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*
last@braxo.se, first.last@mdu.se

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 not really required in systems only having fully independent CRDT objects. We identified a set of use cases related to resource usage such as text messaging where there is a need for a replication mechanism of CRDTs with lower code complexity and network usage 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, which can be batched as needed. When there are no updates, it uses no network resources at all. Furthermore, it is more resilient to server failures than the state-of-the-art solutions, as new values are available to the other nodes directly 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. The payment can either be made in advance, or afterwards, based on the actual resource consumption for the past billing period. 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 maintain 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, Yousif, 2018]. 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 then to maintain accurate records of the resource consumption across all these servers.

One of the earliest works on database replication is RFC 677 [Johnson and Thomas, 1975], which uses increased reliability and efficiency of data access as the main motivations. To also achieve maximum availability, the authors conclude that

"... a completely general system must deal with the possibility of communication failures which cause the network to become partitioned into two or more sub-networks."

Given the current prevalence of cloud-based systems, both reliability, efficiency, and availability are just as important today. However, 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 coexist [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

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

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, as there is no need to wait for time-consuming network round-trips. The new state, or records of the performed operations, are instead regularly broadcast to the other nodes. As these reach them, all nodes eventually get the same value for the object. A straightforward example is the set of integers $\mathbb{Z}$ over the operation `max()`, but a CRDT can also be a collection, e.g., a set or a tree. There are two main groups of CRDTs:

State-based CRDTs send their full state [Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, 2017] between the servers. 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], thus reducing the network traffic.

Operation-based CRDTs send only the individual operations [Baquero et al., 2014, 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. For example, a log entry could be created every time a user's resource usage goes above a predefined limit. 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 both that all packets are delivered in the same order, and that this happens only if all nodes are still reachable. Atomic broadcast can be implemented in several ways, requiring a different number of messages sent over the network, and maxing out the CPU and network in different situations [Urbán et al., 2000]. A simple example is Skeen's algorithm, which requires a network packet to be returned 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 without any causal dependencies at all, 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 pair 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 of 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, similar to what Enes et al. [Enes et al., 2019] did for operation-based CRDTs. 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. In this work, the setting is *Method*, 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 exactly we mean by "better" in Section 2.3.

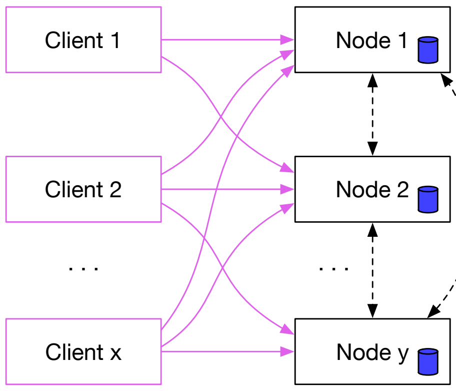

Figure 1: X clients connected to Y nodes, which in turn are all connected to each other. The nodes maintain a database of the credit balance for each client.

2.1 System Model

We assume we have a distributed system of independently running nodes, communicating over an asynchronous network. The network can use any topology, as long as there is at least one 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 or reordered. The nodes have local memory and a 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 then. 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, 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. Any negative credit balance can be adjusted afterwards. This allows us to make balance updates without first making a network round-trip to the other nodes to verify that the result would not be negative.

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. The lower the value of λ, the higher the probability for this is. If x is measured in seconds, we will have a repeating sequence of some number of seconds with updates, and some other number of seconds without. It would be beneficial if we could avoid sending any data during these seconds of silence, to lower both the amount of network traffic and the amount of local processing on each node.

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.

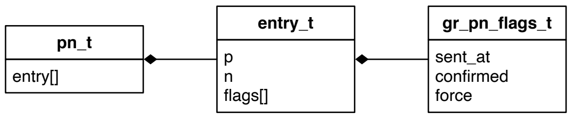

Figure 2: The three types used by CReDiT.

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, although each counter will be managed separately by CReDiT. 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 following fields. 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 data is 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 really should 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**". All functions perform all their steps as a single atomic operation, to allow them to be used directly in multi-threaded applications.

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

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

New: It clears the `entry[x].flags[*]` structures.

`update(delta)`
This is called on the local node A, updating the resource counter. It corresponds to *increment* for an ECDC [Almeida and Baquero, 2019], but allows both positive and negative modifications.

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

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 to the other nodes.

PN: It will execute the same logic for all nodes it knows about, and for all `entry_t` instances. The idea is that all nodes should get the full array of values on all nodes. It does not wait for any replies from the other nodes.

New: If the `force` flag is set, the entry is sent. Otherwise the entry is sent if the `confirmed` flag is not set, or if `sent_at` is different than the current time. Afterwards, `sent_at` is set to the current time, and `force` is set to false.

`receive(B, x, p, n)`
This function is called by the transport layer, when new data has been received by node B concerning values on node `x` (where `x` can be both A, B, or another node).

PN: The two fields `entry[x].p` and `entry[x].n` are updated to their respective maximum.

New: If the `entry[x].flags[B].confirmed` flag is set, `entry[x].flags[B].force` is set. If the incoming values differ from the local values in any way in the PN step, the `entry[x].flags[*].confirmed` flags are cleared. The `entry[x].flags[B].confirmed` flag is always set though, as we know that node B has these particular values. Finally it makes a callback to the application, which can persist the new data. This persisted data is what the application

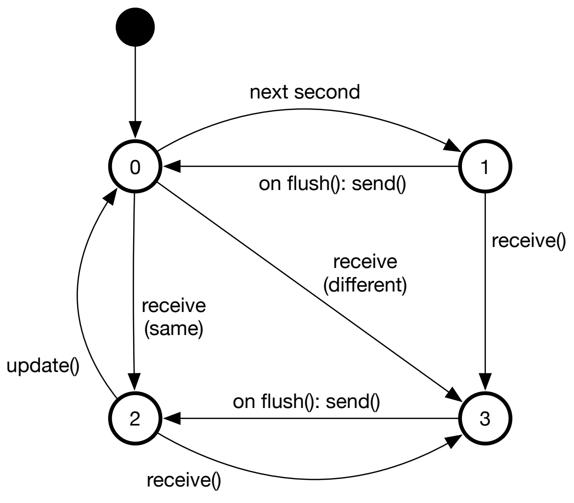

Figure 3: States on a node. Each pair of nodes has its own state. The black circle is the starting point. For the states 1 and 3, the `force` flag is set. For the states 2 and 3, the `confirmed` flag is set.

should provide to the `init()` function after being restarted.

The implementation was based on GeoRep [Brahneborg et al., 2020]. This provided us with 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 second (as returned from the `time(3)` system function or similar) 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 the functions not affecting a particular state are omitted for clarity.

3.3 Data Flow

Figure 4 shows the data flow between the two nodes A and B, following the algorithms in Section 3.1, each one making a single update to a shared counter. The steps are as follows.

1. The sequence begins by A updating the value of a new counter with $+2$. This creates the counter, and A sets p to 2 and n to 0 in `entry[A]`.
2. After at most one second, A moves B to state 2. On the next call to `flush()` from the upper application layer, the values for A are sent to B, after which A sets `entry[B].sent` to `now`.
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` = `now` for B and `force` is not `true`, any additional calls to `flush()` will not cause more data to be sent to B.
5. At B, it has `force` set to `true` for A, so the next time `flush()` is called, the pair p=2 and n=0 for A is sent back to A.
6. Next, A gets the $(2,0)$ 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 $(2,0)$ pair. No more data is sent by either side.
7. In the second part of the figure, B updates the value with -3, adding the pair $(0,3)$ for B.
8. At most one second later, B moves to state 1. Here, `flush()` sees that the pair $(0,3)$ does not have the `confirmed` flag set (for any host), and sends it to A.
9. A receives the $(0,3)$ pair, and sets `confirmed` and `force`, just as node B did earlier in step 3.
10. As `force` is set, the $(0,3)$ pair is sent back to B the next time A calls `flush()`. The pair $(2,0)$ for A still has the `confirmed` flag set, so it is not sent.
11. B receives the pair $(0,3)$ for B. As in step 6, this matches what it already has, so it sets `confirmed` but not `force`.
12. Again, both nodes have the same set of values, agreeing on the total sum. They also know that the other node has these exact values, so they are not sent again.

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, it will see the missing `confirmed` flag, and send the data again. As we assume *fair lossy* links as mentioned in Section 2.1, B will eventually receive this data.

If the reply 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 it to B again. B has the `confirmed` flag

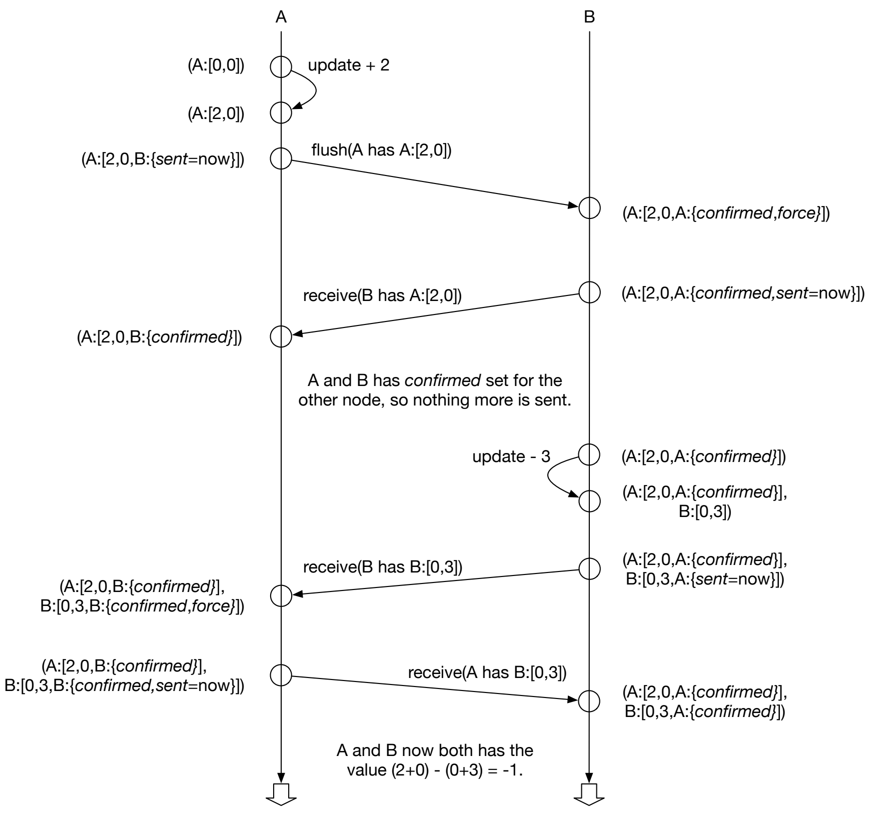
A
B
(A:[0,0])
update + 2
(A:[2,0])
(A:[2,0,B:{*sent*=now}])
flush(A has A:[2,0])
(A:[2,0,A:{*confirmed*,*force*}])
receive(B has A:[2,0])
(A:[2,0,A:{*confirmed*,*sent*=now}])
(A:[2,0,B:{*confirmed*}])
A and B has *confirmed* set for the other node, so nothing more is sent.
update - 3
(A:[2,0,A:{*confirmed*}])
(A:[2,0,A:{*confirmed*}], B:[0,3])
receive(B has B:[0,3])
(A:[2,0,A:{*confirmed*}], B:[0,3,A:{*sent*=now}])
(A:[2,0,B:{*confirmed*}], B:[0,3,B:{*confirmed*,*force*}])
(A:[2,0,B:{*confirmed*}], B:[0,3,B:{*confirmed*,*sent*=now}])
receive(A has B:[0,3])
(A:[2,0,A:{*confirmed*}], B:[0,3,A:{*confirmed*}])
A and B now both has the value (2+0) - (0+3) = -1.

Figure 4: Protocol running on nodes A and B. The current values for p, n, and the flags, are shown within brackets.

set, so it sets the `force` flag, causing the data to be sent back to A by the next call to `flush()`.

The combined effect of the `confirmed` and `force` flags is that any data packet can be lost, and the protocol will still recover, making sure all nodes have the same set of values, as well as having confirmed values from these nodes. After a period without updates, no more data will be sent until after the next call to `update()` for this counter.

4 EVALUATION

We here 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 (albeit limited by the availability of the server itself, which is beyond our control). Even though we in Section 2.3 state that we prioritize availability over correctness, we therefore do not evaluate the availability.

4.1 Functional Validation

We validated the functionality both manually and with a test program. The manual validation was done by first listing the most important events from the perspective of some node A, communicating with another node B:

1. Node A makes an update.
2. Node B makes an update.
3. After an update, a packet from node A to node B is lost.
4. After an update, a packet from node B to node A is lost.

We then manually examined all 16 possible combinations of whether each one of these events happened, making traces similar to the one in Figure 4. In an earlier version of the protocol, this revealed an issue when it could not properly recover from a lost packet, leading to the introduction of the `force` flag.

Next, we wrote a test program which was run on 3 nodes, on each one generating random sequences of updates and pauses as described in Section 2.3. The updates varied linearly between -2 and 2. It then verified that final value on all nodes were identical. We also manually inspected the generated log files, to verify that nothing unexpected occurred.

One such log file is shown below. It runs on the 3 nodes ams1, ams2, and ams3, each one with 2 client threads. After each update, the current value on the node is shown. Finally, after a few seconds of being idle, they print their final value, which here is 22.

```
test 2 clients
start on ams1 at 10.133.72.149
start on ams2 at 10.133.112.180
start on ams3 at 10.133.112.195
node 1 currently has the value 7
node 2 currently has the value 7
node 2 currently has the value 16
node 1 currently has the value 17
node 3 currently has the value 16
node 3 currently has the value 21
node 2 currently has the value 22
node 1 currently has the value 21
node 3 currently has the value 25
...
Final value on ams1:  22
Final value on ams2:  22
Final value on ams3:  22
```

4.2 Correctness Conditions

By construction, a state-based CRDT ensures that all updates originating on a particular node are done in the same order on all other nodes as well, 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, 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. We have no transaction log, so for a given n the memory requirement is constant.

If there has been an update on node A, up to 4 sets of network packets are triggered. After these steps, all n nodes will have the same value, as well as knowing that the other $n-1$ nodes have it too. Thanks to having this knowledge, no more data is sent until the next update is made.

1. Node A sends the updated (p, n) pair to the other $n-1$ nodes.
2. After receiving the new pair, these $n-1$ nodes send back their updated values. For a system with 2 nodes, as in Figure 4, no more packets are sent after this step.
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. Therefore `flush()` on these nodes will broadcast the updated value to the remaining $n-2$ nodes[1]
4. If a packet in the previous set is received from a node y on a node x before it 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. Furthermore, every lost packet results in two additional packets being sent.

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 of a particular counter. The number of updates also has no effect on the number of required network packets, making the quadratic scale-up above less of a problem than it may seem.

Additionally, counters with no updates on a particular node, after having its `confirmed` flag was set,

[1]Neither to A nor to itself.

stay in state 2 in Figure 3. In this state `flush()` causes no action, generating no network traffic at all.

4.4 Real-world Evaluation

There are a few seemingly obvious measurements that can be done to evaluate how the protocol behaves in real-world situations. First, we can measure the number of function calls per second. However, as all functions either just modify local data structures or are asynchronous, this would effectively measure just 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 just measures the round-trip time between the nodes, and if `flush()` is called on node A, it does not know when node B and node C have completed their communication between each other. Third, we could compare some performance aspect of the application that originally triggered this work. Currently, the best solution in that application is to use a replicated MySQL server, but we have not found a way to do the required multi-master replication with that database with geo-separated nodes, and still get acceptable performance (at least 100 updates per second, ideally closer to 1000).

We will instead compare our protocol with PN-counters based on atomic broadcast. In particular, we observed that for counters with updated data, most algorithms for atomic broadcast use fewer communication steps and network resources than CReDiT does. For other counters, CReDiT uses fewer. So, we want to measure the relative frequency between these two cases. From two production systems using the motivating application mentioned above, we got sample log files containing the time stamps of events that would update 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 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 could 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, e.g. Skeen's protocol, 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 messages. However, this is only true when there has been an update. For periods of time[2] without any updates, the protocol instead uses no communication at all.

The round-trip times between each pair of nodes, i.e. whether the nodes are running within the same data-center or are geographically separated, has little or no effect on this protocol, for several reasons. First, the updated data can be flushed at any suitable 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 directly available to each node after being received, a temporary delay on one link between two nodes only affects those two specific nodes. This also improves the reliability of the system, as there is a greater possibility for updated values just before a crash to be successfully received by the other nodes than when multiple communication steps are needed.

The described approach works for any state-based CRDT, as long as it is possible for `receive()` to determine if the incoming values differ from the local values.

In the first iterations of our protocol, we stored a copy of the data sent to each other node. This worked well, but would for larger CRDT objects lead to excessive memory usage. When these copies were replaced by the `confirmed` and `force` flags, the memory usage became both smaller and constant.

[2]The length of such a period depends on the resolution of the `sent_at` field.

6 RELATED WORK

Almeida et al. [Almeida et al., 2018] address a problem very similar to ours, presenting δ-CRDTs which support both duplicated network packets just as state-based CRDTs as well as achieving the lower bandwidth requirements of operation-based CRDTs. Their anti-entropy algorithm, corresponding to our `flush()`, sends just 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. We use the increased storage requirement for the `confirmed` flag to eventually not having to send any data at all.

One way to ensure that all servers has 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, Howard and Mortier, 2020] 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 not needed.

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] addressed the synchronization overhead of state machine replication by using a single network round-trip for updates and not having a leader, just as in this work. To get linearizability [Herlihy and Wing, 1990], their coordination is done by the query operations, using repeated round-trips until the returned values stabilize. In contrast, we can accept both updates and queries during all types of network partitions, and can respond to queries without any network round-trips.

Using Gossip also reduces the network usage, but increases the number of communication steps until all other nodes have the most recent data.

7 CONCLUSIONS

Generally, layered architectures are of course 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 can use unnecessary network and CPU resources. By instead taking advantage of the lack of causality between the operations of our CRDT counters, we could create a new protocol with lower network requirements during periods without any updates. Additionally, its constant memory usage makes it suitable for use in embedded devices and similar systems with limited resources.

ACKNOWLEDGMENTS

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

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 Proceedings – 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. (2014). Making operation-based crdts operation-based. In Proceedings – IFIP International Conference on Distributed Applications and Interoperable Systems, DAIS. Springer.

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 Proceedings – 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., Caušević, A., and Björkman, M. (2020). Superlinear and Bandwidth Friendly

Geo-replication for Store-and-forward Systems. In Proceedings - International Conference on Software Technologies, ICSOFT.

Brewer, E. A. (2000). Towards Robust Distributed Systems. In Principles Of Distributed Computing. 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):300–313.

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

Enes, V., Almeida, P. S., Baquero, C., and Leitao, J. (2019). Efficient Synchronization of State-Based CRDTs. In Proceedings – Int'l 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 Proceedings – 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.

Howard, H. and Mortier, R. (2020). Paxos vs Raft: Have we reached consensus on distributed consensus? In Proceedings — Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC.

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

Johnson, P. R. and Thomas, R. H. (1975). RFC 677 – The Maintenance of Duplicate Databases.

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 Proc. – Int'l 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 Proc. – Symp. 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 Proceedings – 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 Proceedings – Workshop on Programming Models and Languages for Distributed Computing, PMLDC, New York, NY, USA. Association for Computing Machinery.

Yousif, M. (2018). Cloud Computing Reliability – Failure is an Option. IEEE Cloud Computing, 5(3):4–5.