# Making Causal Based CRDTs Scalable

Juliane Marubayashi[1] and Carlos Baquero[2]

[1] MEIC, Universidade do Porto juliane.marubayashi@gmail.com
[2] DEI, Universidade do Porto & INESC TEC cbm@fe.up.pt

**Keywords:** CRDTs · Scalability · Add-wins sets

## 1 Introduction

When applications become more broadly distributed, they also become more exposed to network failures. As the CAP theorem [2] states, a system cannot simultaneously be available, consistent, and fault-tolerant. Fault tolerance is indispensable, since faults are imposed on the system by the environment, leaving developers to choose between availability and consistency.

Given this trade-off, a recent trend in building highly available systems is now focusing on local first software [4], a design paradigm where the primary version of the data stays in the originating node, and the network and data-center infrastructure is now dedicated to data propagating and durability.

In this context, Conflict-free Replicated Data Types (CRDTs [7]) play an important role by proving a foundational approach to eventual consistency. They allow the system to be temporarily inconsistent but guarantee that all replicas will converge to the same state if no new updates are made. CRDTs play a significant role in the industry and are extensively used (e.g., Redis [6], Atom Teletype, Phoenix).

Although CRDTs provide many benefits, they also have some drawbacks if used directly over a high number of replicas, as their state linearly increases with the number of active replicas in the network. Devices with insufficient memory resources may experience poor performance in large networks. For example, in an application network with a million nodes, mobile devices that use this application would have to store the state of all nodes, draining the user's device resources. Since the number of servers is lower than the number of clients, CRDTs usually are implemented only on the server side. In such deployments, the client-server connection can become brittle and lose data when connections drop and are later resumed. An ideal solution would be to have CRDTs both on the client and server side, if metadata size can be controlled and not grow linearly with the number of clients.

Actually, such solutions exist [1,3,5] for a specific type of CRDTs, counters. Counters have the advantage of concerning fungible quantities, which simplifies value aggregation and transfers among nodes. However, there is no solution yet to make CRDT sets, multi-value registers, and maps scalable in the same way.

We propose a solution built on top of Add-Wins Observed Removed-Set (AWORSet) model, which can be easily generalized to other causal CRDTs. The

AWORSet supports two fundamental operations: addition and removal. When a client adds an element $e \in \mathbb{E}$, the AWORSet adds a tagged element $(id, n, e)$ in its state $s$, and a tag/tombstone $(id, n)$ to its causal context $c$. Conversely, when a client attempts to remove an element, the AWORSet deletes the corresponding entries from $s$, but the causal context remains unchanged. The presence of a given tag in the causal context that does not correspond to an active entry means that some element was present and later was deleted, and this helps the merge procedure.

## 2 ROSES protocol

In this section, we give an intuition over the ROSES (**R**enaming **O**perations for **S**calable **E**ventually-Consistent **S**ets) protocol and its guarantees. As the name suggests, ROSES is a scalable, eventually consistent protocol for sets and registers CRDT-based data structures. It allows users to add and remove elements from nodes while providing high availability due to renaming mechanisms.

We consider a distributed system model where each node interacts asynchronously with one another and operates under logical time and unrestricted computational processing speed. The network does not corrupt data but can lose, re-order, or duplicate messages. Any partitions will eventually heal, and messages will be delivered as expected. To recover data after a crash, each node in the system has a unique identification and permanent storage.

The protocol requires a two-layer network, with clients in the lower layer and servers in the upper. Clients only communicate with upper-layer nodes, while servers can exchange information with other servers and nodes in the lower layer. This structure allows clients to apply operations locally and eventually upload information to servers. In this context, clients in ROSES have the flexibility to add and remove elements without immediate synchronization with upper-layer servers. The upload frequency depends on the system's needs, either happening at specific time intervals or after a certain number of operations. Therefore, write-intensive systems might get performance benefits when clients upload information in time intervals.

In ROSES, clients don't have server affinity, they can start a new server connection upon network partitions or server crashing. Even when a protocol is ongoing and some messages have been exchanged, upon crashing the client node can connect with another server, which in turn becomes responsible for sending the desired information to the offline server when it becomes operational again. Therefore, the design ensures that even if the client goes offline while the destination server is down, an auxiliary server guarantees that the information will be delivered upon recovery. This design contributes to ROSES' high availability service.

Even with the renaming mechanism, ROSES holds idempotency when receiving clients' data guaranteeing that duplicated messages will not affect the system's consistency. When adding new elements, the client tags these elements with their unique identifier. Upon upload, the server replaces these tags to in-

clude its identifier. The renaming trick makes the causal context proportional to the number of servers instead of clients, and thanks to that, the system is largely scalable. However, if the server receives a duplicated message, it will not recognize that the elements were already incorporated. This is natural since the server renamed the tags and therefore is not able to verify that the elements are already part of its state. To address this issue, we implemented a four-handshake protocol equipped with clocks, which guarantees that duplicated messages are discarded if received.

A node can have many ROSES executions happening simultaneously, but it can't have more than one execution per destination. An origin node needs to finish the current execution in order to send more data than what was assigned. This fact, summed with four-handshake communication, makes the propagation of new elements slower. The client, however, will not suffer from the delay since it can read its own writes.

### 2.1 Evaluation

In this section, we conduct an analysis and provide insightful observations regarding the performance of the ROSES protocol with an emphasis on memory usage. To achieve this, we compared its memory usage against a plain AWORSet datatype. Furthermore, we developed an event simulator using Rust programming language. The source code for the simulator can be found at `https://github.com/Jumaruba/ROSES-protocol`.

Several simulations were executed and encompassed a diversified number of clients $c = \{16, 64, 256\}$ and servers $s = \{4, 16\}$. Figure 1 a network of 4 servers and 64 clients spanning 30 units of time. In each unit of time, nodes can add or remove a maximum of 10 elements chosen from a predefined set.

Upon transmitting its state to another node, the receiving node has a probability of prompt response. The nodes cease issuing operations after 20 units of time. Notably, before this mark, the ROSES and the AWORSet line curves seem to be flat, but it actually increases slowly. Although the causal context always increases, the probability of removing and adding elements is the same.

After time unit 20, the nodes seize generating new operations. As time progresses, the space occupied by nodes using ROSES reduces gradually. This reduction occurs because the information stored under the client's identifier begins to be held on servers instead.

In contrast, the AWORSet exhibits different behavior. As the network strives to achieve consistency and nodes begin to acknowledge elements received from other nodes, the mean memory consumption of the AWORSet tends to increase.

## 3 Conclusion and Future Work

We introduced and explained the assurances of ROSES protocol, which improves the scalability of CRDTs based on sets and registers through a renaming mechanism. The protocol significantly improves memory usage compared to classical

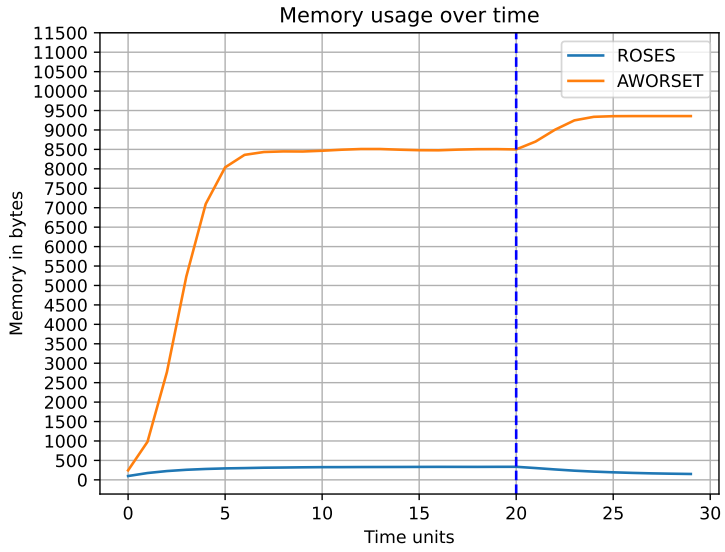


Fig. 1: Memory usage analysis over time between AWORSETs and ROSES

CRDTs. As the used space increases linearly with the number of servers instead of the number of replicas, nodes implemented on top of ROSES tend to use less memory in large networks.

Nonetheless, we are still improving the protocol to implement optimizations, such as reducing the space occupied by a token, as it contains invariants. The main probable improvement would be expanding the protocol to support a topology with $n$ layers. We are aware that the two layers design is not ideal, but it provides a base for future and more robust designs.

## References


1. Almeida, P.S., Baquero, C.: Scalable eventually consistent counters over unreliable networks. Distributed Computing **32**(1), 69–89 (Feb 2019). https://doi.org/10.1007/s00446-017-0322-2, http://link.springer.com/10.1007/s00446-017-0322-2
2. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing. p. 7. PODC '00, Association for Computing Machinery, New York, NY, USA (Jul 2000). https://doi.org/10.1145/343477.343502, https://dl.acm.org/doi/10.1145/343477.343502
3. Enes, V., Baquero, C., Almeida, P.S., Leitão, J.: Borrowing an Identity for a Distributed Counter: Work in progress report. In: Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data. pp. 1–3. ACM, Belgrade Serbia (Apr 2017). https://doi.org/10.1145/3064889.3064894, https://dl.acm.org/doi/10.1145/3064889.3064894
4. Kleppmann, M., Wiggins, A., van Hardenberg, P., McGranaghan, M.: Local-first software: you own your data, in spite of the cloud. In: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2019. Ink and Switch, ACM Press (2019). https://doi.org/10.1145/3359591.3359737, https://www.inkandswitch.com/local-first.html
5. Power, C., Laddad, S., Douglas, C., Hellerstein, J., Suciu, D.: TopoloTree: From O(n) to O(1) CRDT Memory Consumption Via Aggregation Tree Gossip Topologies (2023)
6. Redis: Diving into Conflict-Free Replicated Data Types (CRDTs) (Mar 2022), https://redis.com/blog/diving-into-crdts/
7. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-Free Replicated Data Types. In: Défago, X., Petit, F., Villain, V. (eds.) Stabilization, Safety, and Security of Distributed Systems. pp. 386–400. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24550-3$_2$9