Memory Efficient CRDTs in Dynamic Environments

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 in order to guarantee both high availability of systems and an optimal user experience. Conflict-Free Replicated Data Types (CRDTs) are a family of data types specially designed for highly available systems which guarantee some form of eventual consistency. However, memory usage may grow unboundedly in their implementations, as garbage collection of meta-data is not tackled in most approaches.

In this paper, we explore a memory management model for operation-based CRDTs in dynamic setting, where nodes can dynamically join a network, and where the implementation can remove unnecessary meta-data employed by CRDTs used to determine the order of operations applied in different replicas. We first describe how new nodes will be brought up-to-date and fully linked with other replicas, and later we introduce our memory management model which allows meta-data to be removed. We benchmark the memory usage of an add-wins set using different garbage collection techniques in various situations and show how our approach can be beneficial in comparison to state of the art techniques.

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

Keywords Replication, CRDTs, Memory management, Dynamic networks

ACM Reference Format:
Jim Bauwens and Elisa Gonzalez Boix. 2019. Memory Efficient CRDTs in Dynamic Environments. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '19), October 22, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358504.3361231

VMIL '19, October 22, 2019, Athens, Greece
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6987-9/19/10…$15.00
https://doi.org/10.1145/3358504.3361231

1 Introduction

Many modern distributed systems keep multiple copies of data (replicas) between distributed components. When a partial failure occurs, the copies ensure availability of the data in the system. This also improves performance by lowering request latencies and as a result, provides an better user experience as requests are served faster. A system is expected to provide users with up-to-date information, but keeping replicas consistent is a complex task. One of the main reasons to the complexity of ensuring consistent behaviour is that there is no notion of a global clock in distributed systems. This has as result that the order of updates applied to different replicas in the system cannot be precisely determined, which complicates determining when updates are concurrent and how conflicts caused by concurrent updates should be resolved.

Conflict-Free Replicated Data Types [16] (CRDTs) are a promising approach for replication because they avoid the need to deal with conflicts. CRDTs are replicated data structures which can be concurrently updated without requiring synchronisation among replicas. To this end, CRDTs constrain the type of operations which can be applied to them to be commutative. As such, CRDTs are said to be strongly eventual consistent (SEC). Assuming no new updates happen to a set of replicas, they will eventually converge to the same state without conflicts.

In order to handle concurrent operations and ensure commutativity, a CRDT implementation typically keeps some meta-data. For example, some CRDTs might use tombstones to ensure that removal operations are commutative [16]. Tombstones act as placeholders for removed entries and ensure that if a replica receives a removal operation for an item before its add operation, that the removal is still processed when the add arrives. However, for many CRDT types this meta-data grows unboundedly, and it should be removed once it becomes useless to ensure that the application remains responsive.

Most of CRDT research has focused on providing formal specifications of different data types (e.g. OR-Sets, replicated growable arrays, embeddable counters and more) [2, 6, 10, 16, 20], but very few efforts have focused on embedding CRDT in actual language implementations [11, 14]. Previous work on CRDT specifications has already pointed out that memory usage may grow unboundedly, however, this is still an open issue in the community. Bieniusa et al. [4]

introduced an optimised version of the OR-Set CRDT type, where tombstones were removed when redundancy could be determined. However, this approach is specific to the OR-Set CRDT data type and as such, cannot be generalised to any replicated data type. To the best of our knowledge, Baquero et al. [3] is the only work that proposes a general scheme for removing meta-data. In this work, CRDTs are built on top of a Reliable Causal Broadcast (RCB) middleware that provides causality information for operations and notifies CRDT implementations when certain operations become causally stable, allowing for the removal of meta-data belonging to operations that can no longer be concurrent. However, their solution assumes a fixed network where the number of replicas is known when a CRDT is created, while dynamic environments can be found in many real-life CRDT use-cases.

In this paper we propose a model that is able to work in dynamic environments. We first describe how a CRDT can work in a dynamic environment by describing a join model that allows new nodes to get a replica of a CRDT. We then detail our memory management scheme to remove meta-data in this context. Furthermore we show how to speed up the process of meta-data removal by being pro-active in determining when an operation can no longer be concurrent. This allows us to relieve memory pressure at a faster rate when compared to previous approaches. Finally we evaluate our implementation by benchmarking the memory usage of an add-wins set. We configure the set to work with various meta-data removal techniques, including our own approach and test its behaviour in various situations, and show that our approach offers benefits over other tested approaches.

2 Background

Eventual Consistency (EC) is a family of consistency models in which replicas are allowed to diverge from one another with the condition that the state of the replicas must eventually converge [19].

One specific model in the EC family is strong eventual consistency [17] (SEC). In this model, replicas converge to a consistent state without any form of synchronisation. The model guarantees that regardless of the order of operations and assuming no new updates happen to the set of replicas, they will eventually converge to the same state, i.e. all replicas that have observed the same updates will have an equivalent state. This way, replicas can be updated simultaneously and concurrent updates will be resolved without requiring synchronisation.

2.1 CRDTs

Conflict-Free Replicated Data Types [15, 16] (CRDTs) are a family of data structures that adhere to the SEC consistency model. As defined by Shapiro et al. there are two main types of CRDTs: operation-based CRDTs and state-based CRDTs. Operation-based CRDT replicas share mutation events, while state-based CRDTs replicas share their entire state. Both types are functionally equivalent to each other, meaning that they can be transformed to either type. The main implication of the CRDT type is at implementation level, where the choice can impact processing and network usage.

```lua
local counter = CounterCRDT("shared_counter", function (value
    )
      print('Counter updated', value)
end)

counter:increment(5)
counter:decrement(2)
counter:increment(1)
```

Listing 1. Using a counter CRDT

An example of a commonly used CRDT is a counter as shown in listing 1. A counter implementation has a single integer register that can be incremented and decremented. The addition and subtraction operations are commutative, e.g. 5 - 2 + 1 is equivalent to 1 + 5 - 2. Intermediately between the operations different states may be observed, but eventually when all updates are applied to all copies all states will be numerically equal.

Listing 1 shows how such a counter CRDT could be used in LuAT, an extension to Lua featuring CRDTs that we will use for our experiments. We will further describe LuAT in section 5, but this section briefly introduces the necessary syntax to illustrate a counter CRDT. `CounterCRDT` takes a string representing a nominal type used for other nodes in the network to discover this CRDT, and a callback function which will be applied when the CRDT is updated. It then creates a counter CRDT instance which can be discovered in the network by means of the `shared_counter` string. Mutation of the CRDT happens by calling the `increment` or `decrement` operations on the CRDT reference. The underlying framework will ensure that these operations are replicated to all other nodes.

Another example of CRDT is an OR-Set as seen in listing 2, which replicates a set of items.

```lua
local set = ORSet("shared_set", function (set)
      print('Set updated: ', table.concat(set:toList(), ", ")
          )
end)

set:add("element")
set:add("this is another item")
set:add(5)
set:remove("element")

if set:lookup(5) then
  print("Element 5 is in the set")
else
  print("Element 5 is not in the set")
end
```

Listing 2. Using an OR-Set CRDT

Just like `CounterCRDT`, `ORSet` takes a string representing a nominal type for linking replicas together, and a callback function which is applied when the set is updated. Mutation of the set happens by calling the `add` or `remove` operations on the set reference. Using the `lookup` method items can be tested if they exist in the set. The `toList` method returns a Lua table containing all elements as a non-replicated list.

At implementation level, the OR-Set CRDT generates a unique identifier per add operation, and uses these tags to determine the causal ordering of add operations. Remove operations leave tombstones behind and record causal ordering of remove operations. This causal information is required to ensure the add-wins semantics that the OR-Set promises, e.g. by ordering adds over removes in concurrent situations.

2.2 Problem Statement

Most CRDTs rely on timestamps or unique identifiers to help determine causal ordering of operations. For example, as explained in the previous section an OR-Set uses unique identifiers and tombstones for this purpose, and both accumulate over the lifecycle of an OR-Set. This meta-data becomes useless as soon as operations fully converge and no concurrent operations are possible anymore. Because this meta-data is useless, it should be removed in order to have a more efficient memory usage. We call operations that have been observed by all replicas and thus aren't able to have any concurrent operations ongoing "causally stable".

In order to be able to determine if an operation if causally stable, the CRDT middleware needs to be able to access causality information. Baquero et al. [3] have proposed to rely on a Reliable Causal Broadcast [5] (RCB) middleware which tracks causal information for all messages sent in 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)

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

Note that this definition of causal stabilities implies that some nodes may not yet be aware of the causal stability status for an operation, while others do. Because causal stability is used to remove redundant meta-data, it is possible that a node that is not yet aware of the causal stability of an operation receives an operation with lacking meta-data. Baquero et al. solve this by making that the RCB middleware buffers the operation until the node is aware of the causality state.

Note also that this definition is limiting: it is only possible to determine causal stability for an operation if and only if every other node sends a message following the operation, in order to collect enough causal information. This means that if one node doesn't issue any operation, no causal stability can be determined at any node. Baquero et al. also assume a fixed network, i.e. that every replica in the network is known at set-up. This is problematic as it limits the applicability of CRDTs. Many modern distributed applications are collaborative or require that the number of replicas can grow dynamically. Consider for example, collaborative text editors like Google Docs in which an unknown number of users can join an editing session. Even web-shop applications such as Amazon allow the same user to hold several replicas of a shopping list or card which can be accessed via different devices (e.g. mobile phone, tablet, laptop..).

2.3 Our Approach

In this paper we explore a memory efficient implementation approach for CRDTs which can work in a dynamic environment in which the number of replicas is not known beforehand. In order to achieve this we first need to design a mechanism to allow new nodes to be added to a system and obtain a replica. We call this a join model. We show how such a system could be set-up and what is required to allow new nodes to join during the life time of a system. Consequently, we design a memory management model taking into account the join model for dynamic networks. More concretely, we adapt the definition of causal stability by Baquero et al. to a dynamic environment, and combine it with a memory management model that aims to remove meta-data eagerly without relying on operations to be issued to garbage collect causal information.

3 A Join Model for CRDTs

We design a join model in which new nodes can join the network and are able to construct a proper state for their local replica and start collaborating with others. We focus on operation-based CRDTs in this model, but will later discuss the impact on state-based CRDTs.

3.1 Assumptions

The original paper on CRDTs [16] actually assumes that all replicas are in a correct state when performing updates, and does not specify how to ensure this for replicas that have not started with the same initial state. Follow-up papers on CRDTs targeting systems that are typically used in dynamic environments, such as for use in collaborative tools, generally rely on a centralised approach with a membership protocol [12, 13]. To the best of our knowledge this is the first paper to describe a join model for CRDTs, in particular, for a dynamic environment.

In our approach we do not assume a centralised design, but rather a full-mesh peer-to-peer configuration. We define a node in the system as a VM or machine which hosts a replica of a CRDT. We employ the term network as the set of nodes hosting a replica for one CRDT. We assume that every node in the network holds a single replica of a CRDT. In the case that a node disconnects we assume this to be a transient failure [18], and expect that the node will eventually recover and return to the network. In other words, we assume a fail-and-recover failure model. Messages that cannot be delivered due to transient failures are assumed to be buffered until the network is restored. Furthermore, we assume that eventually all messages arrive, i.e. reliable communication with no message lost nor duplication, e.g. TCP/IP, and that there are no byzantine failures, i.e. no malicious nodes. We expect that the middleware relies on acknowledgement messages in order to ensure message delivery and processing, which is a reasonable expectation as TCP/IP and similar protocols also make use of handshakes and acknowledgements for ensuring in-order delivery of messages.

3.2 Defining a Join Model

The challenge in defining a join model for a dynamic environment is getting the new node an up-to-date state without missing operations or performing operations multiple times. In our approach for a new node to join it has to make contact with *only* one node within the network, and request to join. This can be seen in Figure 1 where the white *N* node sends a join message to grey node *A*. Grey nodes represent nodes that are fully part of the network. Dashed lines between nodes represent a 'knows' relation. In our example nodes *A*, *B* and *C* all know each other.

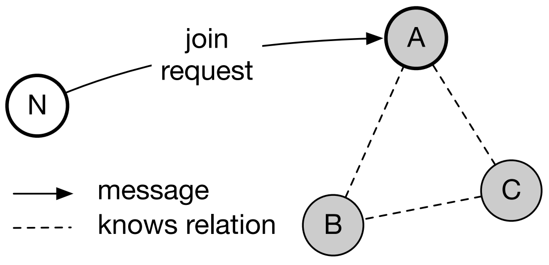

Figure 1. Step 1: A node requests to join a network

When a node receives a join request, it responds by sending network information about its known nodes to the new node, and will then add the new node to its known nodes.

The new node can start receiving updates from nodes that know it, but it has to buffer all incoming messages until it has been fully acknowledged by all members of the network.

The new node then uses the network information that it has received to link to all other nodes in the network, as can be seen in Figure 2. This is done so that the new node can start receiving updates from the entire network.

When the new node has been acknowledged by the entire network, it is still missing a full state. This is solved by requesting the state from the node it sent its join request to, as visualised in Figure 3.

The new node can now apply the received state as its initial state and start applying any buffered operations that do not have a timestamp earlier than the state that was received, as there may be duplicate operations between the state and operations received from other nodes.

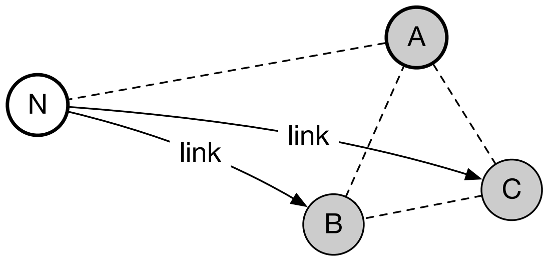

Figure 2. Step 2: The new node contacts all other nodes in the network

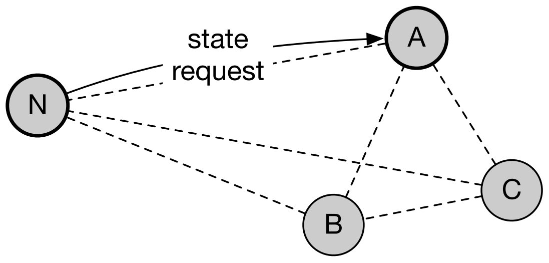

Figure 3. Step 3: The new node performs a state request once it has been fully acknowledged

The state has to be transferred when the new node is fully linked to the network, as otherwise operations concurrent to the link requests may never be delivered to the new node.

3.3 Concurrent Joins

We now detail how our model handles concurrent joins, i.e. when two (or more) nodes of a network are simultaneously handling the join request of other new nodes. To this end, consider the scenario shown in Figure 4, where node *N* is sending a join request to node *A* and concurrently node *O* is a sending a join request to node *C*. Following the protocol described above, *O* will receive the network state from node *C*, however this state may still lack information about node *N*. Similarly, the network state that node *N* receives may be lacking information on node *O*. As both nodes may not be aware of each other they will not be able to link with each other as-is.

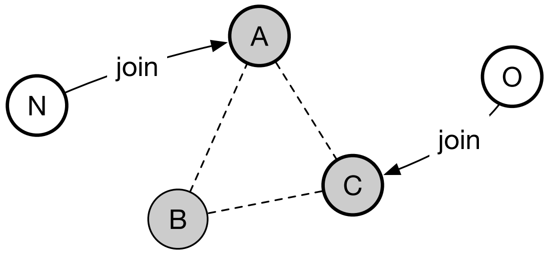

Figure 4. Two nodes perform simultaneously join requests to different nodes in the network

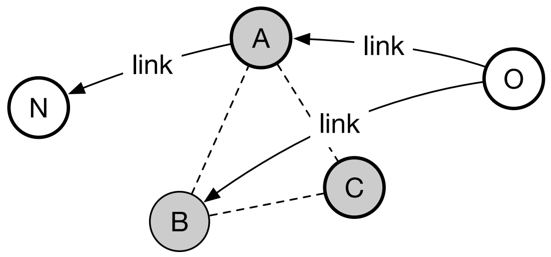

Figure 5. Node A forwards a link message from node O to node N

In order to handle such concurrent join request, our approach makes that nodes that have received join request send copies of the link requests they received to the new nodes they are handling. In the example in Figure 5, node *A* forwards a link request from node *O* to node *N* (the new node being handled by *A*). Similarly, node *O* will eventually receive a forwarded link request from *N*.

By forwarding the link requests, nodes that join concurrently will always receive link requests from each other before they can finalise joining the network. The nodes handling the join request (e.g. *A* and *C* in Figure 4) will reject state requests from new nodes if they have not handled additionally forwarded links, e.g. a node is only allowed to finalise its join if it has connected to all nodes from the network state plus later forwarded link messages.

4 Memory Management

Recall that memory management in CRDTs is all about being able to remove meta-data that is used for causal tracking of operations. This meta-data is necessary in order to enforce ordering between concurrent operations. However, when an operation can no longer be concurrent with other operations, e.g. all subsequent operations have a greater timestamp, this meta-data is no longer required and needlessly consumes resources. The operation is said to be causally stable.

In this section we first explore how causal stability can be determined in our join model and then we explain how we can determine causal stability in a more eager way and thus speed up time to garbage collection.

4.1 Causal Stability in Dynamic Environments

In order to determine causal stability, we need to examine operations that are sent between replicas. Depending on the node that issued the operation, actions may need to be taken that diverge from the causal stability algorithm for fixed networks. We will look at the different cases an operation on the system can be categorised in, when performed in a network where a node is actively joining. With this, we look at what information is required for determining causal stability. Operations in a network where a node is joining can be categorised as follows:

1. Operations sent from the node contacted in the join request
2. Operations sent from the new node
3. Operations sent from other nodes

Most of these cases can be divided into sub-cases, depending on the state of the operation and node.

First, in the case where operations originate from the node responding to the join request there are two sub-cases.:

1. The operation is already (causally) stable: the operation will be transferred in compacted state to the new node, there is no impact on causal stability determination.
2. The operation is not yet stable: in this case the node will need to add the new node to its list of nodes that it needs to receive a message from in order to determine causal stability. This is because causal stability can only be determined if a node receives causal information from all nodes. This is no exception for the node handling the join request, it will eventually need this information from the new node as well.

In the case that the operation originates from the node that is joining, the new node should buffer all applied operations until that the node itself is fully acknowledged by all other nodes in the network. As soon as it's fully acknowledged, it can issue its operation just as any other node that is part of the network.

When the operation originates from neither the new node nor the node handling the join request, we there are two sub-cases that have to be checked:

1. The node has already acknowledged the new node: in this case, the source node can issue an operation as usual: the operation will be stable if all nodes, including the new node, eventually send a message to the node so that it has enough causal information to decide on stability.
2. The node has not yet acknowledged the new node: in this case the join operation of the new node to the source node is concurrent with the operation. The operation can be issued as usual, however the source node needs to add the new node to the list of nodes it needs to hear from to determine causal stability. Because the underlying middleware is using RCB, and the join node has already processed messages from the new node, the source node will have to process the join before it can process any message from the join node. This way there can be no issue that the new node decides that the operation is causally stable before the new node has been linked to it. The operation itself will be transferred to the new node in the state message from the join node at the end of its join process.

4.2 Eagerly Collecting Meta-data

Relying on the determination of causal stability is not enough to collect meta-data. Because a node needs to receive an update from every other node before it can determine if an operation is causally stable, this may never happen. In this paper we propose to piggyback on the communication layer between nodes, relying on operation acknowledgements to determine when all nodes have observed an operation.

When a node issues an operation, it broadcasts this to all other nodes. Once it has received acknowledgement that all nodes have processed the operation, it follows that no concurrent operations can occur after this, and that the operation in fact is causally stable. In contrast to the classical algorithm where all nodes need to wait on causal information to determine causal stability, the node that issued the operation will broadcast the causal stability state of the operation to all nodes. Receiving nodes can then use this information to remove meta-data belonging to the operation. Naturally this imposes a network overhead on the system, but this may be a necessary tradeoff when memory resources are scarce.

In the pure-op based approach, the middleware notifies the CRDT layer that an operation is causally stable. With our approach the middleware will be able to provide this information earlier if it receives such a causally stable message from another node.

5 Implementation Details

We implemented the pure-op based framework from Baquero et al. along with our join model and eager stability protocol in LuAT [1]. LuAT is a Lua library for distributed programming which incorporates the concepts of Ambient-oriented Programming [8]. Similar to the AmbientTalk language[7], LuAT features an actor-based programming model that allows actors to communicate and coordinate over a mobile ad hoc network in a distributed setting. It has support for ambient acquaintance management, non-blocking message passing between actors, failure handling through message buffering, leasing and future-like synchronisation constructs. In what follows, we provide enough details on LuAT's programming model to follow the contributions of this work.

In order to facilitate the development of programming constructs for CRDTs, LuAT features a generic CRDT framework which allows for development of CRDT data types with built-in support for replication. Implementers can easily extend the built-in CRDT support with new CRDTs by inheriting from provided prototype objects and then extending it with the required CRDT logic. A meta-object-protocol (MOP) allows for inspecting and modifying CRDT implementation concepts like upstream and downstream phases.

[1]https://gitlab.soft.vub.ac.be/jimbauwens/LuAT

5.1 Extending LuAT CRDT Support With RCB

The heart of our implementation begins with an extension to the existing LuAT CRDT support with reliable causal broadcast (RCB). This ensures that all operations received by CRDTs will be processed according to their causality. Furthermore, the implementation also keeps track of what timestamps are causally stable.

Listing 3 shows the core structure of the implementation. Method implementation code has been removed for brevity. `CRDT_RCB` is an abstract prototype that implements this logic, that can be used to prototype CRDTs. Besides the behaviour described above, it provides automatic discovery of other replicas in the network and operation replication to known replicas. It piggybacks on the network publication and discovery mechanisms provided by LuAT for this functionality.

The prototype provides some methods that are expected to be extended or invoked by CRDT implementations. `performOperation` is used to apply an operation to a CRDT. It will first trigger the `onOperation` hook locally, which can be used by CRDT implementations to apply CRDT-type specific operations and then broadcast the operation to all know replicas. `doOperation` will be invoked on those replicas, which will use `tryOperation` to see if operation is not missing any causal dependencies. If it is not, it will issue the `onOperation` hook and then call `tryBufferedOperations` to check if there are any buffered operation that can be applied, as the operation received may have been a causal dependency to operations that were buffered in an earlier stage. If the operation does have causal dependencies it will be put in a buffer and only be applied when its dependencies have been resolved.

```lua
local CRDT_RCB = Object()

-- constructor
function CRDT_RCB:init(tagname) ... end

-- internal behavior
function CRDT_RCB:tryOperation(rclock, op, args, local_) ...
      end
function CRDT_RCB:tryBufferedOperations() ... end

-- public behavior
function CRDT_RCB:isCausallyStable(id, ts) ... end
function CRDT_RCB:doOperation(rclock, op, args, local_) ...
      end
function CRDT_RCB:performOperation(op, args) ... end

-- hook entry points
function CRDT_RCB:onOperation(rclock, op, args) end
function CRDT_RCB:onNewReplica(ref) end
function CRDT_RCB:onLoaded() end
function CRDT_RCB:onCausallyStable(id, ts) end
```

Listing 3. Core structure of the CRDT RCB base prototype

Every time an operation is performed, `performOperation` will check what timestamps are causally stable. If such timestamps exists, the `onCausallyStable` hook will be invoked for this timestamp. CRDT implementations can use this hook to remove meta-data belonging to causally stable operations.

The prototype also provides hooks for monitoring new replicas joining the network, and for when a local CRDT is fully initialised. It does not however decide on how a join has to be handled, this is left for implementations of the prototype.

5.2 Pure-operation Based CRDT

The POLog prototype as seen in listing 4 provides an implementation for pure-operation based CRDTs with support for our join model. This prototype extends on CRDT_RCB and implements several of its hooks.

There are three main parts to this component: operation handling, causal stability handling and join handling. The prototype hooks into the onOperation method to receive updates on applied operations. This method stores the operation in a local log and if the isRedundantByLog and isRedundantByOperation hooks are implemented they will be used to check if any existing entry in the log might be redundant. If this is the case, any redundant entry will be removed them from the log. These hooks help CRDTs that extend on the POLog to implement custom semantics.

Meta-data removal is handled by implementing the isCausallyStable hook from the CRDT_RCB prototype. It uses the hook to iterate over the operation log to find operations that may be stable. If any operation is stable it will invoke the setEntryStable hook and remove its associated entry from the log. The setEntryStable hook must be implemented by sub-prototypes to store the operation in compacted form without meta-data.

```
local POLog = Object(CRDT_RCB)

--constructor
function POLog:init(tagname) ... end

-- join logic
function POLog:setupState(state) ... end
function POLog:getState() ... end
function POLog:join() ... end
function POLog:link(id) ... end

-- operation from CRDT middleware
function POLog:onOperation(rclock, op, args) ... end
function POLog:isCausallyStable(id, ts) end
function POLog:onNewReplica(ref, refs) ... end

-- hooks for helping with log compaction
function POLog:isRedundantByLog(entry) return false end
function POLog:isRedundantByOperation(e1, e2, er) return
    false end
function POLog:setEntryStable(entry) end
function POLog:removeEntry(entry) end
```

Listing 4. Core structure of the pure operations based CRDT prototype

The join model is implemented by using the onNewReplica hook to listen for new replicas in the network. When a node joining the network discovers a first other replica, it will send out the join message (as in our join model) to the other replica. This message is handled by the join method at the receivers side, which will respond by returning a list of known nodes. Following this the new node will send the link message to every node in this list. Incoming link messages are handled by the link method. Finally, when the new node has received a response to all link requests, the getState method will be invoked on the initially discovered replica to get a full state, which will then be applied on the new replica using the setupState method. After this process the new replica will have a full up-to-date state and be properly linked to all other replicas.

5.3 Add-wins CRDT

Finally, on top of the pure operation-based CRDT layer we create an actual CRDT that can be used. We implement an add-wins CRDT, where add operations have higher precedence over concurrent remove operations. Listing 5 shows the implementation of this CRDT. By implementing the isRedundantByLog and isRedundantByOperation provided by POLog the semantics of the set are encoded: add operations that are ordered before remove or clear operations become redundant (they are removed from the log). The setEntryStable hook implementation ensures that causally stable operations are stored in a more compacted form without meta-data. The POLog layer will ensure that the operation is additionally removal from the log.

In order to obtain the state of the CRDT both the log and set storing compacted entries have to be queried. This is implemented by the toList method, which iterates over the log and builds up a state, which is then merged with the causally stable entries from the cc table. The operations that CRDT itself accepts are implemented as simple methods which relay the operation to the CRDT_RCB layer which ensures that they will be applied and replicated.

```
local AWSet = Object(POLog)

-- constructor
function AWSet:init(tag, cb)
  POLog.init(self, tag)
  self.cb = cb
  self.cc = {}
end

-- POLog hooks

function AWSet:isRedundantByLog(entry)
  local op = entry.operation
  return op == "rmv" or op == "clear"
end

function AWSet:isRedundantByOperation(e1, e2, er)
  return VectorClock.precedes(e1.clock, e2.clock) and (
        e2.operation == "clear" or e1.args[1] == e2.args[1] )
end

function AWSet:setEntryStable(entry)
  local element = entry.args[1]
  self.cc[element] = true
end

function AWSet:removeEntry(entry)
  local element = entry.args[1]
  self.cc[element] = nil
end


-- basic api

function AWSet:toList() ... end

function AWSet:add(...)    return self:performOperation("add"
      ,   {...}) end
function AWSet:remove(...) return self:performOperation("rmv"
      ,   {...}) end
function AWSet:clear(...)  return self:performOperation("
      clear", {...}) end
```

Listing 5. Implementation of an add-wins CRDT

6 Evaluation

In order to evaluate our implementation we performed several benchmarks on an add-wins set CRDT, implemented as a pure-op based CRDT. First, we look at how the memory consumption of a set varies with new nodes joining and new replicas being created. Then we compare how the set behaves under operations in several configurations where we change how garbage collection is performed. Finally we benchmark memory consumption in an environment with periodic network disconnections and evaluate how it effects garbage collection.

6.1 System Information

All experiments were performed on devices running Lua 5.1.5 on Ubuntu 19.04, with 16GiB RAM and an Intel i7-6500U CPU. In order to correctly measure heap using the Lua VM, a Lua garbage collect was issued before every memory measurement.

6.2 Performance Of a System After Growth

In our first experiment we evaluated the effectiveness of our join algorithm and its integration with the garbage collection system.

We start out by creating an add-wins CRDT set, which we fill with 100 distinct elements. We then dynamically start adding nodes to the network, and for every node added we perform an additional 1000 add operations of the same elements to the set (as to ensure that the total set size will remain 100 items large) and measure its memory usage. Figure 6 shows the stabilised memory consumption for a number of nodes. E.g. after 8 nodes have joined the system the stabilised memory consumption is 420Kb, if we keep on adding items (without growing the network) the memory consumption will stay idle. The raise in memory consumed when the number of nodes is higher is due to extra meta-data information on the different nodes in our middleware (this includes references to other nodes, larger vector clocks, causal bookkeeping).

6.3 Memory Consumption Over Several Models

In this experiment we compared the memory consumption of three variants of the add-wins set. We compare a version without garbage collection, a version with garbage collection (using causal stability as described by Baquero et al.) and finally a version extended with the changes proposed in section 4.2. We performed 1000 add operations on the set, repeatedly selecting numbers between 1 and 100. The time between every operation was set to 400ms. Every 100 operations the source node for the operations was switched. This test was performed on a network with 2, 4 and 8 nodes. The results are visible in Figure 7.

We can observe several behavioural patterns for the different algorithms from the graph. Firstly, the set that does not

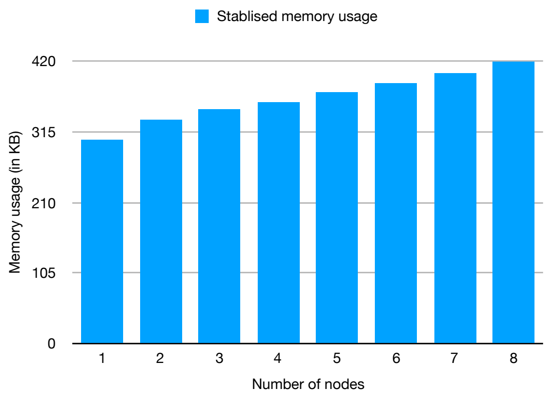

Figure 6. Stabilised memory usage after growing a system with an add-wins set CRDT with additional nodes.

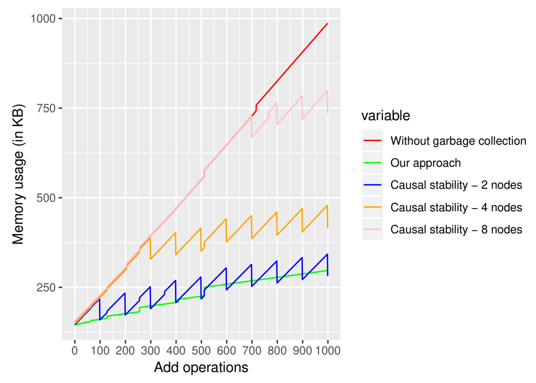

Figure 7. Comparison between an add-wins set implementation without memory management and with, with a varying number of nodes.

utilise memory management clearly uses more memory per add, and no memory is ever reclaimed. When we add causal stability determination and associated GC to the set, which benchmarks are represented in the graph under the label "causal stability", memory is reclaimed as soon as the system has enough causal information to determine causal stability. For 2 nodes this is as of 100 adds, every 100 adds, when the system switches to another node for initiating operations. For 4 and 8 nodes the period to initial cleanup is larger as the system can only perform GC once it has learned the causal information from all other nodes. Once causal information has been received from every node, some cleanup can be performed per 100 adds. Finally, we benchmarked our version in which causal stability information is actively shared in the network after every operation, and as the resulting graph shows, meta-data is removed at a much higher pace. The only growth in memory is directly related to the extra memory used for the add operation. This benchmark was performed with 4 nodes, we did not use a varying number of nodes as it did not impact the results for this specific measurement greatly.

6.4 Memory Consumption Under Partial Failures

In our final test we benchmark the behaviour of our add-wins set in a system that suffers from disconnections. We perform 10.000 add operations, of 100 distinct elements. Every 100 adds we toggle the network state and switch to the next element. When offline, the CRDT is unable to sent out operations to other replicas in the system, and is thus unable to determine the causal stability of those operations. The results of the benchmark can be seen in Figure 8.

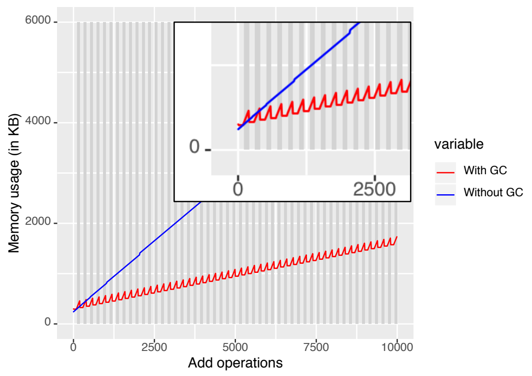

Figure 8. Comparison between an add-wins set implementation without memory management and one with. Grey bars represent when the node of measurement is taken offline. The system consists out of 4 nodes.

The graph shows that the set without GC is unaffected by the offline status, which is normal as it never performs GC. With our eager approach, the system is unable to transfer causal stability information during an offline period, and is thus unable to perform any meta-data removal which results in a higher memory usage during this period. As soon as the network returns to an online state, causal stability can be determined and garbage collection can occur.

7 Conclusion

Conflict-free Replicated Data Types (CRDTs) are promising programming abstraction to replicate data in a distributed system as they guarantee that eventually all replicas will end up in the same state. However, in order to handle concurrent operations, some meta-data is kept in the implementation of the data type to track causal relationships between operations. In order to remove this data and compact internal data structures of CRDTs, the causal stability of operations has to be determined. When an operation is causally stable, its associated meta-data can be removed.

In this paper we show how the concept of causal stability can be brought to dynamic environments. We first describe a join model, in which a CRDT network can grow, and then how causal stability determination can work in this environment. Finally, we propose an adaptation to be more eager in determining causal stability, allowing for quicker meta-data removal. We evaluated our approach by performing several experiments, and demonstrate the effectiveness of our join model. The results show that our more eager approach to determine causal stability yields benefits in garbage collection time and remains functional after partial failures.

We believe this paper is a first milestone into having efficient language implementation of CRDTs. As future work, we envision three main tasks. First, we would like to formalise our join model (with eager garbage collection) and prove its correctness. We plan to model our approach using the the framework from Gomes et al. [9], which provides abstractions for the behaviour of networks and its interaction with CRDTs. Second, we will look into the applicability of state-based CRDTs and their more optimised variants such as Delta CRDTs [1] in dynamic environments. Full state-based CRDTs may have a less complex join model due the nature of full state sharing, but their design implies a much larger network overhead. Delta CRDTs optimise on this network usage, but state sharing with new nodes will be closer to the operation-based CRDT approach in this paper. Finally, we would like to adapt our join model to tackle permanent failures, and to allow nodes to unanticipated leave a network.

References

[1] P. S. Almeida, A. Shoker, and C. Baquero. 2014. Efficient State-based CRDTs by Delta-Mutation. *CoRR* abs/1410.2803 (2014). arXiv:1410.2803

[2] C. Baquero, Paulo S. Almeida, and C. Lerche. 2016. The Problem with Embedded CRDT Counters and a Solution. In *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC '16)*. ACM, New York, NY, USA, Article 10, 3 pages. https://doi.org/10.1145/2911151.2911159

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

[5] 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

[6] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 271–284. https://doi.org/10.1145/2535838.2535848

[7] 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

[8] 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.

[9] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. 2017. Verifying Strong Eventual Consistency in Distributed Systems. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 109 (Oct. 2017), 28 pages.

https://doi.org/10.1145/3133933

[10] 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.

[11] M. Kleppmann and A. R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel & Distributed Systems* 28, 10 (oct 2017), 2733–2746. https://doi.org/10.1109/TPDS.2017.2697382

[12] M. Letia, N. Preguiça, and M. Shapiro. 2010. Consistency Without Concurrency Control in Large, Dynamic Systems. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 29–34. https://doi.org/10.1145/1773912.1773921

[13] X. Lv, F. He, Y. Cheng, and Y. Wu. 2018. A novel CRDT-based synchronization method for real-time collaborative CAD systems. *Advanced Engineering Informatics* 38 (2018), 381 – 391. https://doi.org/10.1016/j.aei.2018.08.008

[14] C. Meiklejohn and P. Van Roy. 2015. Lasp: A Language for Distributed, Coordination-free Programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP '15)*. ACM, New York, NY, USA, 184–195. https://doi.org/10.1145/2790449.2790525

[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] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. 2011. Conflict-free Replicated Data Types. In *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems (Lecture Notes in Computer Science)*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.), Vol. 6976. Springer, Grenoble, France, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

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

[19] W. Vogels. 2008. Eventually Consistent. *Queue* 6, 6 (Oct. 2008), 14–19. https://doi.org/10.1145/1466443.1466448

[20] P. Zeller, A. Bieniusa, and A. Poetzsch-Heffter. 2014. Formal Specification and Verification of CRDTs. In *Formal Techniques for Distributed Objects, Components, and Systems*, E. Ábrahám and C. Palamidessi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 33–48.