Resilient Conflict-free Replicated Data Types without Atomic Broadcast

Daniel Brahneborg[1][a], Wasif Afzal[2][b] and Saad Mubeen[2][c]
[1]*Braxo AB, Stockholm, Sweden*
[2]*Mälardalen University, Västerås, Sweden*

Keywords: Resilience, Data Replication.

Abstract: In a distributed system, applications can perform both reads and updates without costly synchronous network round-trips by using Conflict-free Replicated Data Types (CRDTs). Most CRDTs are based on some variant of atomic broadcast, as that enables them to support causal dependencies between updates of multiple objects. However, the overhead of this atomic broadcast is unnecessary in systems handling only independent CRDT objects. We identified a set of use cases for tracking resource usage where there is a need for a replication mechanism with less complexity and network usage as compared to using atomic broadcast. In this paper, we present the design of such a replication protocol that efficiently leverages the commutativity of CRDTs. The proposed protocol CReDiT (CRDT enhanced with intelligence) uses up to four communication steps per update, but these steps can be batched as needed. It uses network resources only when updates need to be communicated. Furthermore, it is less sensitive to server failures than current state-of-the-art solutions as other nodes can use new values already after the first communication step, instead of after two or more.

1 INTRODUCTION

Many distributed systems need to efficiently manage external resources. These resources could be, e.g., network traffic, the number of times to show a specific web advertisement, and more. In this work, we will consider an application with one or more users, each one paying for the resources they use. Each user has a credit balance, representing payments made and resources used. This balance is then used as basis for their next invoice. The system clearly must take great care in maintaining these credit balances.

Regardless of how reliable modern computer components have become, occasional server outages are unavoidable (Aceto et al., 2018; Bailis and Kingsbury, 2014). In order to make the *service* available despite these *server* outages, we need multiple servers (Cheng et al., 2015; Rohrer et al., 2014; Rothnie and Goodman, 1977; Vass et al., 2020), preferably independent and geographically separated (Dahlin et al., 2003). The challenge is to maintain accurate records of the resource consumption across all these servers.

Unfortunately, maintaining consistency in a distributed system can easily lead to decreased performance (Didona et al., 2019), and in the presence of network partitions, fully distributed consistency and high availability simply cannot co-exist (Brewer, 2000; Gilbert and Lynch, 2004). An interesting exception was identified by Alsberg and Day (Alsberg and Day, 1976), suggesting what is basically a precursor to Conflict-free Replicated Data Types (CRDTs) (Shapiro et al., 2011):

> *"An example [of a specific exception] is an inventory system where only increments and decrements to data fields are permitted and where instantaneous consistency of the data base is not a requirement."*

CRDTs have become popular for distributed systems over the past few years, partly because of their convenient property of having the same value regardless of the order of the operations performed on them. When instantaneous consistency is not required, local operations can be performed on them immediately, without the need for time-consuming network round-trips. The new state is instead regularly broadcast to the other nodes. All nodes therefore eventually get the same value for the object. There are two main groups of CRDTs:

State-based CRDTs send their full state (Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha,

[a] https://orcid.org/0000-0003-4606-5144
[b] https://orcid.org/0000-0003-0611-2655
[c] https://orcid.org/0000-0003-3242-6113

Brahneborg, D., Afzal, W. and Mubeen, S.
Resilient Conflict-free Replicated Data Types without Atomic Broadcast.
DOI: 10.5220/0011314500003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 516-523
ISBN: 978-989-758-588-3; ISSN: 2184-2833
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

2017) between the nodes. This makes them immune to both packet loss and packet duplications, but it can quickly lead to excessive network usage for data types with a large state, and to massive storage requirements when there is a high number of clients (Almeida and Baquero, 2019). A special case of these are **delta-based CRDTs**, which only transmit the part of the state changed by local updates (Almeida et al., 2018; Enes et al., 2019).

Operation-based CRDTs send only the individual operations (Baquero et al., 2017). These typically use less network resources, but require reliable delivery where all operations are successfully received by all nodes exactly once (Younes et al., 2016).

Even if the operation order on a single CRDT object does not matter, many applications update an object based on the values of another. In order to enforce such causal dependencies, most CRDT implementations use reliable causal broadcast (RCB) where all nodes get the same set of packets in more or less the same order (Birman and Joseph, 1987; Schneider et al., 1984). RCB is typically based on atomic broadcast, which can ensure not only that all packets are delivered in the same order, but also that this happens only if all nodes are still reachable. A simple way to implement atomic broadcast is Skeen's algorithm, which requires a set of network packets sent back to the sender, and then a third set of "commit" (Gotsman et al., 2019) packets to all destinations from which the sender got the reply. When the causality check is based only on Lamport clocks (Lamport, 1978) this can give false positives, in turn leading to unnecessary network traffic and delays (Bauwens and Boix, 2021).

The purpose of this work is to find a replication protocol for state-based CRDTs not having any causal dependencies, where the replication uses less network resources than previously proposed solutions. We use user credits as the motivating example, typically implemented as CRDT PN-counters (Shapiro et al., 2011). In short, a PN-counter is a set of pairs of integers $\mathbb{Z}$, merged by the operation `max()`, where the integers are used for positive and negative changes, respectively. Its value is the difference between these two integers. We refer to the paper by Shapiro et al. for a more detailed description.

Our proposed protocol *CReDiT* (CRDT enhanced with intelligence) extends state-based CRDTs by augmenting the local state with additional information in order to avoid unnecessary network traffic. All data is periodically resent until it has been acknowledged by each other node, making the protocol immune to occasional packet loss.

We will describe this work using Shaw's framework (Shaw, 2001), which categorizes research in three different ways. First is the research setting, which is what kind of research question or hypothesis is being addressed. Our setting is *Methods/Means*, described in Section 2. Next is the research approach. Here the desired result is a new *Technique*, described in detail in Section 3. The third way is the result validation, which is done in Section 4. We discuss the results in Section 5, present related work in Section 6, and end the paper with our conclusions in Section 7.

2 METHOD

In Shaw's framework (Shaw, 2001), the purpose of a "Methods/Means" setting is to find an answer to a research question such as "what is a better way to accomplish X". After defining our system model in Section 2.1, we will therefore define our "X" in Section 2.2, and what we mean by "better" in Section 2.3.

2.1 System Model

We assume that we have a distributed system of independently running nodes, communicating over an asynchronous network. The physical network can use any topology, as long as there is at least one logical path between each pair of nodes. We further assume fair-lossy links, i.e., packets may be dropped, but if a packet is sent infinitely often it will eventually be received. Packets may also be duplicated and received out of order. The nodes have local memory and stable storage, and can recover after crashing. We also assume there are some number of clients, each one connecting to any node or nodes. As the clients send requests to a node, their resource counter in that node is updated. Figure 1 shows the situation with x clients and y nodes. This work addresses the communication between the nodes, shown with dashed lines.

2.2 Functional Requirements

The functionality we need, i.e. our "X", matches almost exactly what Almeida and Baquero (Almeida and Baquero, 2019) call *eventually consistent distributed counters* (ECDCs). These use the *increment* operation for updating the counter, and *fetch* for reading its current value. *Fetch* returns the sum of updates made. A second call to *fetch* returns the previous value plus any locally made updates since the previous call. Eventually, *fetch* will return the same value on all nodes, i.e., the above named sum of updates. In addition to an ECDC, we also allow negative updates,

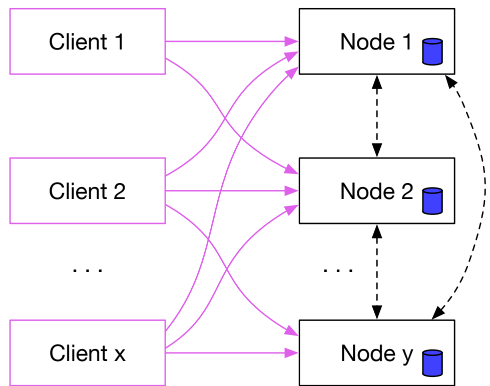

Figure 1: We have x clients connected to y nodes, which in turn are all connected to each other. Each node maintains its own database of the credit balance for each client.

which means we can count both the resources used and the payments made.

2.3 Quality Requirements

We also need to specify our quality requirements, i.e. what we mean by "better". We base these on ISO 25010 (ISO/IEC, 2020), a taxonomy which puts quality attributes into eight different groups of characteristics, each one divided into a handful of sub-characteristics. The latter are written below in the form *Main characteristic / Sub-characteristic*.

The CAP theorem (Brewer, 2000; Gilbert and Lynch, 2004) tells us that given a network partition, we cannot have both data consistency and availability. With the ISO 25010 nomenclature, this means we need to choose between *Functional Suitability / Functional Correctness* (the needed degree of precision) and *Reliability / Availability*. We strongly prioritize the latter, as it is usually a good business decision to let customers keep using your service, even when facing the risk of occasional overdrafts.

For the *Performance Efficiency / Capacity*, we assume the system has up to about 10 nodes, and that there are up to 1000 clients using its resources. For now, we do not address the remaining quality characteristics in ISO 25010 (ISO/IEC, 2020).

Assuming that all clients are independent, we can model the time between each update for each client using the exponential distribution. This distribution has the probability density function $f(x) = \lambda e^{-\lambda x}$, and the cumulative distribution function (CDF) $P(X < x) = 1 - e^{-\lambda x}$. In both functions λ is the inverse of the client specific mean interval μ, and x is the length of the interval.

This CDF has an interesting property, as it is always less than 1. In other words, there will always exist a time interval of length x without any updates. If x is measured in seconds, we will have an alternating sequence of some number of seconds with updates, and some other number of seconds without.

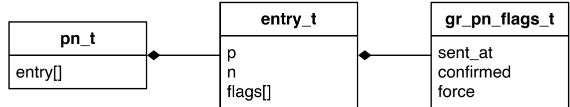

Figure 2: The three data types used by CReDiT.

3 PROPOSED TECHNIQUE

Our proposed protocol is based on PN-counters (Shapiro et al., 2011), augmented with data to keep track of the values on the other nodes. This data, and how it is used, is described in Section 3.1. Section 3.2 describes the protocol as a state machine, and Section 3.3 shows a sample scenario in a system with two nodes. The protocol is named *CReDiT*, inspired by its basis in CRDT.

3.1 Prototype Implementation

We assume the application has some sort of collection of resource counters. For the network communication, CReDiT uses a separate transport layer. Each counter, named `pn_t` in our implementation, contains a map from node identifiers to instances of the structure `entry_t`. An `entry_t` contains the two fields `p` and `n`, just as the original PN-counters.

We extend the `entry_t` structure with a map from node identifiers to `gr_pn_flags_t` structures, containing the fields `sent_at`, `confirmed`, and `force`, described below. These fields are therefore specific for each pair of nodes. The three types are shown in Figure 2.

`sent_at`: timestamp
: This is the most recent time the current value was sent to the other node. In our implementation, for simplicity but without any loss of generality, we use a resolution of one second for this field.

`confirmed`: boolean
: This is set when the incoming values from another node are identical to what is stored locally, so we know that we do not need to send the same data to that node again.

`force`: boolean
: This is set when a value must be sent on the next flush, overriding the `confirmed` flag.

In the function descriptions below, we use `A` for the local node where the code is executed, `B` for one of the remote nodes, `entry` for the instance of the

entry_t structure on A, x for a random node, and * to designate all nodes. The protocol uses the functions listed below, of which only flush() and receive() perform any network operations. We have marked the original PN-counter functionality with "**PN**" and our additions with "**New**".

init(x, p, n)
This is used when loading values from external storage on startup.

PN: It sets entry[x].p and entry[x].n to the supplied values.

New: It clears entry[x].flags[*].

update(delta)
This is called to update the resource counter.

PN: If the delta is positive, entry[A].p is increased with delta, and if it is negative, entry[A].n is increased with -delta.

New: As we know that node A is the only one updating the entry[A] fields, no other node has these exact values now, and we can therefore clear the entry[A].flags[*].confirmed flags.

fetch()
This function is called by the application to get the current value of the counter.

PN: It returns the sum of all entry[*].p fields minus the sum of all entry[*].n fields.

flush()
This should be called regularly by the application, in order to initiate the replication.

New: Entries are sent if the force flag is set, or if the confirmed flag is not set and sent_at is different than the current time. Afterwards, sent_at is set to the current time, and force is set to false. The use of sent_at here allows the application layer to call this function as often as it wants. In contrast to solutions based on atomic broadcast, CReDiT does not wait for any replies from the other nodes.

receive(B, x, p, n)
This function is called by the transport layer, when new data has been received from node B concerning values on node x.

PN: The field entry[x].p is updated to the maximum of its current value and the incoming p, and similarly for entry[x].n and n.

New: If entry[x].flags[B].confirmed is set, entry[x].flags[B].force is set. If entry[x].p or entry[x].n was changed, entry[x].flags[B].force is also set and entry[x].flags[*].confirmed are cleared. The entry[x].flags[B].confirmed flag is always set though, as we know that node B has these particular values. Finally a callback is made to the application, which can now persist the new data. This persisted data is what the application should provide to the init() function after being restarted.

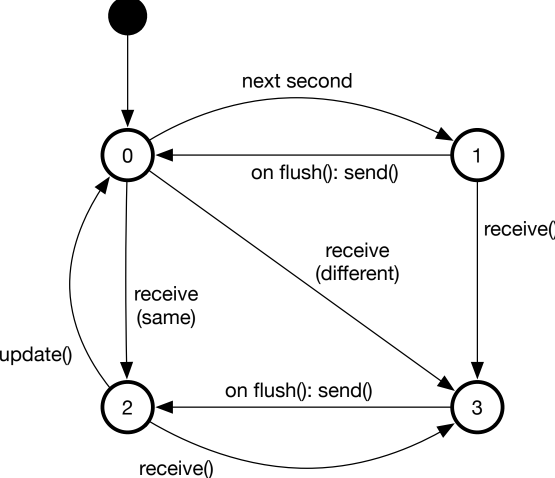

Figure 3: States on a node. Each pair of nodes has its own state. The black circle is the starting point. For the states 1 and 3, the force flag is set. For the states 2 and 3, the confirmed flag is set.

Our implementation was based on GeoRep (Brahneborg et al., 2020), which supplied networking code and configuration management for keeping track of the nodes to which data should be replicated.

3.2 State Machine

Figure 3 shows a compact summary of the algorithms and the effects of the flags. There is a separate state machine for each individual counter, and for all pairs of nodes. The state of each machine is an effect of the node specific flags in entry_t: If the force flag is set or if the current time differs from the value of the sent_at attribute, the machine is in state 1 or 3. If the confirmed flag is set, it is in state 2 or 3. Each counter starts at the filled black circle, and immediately goes to state 0. This represents the case when both force and confirmed are false. In all states, update() and receive() update the corresponding (p, n) pair(s). All functions described in Section 3.1 can be called in any of these four states, but functions not affecting a particular state are omitted for clarity.

3.3 Data Flow

In a system with the two nodes A and B, these are the steps taken when A updates a shared counter.

1. A updates the value of a new counter with +2. This creates the counter, and A sets p=2 and n=0 in `entry[A]`.
2. After at most one second, A moves B to state 1. On the next call to `flush()` from the application layer, the values for A are sent to B, after which A sets `entry[B].sent_at` to the current time.
3. When B receives this data, it stores A's values p=2 and n=0, and sets the flags `confirmed` and `force` in `entry[A]`.
4. As A has `sent_at=now` for B and `force` is not `true`, any additional calls to `flush()` will not cause more data to be sent to B.
5. B has `force` set to `true` for A, so the next time `flush()` is called on B, the (p=2, n=0) pair for A is sent back to A.
6. Next, A gets the (p=2, n=0) pair for A from B. As these are the same values it already has, it sets `confirmed` to `true` for B. It does not set `force`. After this, both A and B has `confirmed` set to `true` for each other, and agree on the (p=2, n=0) pair. No more data is sent by either side.

If the data sent in step 2 is lost, A will obviously not get this data back from B. When `flush()` is called during the next second or later, A will see the missing `confirmed` flag and send the data again. This way, the `confirmed` flag on node A prevents repeated transmissions of the same data from A to B. As we assume fair-lossy links as mentioned in Section 2.1, B will eventually receive this data.

If the data from B to A in step 5 is lost, B will still have the `confirmed` flag set, so it will not send the data again. However, A will not have this flag set, so it will send the data to B again. From B's perspective, as the `confirmed` flag for this entry is set, A and B should already have the same data. Hence, as B sees the same data again from A, it can deduce that A's `confirmed` flag is not set. B can fix that by setting its `force` flag for A, causing the data to be sent back to A on the next call to `flush()`. This way, the `force` flag on node B prevents repeated transmissions of the same data from node A to B.

The combined effect of the `confirmed` and `force` flags is that any data packet can be lost, and the protocol will still recover. Once all nodes have the same set of confirmed values, no more data will be sent until after the next call to `update()`.

4 EVALUATION

Here we discuss the validation of the proposed protocol regarding its functionality, correctness, and scalability. As both `update()` and `fetch()` only work on local data, the availability is 100% by construction.

4.1 Functional Validation

We already know that CRDTs in general, and PN-counters in particular, converge on the same value on all nodes, thanks to the broadcast and merge mechanisms (Shapiro et al., 2011) also used by CReDiT. We therefore only need to show that the new fields do not invalidate this. For `sent_at` this is obvious, as this field only limits how often data is sent.

The `confirmed` flag prevents data being sent from node A to node B, when B has shown that it already has the exact same values as A. As long as this is true, sending this data again is of no use to anyone. When the values on A change, its `confirmed` flags are cleared, giving the original CRDT behaviour. If the values on B change, this field is cleared on B, causing the data to be broadcast to all nodes, including A, which in turn clears the field for the other nodes, also getting us back to the original CRDT behaviour.

As described in Section 3.3, the `force` flag handles the case when the sent values returned to the sender (A) are lost. As long as the values on A are unchanged, A would otherwise keep sending them to B because no confirmation is received. For new values sent by A, B would notice the update and send it back to A, just as for any other CRDT.

4.2 Correctness Conditions

A state-based CRDT ensures that all updates originating on a particular node can never be done in a different order on another node, as its current state always includes its previous state. Its commutativity further ensures that even if the relative order of updates made on different nodes may differ between the nodes, the value of a CRDT object will eventually be the same on all nodes. As this order may differ between nodes, we do not get *serializability* (Papadimitriou, 1979).

Whether we get linearizability (Herlihy and Wing, 1990) is not entirely clear. Herlihy and Wing states that the "real-time precedence ordering of operations" should be respected. This is indeed the case on each particular node. However, in a distributed system with nodes A and B we can have a sequence such as the following.

1. A stores the value 1 in variable x.
2. A stores the value 2 in variable x.
3. B reads the value of variable x.
4. B reads the value of variable x.

The data replication from node A to node B may be initiated both after step 1 and 2. Furthermore, the new data may arrive to node B both before and after step 3, as well as after step 4. Node B can therefore see both the values 1, 2, or something else. Still, if node B would read the value 2 in step 3, we can guarantee that step 4 will not read the value 1 (a.k.a. *monotonic reads*). Also, if node A would read the value of variable x, after step 1 it would get 1, and after step 2 it would get 2 (a.k.a. *read your writes*).

4.3 Scalability

The memory usage for each counter is $O(n)$ for the values, and $O(n^2)$ for the flags. There is no transaction log as for operation-based CRDTs, so for a given n the memory requirement is constant.

For an update on node A, up to 4 sets of network packets are triggered. After these steps, all n nodes will have the same values, as well as knowing that the other $n-1$ nodes have them too. Because of this knowledge, no more data is sent until the next update is made.

1. Node A sends the new (p, n) pair to the other $n-1$ nodes.
2. After receiving the new pair, these $n-1$ nodes send back their updated values.
3. For a system with 3 or more nodes, the $n-1$ nodes has at least one set of `flags` where `confirmed` is not set. So, `flush()` on these nodes will broadcast the updated values to the remaining $n-2$ nodes.
4. If a packet in the previous set is received from a node y on a node x before x has broadcast the update itself, the `force` flag will be set on node x, causing the value to again be sent from node x to node y.

An update will therefore cause a total of up to $(n-1)+(n-1)+(n-1)(n-2)+(n-1)(n-2) = 2(n-1)^2$ network packets to be sent in the system. This quadratic scale-up makes this protocol unsuitable for systems with a large number of nodes, even though the decision for when this is true must be done on a case by case basis.

The packet size will be proportional to the number of updated counters since the last confirmation, but it is not affected by the number of updates to a particular counter. The number of updates also has no effect on the number of required network packets, making the quadratic scale-up less of a problem than it may seem.

Additionally, counters with no updates on a particular node, after its `confirmed` flag is set, stay in state 2 in the state machine shown in Figure 3. In this state `flush()` generates no network traffic.

4.4 Real-world Evaluation

There are a couple of seemingly obvious measurements that could be used in order to evaluate protocol behaviour in real-world situations. First, we could measure the number of function calls per second. However, as all functions either just modify local data structures or are asynchronous, this would effectively only measure the CPU speed of our test machines. Second, we could measure the time from when `flush()` is called until the data has reached all other nodes. Unfortunately, this would just measure the round-trip time between the nodes. Third, we could compare some performance aspect of the application that originally triggered this work. Currently, the best solution for that application appears to be using a replicated MySQL server. However, we have not found any way to do the required multi-master replication with geo-separated nodes, while still getting acceptable performance (of at least 1000 updates per second).

Instead, we will compare our protocol with PN-counters based on atomic broadcast. In particular, we have observed that for counters with updated data, most algorithms for atomic broadcast use fewer communication steps than CReDiT does. For counters without updates, CReDiT uses fewer. So, we want to measure the relative frequency between these two cases. From two production systems running the motivating application mentioned above, we retrieved sample log files containing the time stamps of events that would trigger an update of one of our counters.

The first file covers an interval of 91 hours in the middle of September 2021, with a total of 78 987 events. Within this interval we observed the occurrence of events during each hour, but only during 3358 out of a total of 5460 minutes, and during 35 166 out of the total of 327 600 seconds. Despite an average of 0.241 events per second, there is an event only during 10.7% of the seconds in this interval. The second file covers 6 hours in August 2021, during which there were 328 948 events, an average of 11.4 events per second. Still, there was at least one event during only 28357 of the included 28800 seconds (98.5%).

We do not have enough data points to find the most fitting statistical distribution for the events handled by the application, but it seems to be one of the uneven ones, e.g. the exponential distribution discussed in Section 2.3. The periods without any updates, where CReDiT is maximally efficient, are therefore more frequent than one perhaps would expect.

5 DISCUSSION

According to Urbán et al. (Urbán et al., 2000), having a designated sequencer serializing all operations in the system, uses the fewest number of communication steps per message, namely 2. The trade-off cost to achieve this is that the sequencer node needs much outgoing network bandwidth as it does a broadcast of all messages to all other nodes. Most other atomic broadcast protocols need more communication steps, but let each node broadcast its own messages.

As we saw in Section 4.3, our proposed protocol performs worse than this in both aspects, as it requires up to 4 communication steps and that all nodes broadcast all updated values. When there are no updates, our protocol instead does not communicate at all.

The round-trip times between each pair of nodes has little or no effect on this protocol, for several reasons. First, the updated data can be flushed at any interval, which just has to be longer than the maximum round-trip time. By default, this interval is therefore 1 second. Second, as the data sent is the full new state of each counter, the number of updates between each flush does not affect the amount of data sent. Third, as new data is immediately available to each node after being received, a temporary delay on one link between two nodes only affects those two specific nodes. This improves the reliability of the system, as updated values sent just before a crash can be used by the other nodes immediately after being received.

The increased storage requirements caused by our adding new data fields is well compensated for by the elimination of chatter on the network during quiet periods.

6 RELATED WORK

Almeida et al. (Almeida et al., 2018) address a problem similar to ours. Their δ-CRDTs support both duplicated network packets, just as state-based CRDTs do, and achieving the lower bandwidth requirements of operation-based CRDTs. Their anti-entropy algorithm (corresponding to our `flush()`) sends only the part of the state affected by local operations performed on the current node. For a CRDT with a large total state this δ-state is typically smaller than the full state replicated by state-based CRDTs.

One way to ensure that all servers have the same data is to use a replication protocol which can "guarantee that service requests are executed in the same order at all resource sites" (Alsberg and Day, 1976). The most common solution to this problem is to model the system as a replicated state machine, using a variant of Paxos (Lamport, 1998) or Raft (Ongaro and Ousterhout, 2014). For the counters we need, the request order does not matter. The implementation complexity and network bandwidth required by these protocols are therefore unnecessary.

Almeida and Baquero (Almeida and Baquero, 2019) defined Eventually Consistent Distributed Counters (ECDC), which is the same partition tolerant abstraction addressed in our work. Their solution, called *Handoff Counters*, also works well over unreliable networks. Their counters aggregate the values in a few central nodes, making them scale better according to the number of servers than our solution does. By creating a map of these counters, they would provide a reasonable solution for our resource counting. However, the aggregation is rather complex, consisting of a 4-way handshake and 9 data fields.

Skrzypczak et al. (Skrzypczak et al., 2019) also addressed the synchronization overhead of state machine replication by using a single network round-trip for updates and not having a leader. To get linearizability, they coordinate using the query operations, with repeated round-trips until the returned values stabilize. In contrast, our protocol can accept both updates and queries during all types of network partitions, and can respond to queries without doing any additional round-trips.

7 CONCLUSIONS

Generally, layered architectures are good, reducing the complexity of each individual layer. In the case of building state-based CRDTs on top of atomic broadcast, we saw that the resulting system may use unnecessary network resources. By instead taking advantage of the lack of causality between the operations of our CRDT counters, we were able to design a new protocol with lower network requirements during periods without updates. The described approach can be used with any state-based CRDT, as long as it is possible to determine if the incoming values differ from the local values.

ACKNOWLEDGMENTS

This work was sponsored by The Knowledge Foundation industrial PhD school ITS ESS-H and Braxo AB.

SCIENCE AND TECHNOLOGY PUBLICATIONS

REFERENCES

Aceto, G., Botta, A., Marchetta, P., Persico, V., and Pescapé, A. (2018). A comprehensive survey on internet outages. *Journal of Network and Computer Applications*, 113(2018):36–63.

Almeida, P. S. and Baquero, C. (2019). Scalable Eventually Consistent Counters over Unreliable Networks. *Distributed Computing*, 32:69–89.

Almeida, P. S., Shoker, A., and Baquero, C. (2018). Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162–173.

Alsberg, P. A. and Day, J. D. (1976). A Principle for Resilient Sharing of Distributed Resources. In *International Conference on Software Engineering*, ICSE. IEEE Comput. Soc. Press.

Bailis, P. and Kingsbury, K. (2014). The Network is Reliable. *Communications of the ACM*, 57(9):48–55.

Baquero, C., Almeida, P. S., and Shoker, A. (2017). Pure operation-based replicated data types. *arXiv preprint arXiv:1710.04469*.

Bauwens, J. and Boix, E. G. (2021). Improving the Reactivity of Pure Operation-Based CRDTs. In *Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC.

Birman, K. P. and Joseph, T. A. (1987). Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76.

Brahneborg, D., Afzal, W., Cauševic, A., and Björkman, M. (2020). Superlinear and Bandwidth Friendly Georeplication for Store-and-forward Systems. In *International Conference on Software Technologies*, ICSOFT.

Brewer, E. A. (2000). Towards Robust Distributed Systems. In *Principles Of Distributed Computing*, PODC. ACM.

Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, C. F. (2017). Composition in State-based Replicated Data Types. *Bulletin of EATCS*, 3(123).

Cheng, Y., Gardner, M. T., Li, J., May, R., Medhi, D., and Sterbenz, J. P. (2015). Analysing GeoPath diversity and improving routing performance in optical networks. *Computer Networks*, 82:50–67.

Dahlin, M., Chandra, B. B. V., Gao, L., and Nayate, A. (2003). End-to-end WAN Service Availability. *IEEE/ACM Transactions on Networking*, 11(2).

Didona, D., Fatourou, P., Guerraoui, R., Wang, J., and Zwaenepoel, W. (2019). Distributed Transactional Systems Cannot Be Fast. In *The ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, NY, USA. ACM Press.

Enes, V., Almeida, P. S., Baquero, C., and Leitao, J. (2019). Efficient Synchronization of State-Based CRDTs. In *International Conference on Data Engineering*, ICDE, pages 148–159. IEEE Computer Society.

Gilbert, S. and Lynch, N. A. (2004). Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. In *Principles Of Distributed Computing*, PODC.

Gotsman, A., Lefort, A., and Chockler, G. (2019). White-box Atomic Multicast. In *International Conference on Dependable Systems and Networks*, DSN. IEEE.

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492.

ISO/IEC (2020). ISO 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. Accessed 2020-09-12.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169.

Ongaro, D. and Ousterhout, J. K. (2014). In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*.

Papadimitriou, C. H. (1979). The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653.

Rohrer, J. P., Jabbar, A., and Sterbenz, J. P. (2014). Path diversification for future internet end-to-end resilience and survivability. *Telecommunication Systems*, 56(1):49–67.

Rothnie, J. B. and Goodman, N. (1977). A Survey of Research and Development in Distributed Database Management. In *International Conference Conference on Very Large Data Bases*.

Schneider, F. B., Gries, D., and Schlichting, R. D. (1984). Fault-tolerant broadcasts. *Science of Computer Programming*, 4(1):1–15.

Shapiro, M., Pregui, N., Baquero, C., and Zawirski, M. (2011). A Comprehensive Study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, Inria – Centre Paris-Rocquencourt.

Shaw, M. (2001). The Coming-of-Age of Software Architecture Research. In *International Conference on Software Engineering*, ICSE. IEEE.

Skrzypczak, J., Schintke, F., and Schütt, T. (2019). Linearizable State Machine Replication of State-based CRDTs without Logs. In *Symposium on Principles of Distributed Computing*, PODC. ACM.

Urbán, P., Défago, X., and Schiper, A. (2000). Contention-Aware Metrics for Distributed Algorithms: Comparison of Atomic Broadcast Algorithms. In *International Conference on Computer Communications and Networks*, IC3N. IEEE.

Vass, B., Tapolcai, J., Hay, D., Oostenbrink, J., and Kuipers, F. (2020). How to model and enumerate geographically correlated failure events in communication networks. In *Guide to Disaster-Resilient Communication Networks*, pages 87–115. Springer.

Younes, G., Shoker, A., Almeida, P. S., and Baquero, C. (2016). Integration Challenges of Pure Operation-Based CRDTs in Redis. In *Workshop on Programming Models and Languages for Distributed Computing*, PMLDC, New York, NY, USA. Association for Computing Machinery.