e. messages sent through RCB have to be acknowledged. In this work, we propose to take advantage of reliable delivery to eagerly determine causal stability, minimising memory consumption.

When an operation is applied on a CRDT, the underlying replication mechanism will ensure that it is broadcasted to all other replicas. Reliable delivery requires all receiving replicas to acknowledge the reception of the operation, as shown in figure 2. In our approach we take advantage of this design: if acknowledgements have been received from all replicas, it follows that no new operations can be concurrent to it and as a result, the operation is causally stable. At this point, however, only the replica that issued the operation is aware that the operation is causally stable. We use an additional mechanism to broadcast this information to the other replicas (as can be seen in figure 3).

This strategy improves on the pure-operation based approach, allowing causal stability to be determined even when some replicas may not issue updates. As a result, our approach is more suitable when memory resources are scarce, but it introduces a network overhead as stability messages have to be propagated as well. To allow for a flexible trade-off between memory consumption and network overhead, developers can specify intervals in which stability messages are sent. Between the intervals, our approach will simply rely on the causality information of the middleware to deduce causal stability, reducing the number of missed opportunities for removing meta-data.

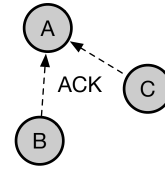

Figure 2: Acknowledgements used by the RCB layer to ensure reliable delivery

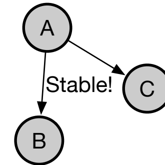

Figure 3: Letting other replicas know that an operation is stable

3.1 An Operation-Based Framework With Eager Stability Determination

In this section, we describe an operation-based framework which has support for causal stability determination using our proposed strategy. We show how its interface allows developers to directly interact with the middleware to implement CRDTs. We then detail how the framework implements the mechanisms for causal stability determination.

The core replication functionality of the framework is built with RCB, as we want both reliable delivery and causal ordering. Causal ordering is used to simplify the design of CRDT implementations, while reliable delivery ensures correctness. We piggyback on these properties in our framework for the eager stability determination approach. Note that in our problem statement we mentioned that causal ordering may lead to less reactive CRDTs. The fact that we rely on it for the framework does not contradict with this, as we will later show in section 5.

The framework provides abstractions that allow developers to propagate and receive operations while abstracting away the details of RCB and stability determination mechanisms. Below we present its interface, which allows the implementation of memory-efficient CRDTs. We employ a class-based language for the description of the interface, along with pseudo-code that describes the logic for some of the essential functionality. In section 4 we present a concrete implementation in TypeScript that takes advantage of this interface. The class should provide the following abstract methods:

- `onOperation(clock, op, args)`: invoked whenever a new update arrives. Will be invoked in causal order.
- `gcStable()`: invoked after a new operation is applied and when a CRDT implementation may want to remove causally stable operations.
- `onLoaded()`: invoked when a new CRDT object finishes initialising.
- `onNewReplica(ref, refs)`: invoked when a new CRDT replica is discovered.

The logic for the replication functionality and stability determination is exposed with the following methods:

- `performOperation(op, args)`: performs an operation on the receiver CRDT object. `op` is an enumerable type and together with the `args` variable it represents the operation that has to be applied. It will cause the `onOperation` method to be invoked first locally and then remotely (by means of message propagation).
- `isCausallyStable(clock)`: can be used to check if a certain clock is causally stable.
- `performPendingStableMsg()`: if the RCB layer has some stability message pending, push them to all replicas. This enables CRDT implementors to encode custom heuristics for causal stability message updates.
- `setStableMsgInterval(interval)`: sets the interval in which stability messages are sent.

The power of this structure is that CRDTs built on top of the API do not need to be aware of the implementation details of the causal stability algorithms and extensions to it.

3.2 Replication Algorithms

We now further detail the replication algorithm employed by our framework based on the aforementioned interface. For the sake of brevity, we only list the essential logic for extending the RCB framework with support for causal stability messages.

Algorithm 1: performOperation

```
Input: an operation o, with arguments args
localClock.increment();
var clock := localClock.copy();
this.doOperation(clock, o, args);
foreach replica in replicas do
  | replica ← doOperation(clock, o, args);
end
when all operations are reliably delivered do
  | this.notifyStable(clock);
end
```

Algorithm 1 is responsible for the application of an operation on a replica. It starts by incrementing the local logical clock of the replica on which it is being applied. Following this, the operation is first applied locally and then propagated to all replicas. The code is extended with a conditional that invokes `notifyStable` (described later) with the clock of the operation when all messages have been reliably delivered. Note that we pass a copy of the original clock, to avoid a race condition where the original clock may have been updated between the delivery of all messages and their acknowledgement.

Algorithm 2 implements the `notifyStable` functionality that handles the stability case after reliable delivery and uses the `setStable` helper function to update all the locally stored replica clocks. To set a particular clock value as stable (meaning, the local clock value for a particular replica), it updates all stored replica clocks with that clock value. It then notifies all remote replicas, if a particular interval (based on the number of operations) is met, that

Algorithm 2: notifyStable

```
Input: a logical clock clock
Data: global pendingStable := False;
Data: global stableCounter := 0;
Data: global stableMsgInterval := 10;
Data: global pendingClock;
var notifyClock := localClock.copy();
notifyClock.setClockAt(clock.getId(), clock.localValue);
this.setStable(clock.getId(), clock.localValue);
pendingStable := ((this.stableCounter++) mod
  stableMsgInterval) != 0 ;
if pendingStable == True then
  | pendingClock := notifyClock;
else
  | performStableMsg(notifyClock);
end
```

the given clock has become stable (using the logic of algorithm 3). If the interval is not met, the clock will be stored and a flag will be set so that it can be applied at a later time. Because concurrent operations may have occurred, the clock for the stability message must be updated with this information. This ensures that the RCB will order the stability messages after any concurrent operation, avoiding that some operations may be compacted before all concurrent operations have arrived, leading to inconsistencies. The replica issuing the stability message will be aware of all concurrent operations as the acknowledgements that are part of reliable delivery will always arrive after any concurrent operation.

Algorithm 3: performStableMsg

```
Input: a logical clock clock
foreach replica in replicas do
  | replica ← doOperation(clock, STABLE, []);
```

The logic for notifying other replicas that an operation is stable is shown in algorithm 3. We use the same delivery mechanism for the stability messages as for normal messages, because we want them to be ordered by the RCB middleware as well. One difference that is not visible in the pseudo-code is that the stability message does not have to be delivered reliable, e.g. we instruct the receiving middleware that no acknowledgement is needed.

Algorithm 4: performPendingStableMsg

```
if pendingStable then
  | performStableMsg(pendingClock);
  | pendingStable := False;
```

Algorithm 4 shows the logic for `performPendingStableMsg` which enables CRDT implementors to force trigger any pending stability messages. In combination with setting a particular interval, this allows developers to use custom heuristics for triggering the stability messages.

4 IMPLEMENTATION

We have implemented our proposed operation-based framework with eager stability determination in Flec [4], an extensible programming framework for CRDTs written in TypeScript. Flec incorporates the concepts of ambient-oriented programming [8, 9], to discover and communicate with replicas in a distributed dynamic network. In ambient-oriented programming, developers are provided with an actor-based programming model where actors can communicate using asynchronous message passing and coordinate through futures [1, 10]. We implement the extended pure operation-based framework for building CRDTs on top of Flec. Flec targets several platforms, such as Node.js[1], and we are currently integrating it with the ESP32[2], a lightweight, power-efficient integrated system-on-a-chip platform. In this section, we first explore the implementation of our framework on top of Flec, and then the implementation of the RW-Set CRDT.

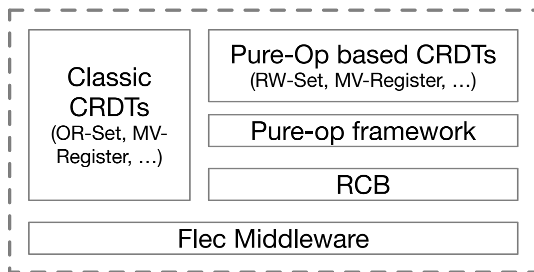

Figure 4: Implementation on top of Flec

Figure 4 depicts the overall architecture of our framework implementation. The RCB layer is implemented directly on top of Flec and brings support for causal delivery and eager stability determination. The pure operation-based CRDT framework builds on this and is used to implement several pure operation-based CRDTs such as the AW-Set, RW-Set, and MV-Register CRDTs. We also implemented several classic CRDTs directly on Flec, which were used for the comparisons in our motivation section.

4.1 An Open Pure Operation-Based Framework

Our pure operation-based framework with eager stability determination is implemented as an abstract class that extends the interface described in Section 3.1, and follows the pure-op design by Baquero et al.. Listing 1 shows the general structure of the class, with the main code redacted.

The class keeps track of several state variables, the most important one being the log. The log is updated when *non-redundant* operations arrive at a replica. To this end, the `onOperation` method is overridden. Listing 2 shows how `onOperation` updates the log. The `onOperation` method is called every time the RCB layer has to deliver an operation, which can originate either from a local or remote `performOperation` invocation.

The `onOperation` method relies on results of the abstract `isRedundantByOperation` and `isRedundantByLog` methods to determine what entries are added or removed from the log. These methods define the redundancy relations for pure-operation based CRDTs and must be implemented by the CRDT implementor. An example of this can be seen in 4.2 where we implement an RW-Set using the framework.

```
1  export abstract class POLog<O> extends CRDT_RCB<O> {
2    // CRDT state
3      log : POLogEntry<O>[] = [];
4      compact = {};
5
6      // CRDT nework
7      network = [];
8      joinNode: FarRef<this>;
9
10     // used to set stability trigger level
11     logCompactSize : number = 100;
12
13     constructor (tag) { ... }
14
15     // Handle replica joins
16     onNewReplica(ref: FarRef<this>, refs) { ... }
17     setupState(state) { ... }
18     getNetwork(){ ... }
19     join(id) { ... }
20     getState() { ... }
21
22     // Handle new operations
23     onOperation(clock: VectorClock, op: O, args: any[]) { ... }
24
25     // Manage cleanup of causally stable entries
26     getConcurrentEntries(entry: POLogEntry<O>) { ... }
27     markStable(){ ... }
28     compactStable(){ ... }
29     gcStable() { ... }
30     cleanup() { ... }
31     setGCParams(logSize: number, intervalSize) { ... }
32
33
34     // Hooks for implementors of pure-op based CRDTs
35     protected setEntryStable( entry: POLogEntry<O> ) : boolean { ... };
36
37     protected removeEntry( entry: POLogEntry<O>) {};
38     protected newOperation(entry: POLogEntry<O>) {};
39
40     protected abstract isRedundantByOperation(e: POLogEntry<O>, entry:
           POLogEntry<O>, isRedundant: boolean) : boolean;
41     protected abstract isRedundantByLog(entry: POLogEntry<O>) : boolean;
42 }
```

Listing 1: Structure of the POLog class, used to implement pure-operation based CRDTs

```
1  onOperation(clock: VectorClock, op: O, args: any[]) {
2      let entry = new POLogEntry<O>(clock, op, args);
3
4      this.newOperation(entry);
5      let isRedundant = this.isRedundantByLog(entry);
6
7      for (let i=this.log.length-1; i>=0; i--) {
8          let e = this.log[i];
9          if (this.isRedundantByOperation(e, entry, isRedundant)) {
10             this.removeEntry( this.log[i] );
11             delete this.log[i];
12         }
13     }
14
15     this.log = this.log.filter(e => typeof e !== "undefined");
16
17     if (!isRedundant) {
18         this.log.push(entry);
19     }
20
21     this.cleanup();
22 }
```

Listing 2: The onOperation method is used to process received operations.

Listing 3 shows how the framework marks and compacts causally stable log entries. The `gcStable` method is invoked by the RCB layer whenever some operations are processed. It ensures that periodically all causally stable log entries are marked as stable (`markStable`) and eventually compacted (`compactStable`). As such, it ensures that an entry will *only* be removed once all concurrent entries are stable as well.

A final aspect of our pure-operation based CRDT implementation is the `cleanup` method, which is invoked at the end of `onOperation` as can be seen in listing 2.

[1]https://nodejs.org
[2]https://www.espressif.com/en/products/socs/esp32

```
markStable(){
    let stableItems = false;

    this.log.forEach(e => {
        if (this.isCausallyStable(e.clock)) {
            e.setStable();
            stableItems = true;
        }
    });

    return stableItems;
}

compactStable(){
    this.log.filter(e => e.isStable && this.getConcurrentEntries(e)
           .map(e=>e.entry.stable)
           .reduce((a,b)=> a && b, true))
           .forEach(e => {
                 if (this.setEntryStable(e))
                     delete this.log[this.log.indexOf(e)];
            });

    this.log = this.log.filter(e => typeof e !== "undefined");
}

gcStable() {
    if (this.markStable())
        this.compactStable();
}
```

Listing 3: The logic used to mark and compact causally stable log entries.

```
cleanup() {
    if (this.log.length === this.logCompactSize)
        this.performPendingStableMsg();
}

setGCParams(logSize: number, intervalSize) {
    this.logCompactSize = logSize;
    this.setStableMsgInterval(intervalSize);

    this.cleanup();
}
```

Listing 4: Methods allowing instrumentation of the cleanup process.

The `cleanup` method checks the size of the log, and if it is higher than a certain limit it will ask the RCB layer to send any pending stability messages (employing `performPendingStableMsg()`).

4.2 Implementing CRDTs on the Framework

We now detail how to implement a CRDT in our pure operation-based framework, using an RW-Set as an example. The implementation is based on the specification for the pure-op RW-Set from Baquero et al., though there are some slight differences with the redundancy relations which we further detail later in the section.

To implement a pure operation-based CRDT the set of operations that can be applied on the CRDT must be defined. Listing 5 defines the operations for the RW-Set using an enumeration with `add`, `remove`, and `clear` values. Then, the PO-Log class of our framework is extended and a constructor ensures that a callback method is registered for state updates. The enum type is used to specify the entries that can be stored by the PO-Log. Following this, the abstract `isRedudantByOperation`, `isRedudantByLog`, `setEntryStable` and `newOperation` methods from the PO-Log class are implemented and extended, providing the behaviour of the RW-Set. Finally, the `toList`, `add`, `remove`, and `clear` methods provide the CRDT with its public interface. We detail the implementation of these methods below.

The redundancy relations for the set are implemented with the abstract `isRedundantByOperation` and `isRedundantByLog` methods from the PO-Log class, as shown in Listing 6 and 7 respectively.

```
enum SetOperation {
    Add, Clear, Remove
}
type SetEntry = POLogEntry<SetOperation>;

export class RWSet extends POLog<SetOperation> {

    constructor(tag, callback) {
        super(tag);
        this.callback = callback;
    }

    ...
}
```

Listing 5: Basic structure of the RW-Set implementation

The relationships between log entries can easily be defined by means of `.is`, `.precedes`, `.follows`, `.isConcurrent` and `.hasSameArgsAs` methods . Contrary to the original RW-Set specification we do immediately remove `add` operations concurrent with `remove` operations. By ensuring that the `clear` operation can never make `remove` operations redundant there is no chance that a `remove` operation is cleared before a concurrent `add` is processed. In the original specification, this was remedied by keeping all concurrent `add` operations in the PO-Log at all times, complicating the CRDT design.

```
isRedundantByOperation(e1: SetEntry, e2: SetEntry, isRedundant: boolean)
       : boolean {
    return (e1.precedes(e2) &&
              ((e1.is(SetOperation.Add) && e2.is(SetOperation.Clear)) ||
                    e1.hasSameArgAs(e2))) ||

           (e1.isConcurrent(e2) &&
                e1.is(SetOperation.Add) && e2.is(SetOperation.Remove) &&
                      e1.hasSameArgAs(e2));
}
```

Listing 6: Logic implementing the RW-Set redundancy relations for operations stored in the log

```
isRedundantByLog(entry : SetEntry ) {
    return entry.is(SetOperation.Clear) || (entry.is(SetOperation.Add) &&
       !!this.log.find(e => e.is(SetOperation.Remove) &&
             e.hasSameArgAs(entry) &&
             e.isConcurrent(entry)));
}
```

Listing 7: Logic implementing the RW-Set redundancy relations for new operations

Listing 8 shows how to remove entries that are causally stable from the log for the RW-Set. To this end, the `setEntryStable` method can be overridden. In the case of the RW-Set, all stable entries corresponding to add operation will be applied to a sequential set (implemented by using an object as a dictionary). The return value of the `setEntryStable` method determines if the entry is removed from the log or not, which is useful for the implementation of more complex stability logic.

```
setEntryStable(entry : SetEntry) : boolean {
    let element;

    if (entry.is(SetOperation.Add)) {
        const element = entry.args[0];
        this.compact[element] = true;
     }

    return true;
}
```

Listing 8: When an add operation becomes stable, the operation is applied to a sequential set

To update the compacted state whenever a new operation is applied, the `newOperation` method can be overriden. For the RW-Set we remove any item from the set if an operation is received

that affects that item, as shown in listing 9. The clear operation will simply clear the entire set. This behaviour is similar to the behaviour of `isRedundantByOperation` (every entry in the compacted state can be seen an entry that always precedes new ones), but instead of being applied to the log, it will be applied to the compacted state.

```
newOperation(entry: SetEntry) {
    if (!entry.is(SetOperation.Clear)) {
        let element = entry.args[0];
        delete this.compact[element];
    } else {
        this.compact = {};
    }
}
```

Listing 9: Cleanup of the compacted state whenever new operations arrive

Listing 10 shows the implementation of `toList`, which evaluates the log and compacted state and constructs the full state of the RW-Set. Because the redundancy methods already filter out adds concurrent to removes we can simply take all the add operations and construct the state from that.

```
toList() {
    let list = {...this.compact};

    this.log.forEach(entry => {
        if (entry.operation == SetOperation.Add)
            list[entry.args[0]] = true;
    });

    return Object.keys(list);
}
```

Listing 10: When an add operation becomes stable, the operation is applied to a sequential set

Finally listing 11 implements the mutator methods for the set. All they do is simply signal the RCB layer using `performOperation` that a particular operation has applied and the RCB layer will propagate and apply it on all other replicas in the system.

```
add(element){
    this.performOperation(SetOperation.Add, [element]);
}
remove(element) {
    this.performOperation(SetOperation.Remove, [element]);
}
clear(element) {
    this.performOperation(SetOperation.Clear, [element]);
}
```

Listing 11: Implementation of the mutator functions for the RW-Set

5 IMPROVING THE REACTIVITY OF CRDTS IN AN RCB-BASED APPROACH

As explained in section 2, RCB ensures that messages are reliably delivered in causal order. To this end, the RCB middleware will hold a message if it can determine that there are missing messages which are causally dependent, and will only deliver the message once all missing dependent messages arrive. As argued in section 2.3.2, this can make a CRDT less reactive, and as a result, the CRDT contains an outdated state even though the information to compute an updated state has already reached the replica node.

To solve this issue, we propose that the buffer of the RCB middleware where these messages are held is made accessible (reified) to CRDT implementors as part of the framework interface. In the context of a pure operation-based CRDT, this buffer can then be used to construct an *incomplete* partial ordered log. We say it is incomplete because the buffer can contain gaps of missing causal dependencies. The incomplete log can then complement the existing log and compacted sequential state to represent the full CRDT state. Furthermore, entries in the main PO-Log and the compacted state can be made redundant by entries from the incomplete log. However, entries in the incomplete log cannot be redundant as long as they have not yet been moved to the main PO-Log as concurrent operations that might be affected by the operation may yet arrive.

This approach has some additional advantages besides providing a more reactive CRDT. Since entries from the incomplete log can cause entries from the main log to become redundant, it can be used for decreasing memory consumption whenever intermediate disconnections are common. In these cases, it is hard to determine causal stability as not all replicas will be responsive. But the incomplete log can be used to determine redundant operations even if they may be missing causal dependencies.

6 EVALUATION

In this section, we validate our approach by running several performance experiments that aim to answer the following questions:

- how does our approach compare to the vanilla pure operation-based framework in terms of log size?
- what is the benefit of stability messages and the overhead it incurs?
- what is the impact of using the log size as a heuristic for triggering the stability messages?

6.1 Setup

We ran our experiments on a notebook machine with the following hardware specifications and software versions:

CPU	2,7 GHz Quad-Core Intel Core i7 (I7-8559U)
Memory	16 GiB
OS	macOS 10.15.5
Node.js	v13.12.0
TypeScript	v3.9.5

For our experimental setup, we are running several Flec actors on the machine with instances of the RW-Set implementation. These instances are configured as replicas of each other. Flec is running on top of Node.js, and is compiled using TypeScript. We then repeatedly perform operations on each replica, and either measure their log sizes or analyse the log contents depending on the experiment. Our results are not platform specific since we evaluate the log size rather than memory usage. This allows us to clearly evaluate the different extensions.

6.2 Methodology

For our experiments, we employ an implementation of the Remove-Wins set (RW-Set). An RW-Set cannot solely rely on the redundancy mechanisms of pure operation-based CRDTs and requires compaction through causal stability to limits its log size, making it ideal for our experiments. In particular, we implemented two versions: a pure-operation based (RW-Set) CRDT implementation as described in [3], and an extended implementation using our eager stability approach.

Algorithm 5 shows the core logic for benchmarking. For every benchmark we configure the properties of the set replicas (this happens in `setupSets`), enabling/disabling stability messages, tweaking the size of the message interval, or setting up a log-size dependent trigger. A total of TOTAL_ROUNDS * ROUND_SIZE items will be added to the sets, where the source set (the set where the operation is directly applied to) is changed every round. After every add operation, statistics regarding the size of first set's log will be measured.

Since the analysis code and CRDT implementations are deterministic, we do not need to perform multiple measurements. All the numbers from the experiments can be exactly reproduced and therefore there is no need for multiple runs from which confidence intervals are computed.

Algorithm 5: Core logic for RW-Set benchmarking

```
Data: sets
var current_set := 0;
var step := 0;
setupSets();
while step < TOTAL_ROUNDS * ROUND_SIZE do
    sets[current_set] ⟵ add("element" . step);
    takeMeasurements();
    if step mod ROUND_SIZE == 0 then
        current_set := ( current_set + 1) mod
          NUMBER_OF_SETS;
    end
    sleep(STEP_TIME);
end
```

6.3 Assessing Meta-Data Removal for the Vanilla Pure Op-Based Framework

Figure 5 plots the results for our first test, where we look at the behaviour of the vanilla pure-op RW-Set implementation. We perform the test on a system with 2, 4, and 8 replicas. As explained in our methodology, we repeatedly keep adding items to the sets. To ensure that every replica eventually performs an update and that we eventually can determine causal stability, the source node for the operations is switched every 100 operations. E.g. the first 100 operations will be performed on set 0, the next 100 on set 1, intermediately wrapping back to the set 0 once we pass the last set. Every operation on one replica will be propagated to the other replicas in the system.

The plot in figure 5 shows a clear zig-zag pattern in the results: only every 100 operations, when the source replica is switched and an update has been pushed from the new source replica, can the CRDT remove elements from the logs. This is because only at that point does new causality information about earlier operations become available, which may be enough for some replicas to determine causal stability. This also explains the initial slope in the graphs: a replica can only start determining stability once it has received updates from all other replicas. The graph shows that for the first set this happens after the 101st operation in a system with 2 replicas. For systems with four and eight replicas,

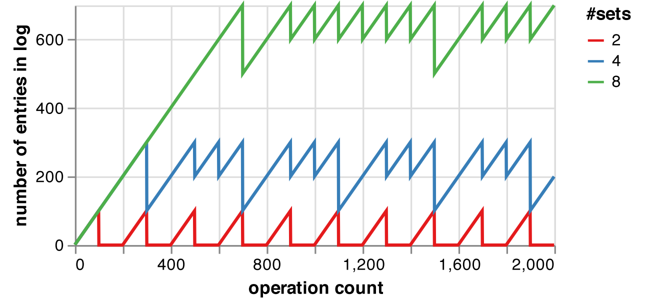

Figure 5: Numbers of entries in log of a pure-operation based Remove-Wins set, as operations are being applied to the sets in the system. Every 100 operations the source replica is changed. Measurements take for a system with 2, 4 and, 8 replicas.

this is after the 301st and 701st operation respectively. From that point on, every 100 operations the system can determine causal stability for operations issued $((NR_OF_REPLICAS - 1) * 100)$ to $((NR_OF_REPLICAS - 2) * 100)$ operations earlier, which implies that the log will always be at least the size of the number of operations issued afterwards.

Note, however, there is an apparent exception to that trend in the graph: there is a slightly larger dip in the log size every $(NR_OF_REPLICAS - 2)$ switches. To understand what exactly is going on we plot a dissected view of the log for the system with 4 replicas in Figure 6. Each colour in the graph represents the source set for a particular entry in the log. For example, a light blue entry means that the log contains an entry for which its operation originated in set 1. For set 0, 2, and 3 it takes 300 operations before the items can be removed from the log, for set 1 it only takes 200 operations.

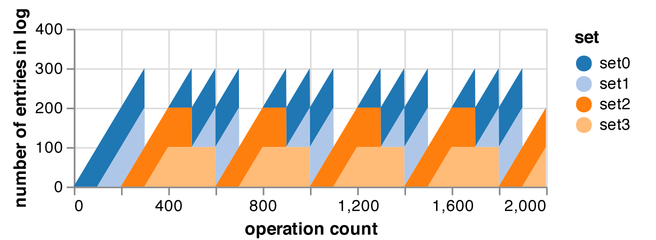

Figure 6: Numbers of entries in log of a pure-operation based Remove-Wins set, as operations are being applied to the sets in the system. Every 100 operations the source node is changed. Measured in a system with 4 replicas. The colours represent the source of the entries in the logs.

The reason for the apparent exception is because as set 1 directly follows set 0 (which we are measuring), set 0 only needs causal information from set 2 and 3 to be able to determine causal stability for the operations issued by set 1. This information becomes available after the next two replica switches. For all the other replicas there is always one extra node in between. E.g. for operations from

set 3 we first have to go through set 0, set 1, and set 2 before enough causal information is available.

6.3.1 Conclusion. These experiments show that there is a large impact on the size of the log when replicas do not regularly push out updates, one that grows with the number of replicas in the system. The results confirm the missed memory optimisation opportunities that the vanilla causal stability algorithm used in pure-op CRDTs suffers from.

6.4 Assessing the Benefits of Stability Messages

In this section, we validate in terms of log size what the benefits of a pure operation-based framework with eager stability determination are when compared to a pure operation-based one in the context of an RW-Set implementation. In particular, we will compare three different setups:

- Vanilla RW-Set replicas without any stability messages, meaning the framework only relies on causality information of propagated messages to deduce causal stability (no acks).
- RW-Sets replicas with our extension for stability messages; interval set to 10 operations (int=10).
- RW-Sets replicas with our extension for stability messages; interval set to 50 operations (int=50).

In all of these setups, we use four replicas, which means that each experiment is performed under circumstances identical to those from the previous section. We compare the results of this set-up to RW-Sets that are implemented on our proposed framework.

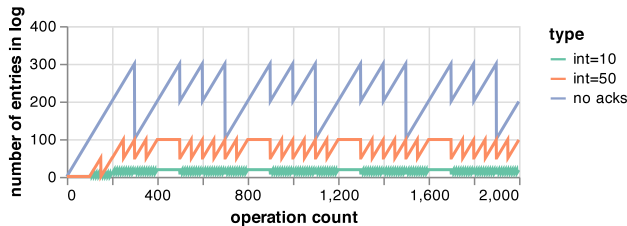

Figure 7: Comparison of the number of entries in the log of a pure-operation based Remove-Wins set, in a system of 4 replicas, but with no additional stability messages, stability messages every 10 operations, and stability messages every 50 operations. Every 100 operations the source node is changed.

Figure 7 shows the result of this experiment. A clear drop in log size can be observed when utilising stability messages, demonstrating their effectiveness. A smaller jigsaw pattern is visible, with drops every 10 or 50 operations depending on the setup. Because stability messages are more frequent and do not depend on multiple replicas communicating, there is no initial slope.

The slight build-up that is visible is due to the nature of the benchmarking setup. Figure 8 shows the log in more detail for the test where the interval is set to 50 (shown in red in figure 7). Because we switch replica every 100 operations and only send stability messages after an interval of 50 messages (meaning, after 51, 101, 151... operations) the last 50 operations from the previous set will remain in the log. As such, the number of entries in the log can stack up at times.

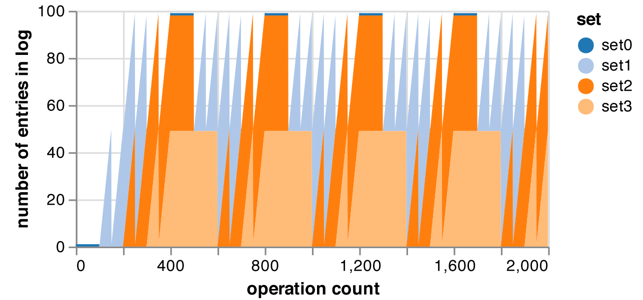

Figure 8: Detailed view at the number of entries in log of a pure-operation based Remove-Wins set, in a system with 4 replicas and stability messages every 50 operations.

Figure 9 shows the results of an experiment with the same three setups but in which we change the source node every 200 operations. In the case of the vanilla RW-Set without stability messages, the initial slope has doubled while the two instances with causal stability messages remain stable. The experiments confirm that there is a large memory improvement when stability messages are utilised when compared to the vanilla pure-op approach.

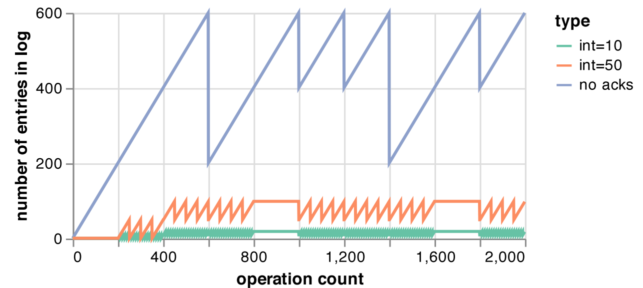

Figure 9: Same comparison as in Figure 7, but with the difference that the source node is changed every 200 operations.

6.5 Assessing the Network Overhead

In this section, we assess the network overhead that our approach incurs. As described in section 3, we use stability messages for announcing causal stability, resulting in an increased network usage (as in bandwidth consumption). Figure 10 shows the total number of messages sent when using the same initial setup as in the previous experiment (following figure 7). As shown in the graph, the overhead decreases when increasing the interval, i.e. the longer the interval is, the less overhead. This illustrates the trade-off between network (i.e. the number of stability messages sent to the network) and memory usage (i.e the size of the log).

In our implementation we utilise separate messages per replica, meaning that the total number of messages (including those for

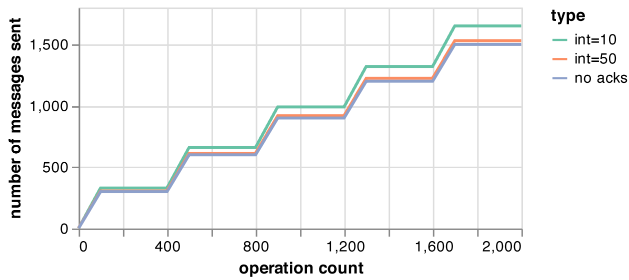

Figure 10: Comparison of the total number of sent messages for a pure-operation based Remove-Wins set, in a system of 4 replicas, but with no additional stability messages, stability messages every 10 operations, and stability messages every 50 operations. Every 100 operations the source node is changed.

propagating operations) in a system is relative to both the number of operations applied and to the number of replicas. Figure 11 shows the result of the previous experiment repeated, but with 8 replicas instead of 4. As expected, the overhead has increased as more stability messages have to be sent.

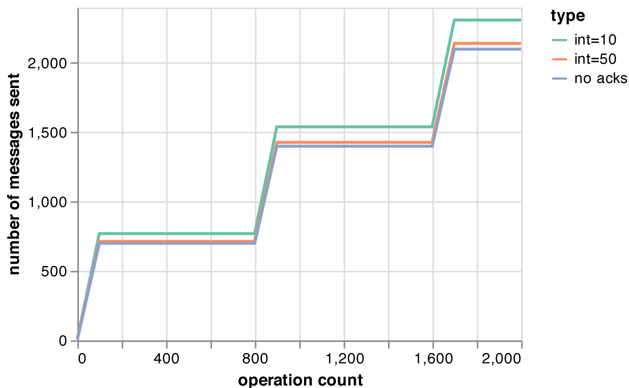

Figure 11: Comparison of the total number of sent messages for a pure-operation based Remove-Wins set, in a system of 8 replicas, but with no additional stability messages, stability messages every 10 operations, and stability messages every 50 operations. Every 100 operations the source node is changed.

This extra overhead is unavoidable in system designs where replicas can only talk directly with each other. In systems where multicasting is possible, the overhead and message sending can be dropped dramatically as all replicas can be addressed in one go.

This experiment shows that there is a network overhead when using eager stability determination, but that it may be acceptable as a tradeoff with the improved memory consumption.

6.6 Assessing the Benefits of Using Log Size as a Heuristic for Stability Messages

In this section, we assess what the impact is of using the log size as a heuristic for triggering the stability messages, aside from the interval-based approach. This may be useful to cope with the build-up of stability messages that was observed in the previous experiments (as seen in figure 8). Additionally, it can be used by developers to implement a better tradeoff between network and memory usage.

Figure 12 shows what the effect of using the log size as a heuristic is when enabled for the RW-Set. Again, we are using the same setup as the previous sections, but with the following additions:

- In the instance where the interval is set to 10 operations, we put the trigger on 15 log entries.
- In the instance where the interval is set to 50 operations, we put the trigger on 75 log entries.

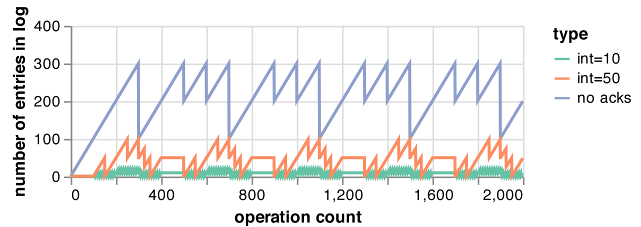

Figure 12: Same comparison as in Figure 7, but with the difference that nodes will additionally try to trigger stability messages if the number of entries log exceeds a certain size (75 entries for the system with int=50 and 15 entries for the system with int=10). Every 100 operations the source node is changed.

In the graph we observe that the log can still grow to be larger than the trigger limit. The reason for this is that replicas can only broadcast stability messages for operations that they initiated. Consequently, replicas may reach their limit by receiving operations of other replicas, and will not be able to remove these entries until they receive stability messages for them. This can also be seen more clearly in figure 13, where we have a detailed plot of the log for the case where the trigger is set to 75 (and the interval is 50).

In general, as all replicas will receive updates and eventually hit the trigger limit, they will push out stability updates. The overall log size drops to about half of the consumption of what it was in previous experiments without this heuristic, with limited build-up. To conclude, we show that allowing custom heuristics, such as a trigger on the log size, can be beneficial for improving memory efficiency and allows developers to fine-tune the tradeoffs according to the data type.

7 CONCLUSION AND FUTURE WORK

Conflict-free Replicated Data Types (CRDTs) are important data structures in the domain of distributed programming for easing the development of geo-distributed applications. They allow for concurrent operations on replicas and guarantee that eventually, all replicas will end up in the same state. To handle concurrent operations, causal relationships between operations need to be tracked.

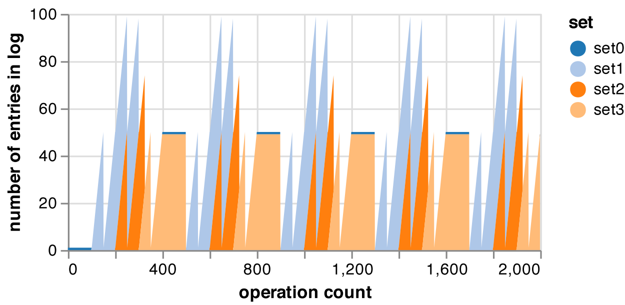

Figure 13: A detailed look at the number of entries in the log of a pure-operation based Remove-Wins set, in a system with 4 replicas and stability messages every 50 operations and when the number of entries in the log exceeds 75. Every 100 operations the source node is changed.

Traditional operation-based approaches model this by explicitly including logical clocks or unique identifiers as meta-data with operation messages. This, however, leads to an over-complicated design where CRDT implementations have to deal with semantics and causal ordering in a mixed way. One way to deal with this is by relying on a communication layer implementing reliable causal broadcasting (RCB) which handles causal ordering and reliable delivery. The pure-operation based CRDT framework takes this approach and compacts and cleans up meta-data when operations are causally stable.

In this paper we showed that relying only on causal information for deducing causal stability of operations may not be enough, as it may take too long before redundant meta-data can be removed. Moreover, we describe how the reliance on RCB limits the reactivity of operation-based CRDTs when compared to traditional approaches. Operations may be buffered for some time, waiting for other causally related operations to arrive.

To solve these issues, we propose a novel operation-based framework which is more eager in determining causal stability, by taking advantage of reliable delivery. We evaluated our approach by performing several experiments demonstrating its effectiveness. The results show that our more eager approach to determine causal stability yields promising benefits in log compaction time. To deal with the reactivity issue, we propose to make the buffer of the RCB layer accessible to developers.

By providing an open operation-based CRDT framework, we believe that this paper additionally contributes to better language implementations for CRDTs. As future work, we would like to efficiently implement incomplete partial logs as an extension for pure operation-based CRDTs. Secondly, we would like to provide a formal description of this extension and explore the difference in complexity with traditional approaches. Finally, we would like to use a formal model for proving the correctness of our eager causal stability approach.

REFERENCES

[1] H. C. Baker and C. Hewitt. 1977. The Incremental Garbage Collection of Processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. Association for Computing Machinery, New York, NY, USA, 55–59. https://doi.org/10.1145/800228.806932

[2] C. Baquero, P. S. Almeida, and A. Shoker. 2014. Making Operation-Based CRDTs Operation-Based. In *Distributed Applications and Interoperable Systems*, Kostas Magoutis and Peter Pietzuch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 126–140.

[3] C. Baquero, P. S. Almeida, and A. Shoker. 2017. Pure Operation-Based Replicated Data Types. *CoRR* abs/1710.04469 (2017). arXiv:1710.04469

[4] J. Bauwens and E. Gonzalez Boix. 2020. Flec: A Versatile Programming Framework for Eventually Consistent Systems. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20)*. Association for Computing Machinery, New York, NY, USA, Article 12, 4 pages. https://doi.org/10.1145/3380787.3393685

[5] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. 2012. An optimized conflict-free replicated set. , 12 pages.

[6] K. P. Birman and T. A. Joseph. 1987. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.* 5, 1 (Jan. 1987), 47–76. https://doi.org/10.1145/7351.7478

[7] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. 2012. Cloud Types for Eventual Consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 283–307. https://doi.org/10.1007/978-3-642-31057-7_14

[8] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix., J. Dedecker, and W. De Meuter. 2007. AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. In *XXVI International Conference of the Chilean Society of Computer Science (SCCC'07)*. Iquique, Chile, 3–12. https://doi.org/10.1109/SCCC.2007.12

[9] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. 2006. Ambient-Oriented Programming in AmbientTalk. In *ECOOP 2006 – Object-Oriented Programming*, Dave Thomas (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 230–254.

[10] D. P. Friedman and D. S. Wise. 1976. The impact of applicative programming on multiprocessing.

[11] R. Hyun-Gul, J. Myeongjae, K. Jin-Soo, and L. Joonwon. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel and Distrib. Comput.* 71, 3 (2011), 354 – 368.

[12] G. Kaki, S. Priya, KC Sivaramakrishnan, and S. Jagannathan. 2019. Mergeable Replicated Data Types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 154 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360580

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

[14] N. Preguiça. 2018. Conflict-free Replicated Data Types: An Overview. arXiv:cs.DC/1806.10254

[15] M. Shapiro. 2017. Replicated Data Types. In *Encyclopedia Of Database Systems*, Ling Liu and M. Tamer Özsu (Eds.). Vol. Replicated Data Types. Springer-Verlag, 1–5. https://doi.org/10.1007/978-1-4899-7993-3_80813-1

[16] M. Shapiro, N Preguiça, C. Baquero, and M. Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Technical Report 7506. INRIA.

[17] A. S. Tanenbaum and M. van Steen. 2006. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.