Delta-State-Based Synchronization of CRDTs in Opportunistic Networks

Frédéric Guidec[1,2], Yves Mahéo[1,2], Camille Noûs[2]
1 IRISA, Université Bretagne Sud, France
2 Laboratoire Cogitamus, France

Abstract—**Conflict-Free Replicated Data Types (CRDTs) are distributed data types that support optimistic replication: replicas can be updated locally, and updates propagate asynchronously among replicas, so consistency is eventually obtained. This ability to tolerate asynchronous communication makes them ideal candidates to serve as software building blocks in opportunistic networks (OppNets), that is, mobile networks in which the dissemination of information can only depend on unpredicted transient radio contacts between pairs of nodes. In this paper we investigate the problem of implementing CRDTs in an OppNet, and we propose a delta-state-based algorithm to solve this problem. Experimental results confirm that this algorithm ensures the synchronization of CRDT replicas in an OppNet, and that it outperforms a pure state-based synchronization algorithm when dealing with container CRDTs.**

I. Introduction

Conflict-free Replicated Data Types (CRDTs) are used in distributed system when eventual consistency of replicated data is sufficient [1], [2], allowing replicas to diverge temporarily while ensuring that they will eventually be reconciled into the same state. Any replica of a CRDT can be updated locally, without any coordination with other replicas. Information about each update is passed asynchronously to the other replicas via a synchronization protocol. When all replicas have received the same set of updates they reach the same state.

CRDTs can be implemented as either operation-based CRDTs, or state-based CRDTs. In an operation-based CRDT, whenever an operation (update) is performed on a replica, this operation is embedded in a message and sent to all other replicas, which can then update their own state accordingly. In a state-based CRDT, an operation is only applied to the local replica's state. Each replica periodically synchronizes with other replicas by sending them its entire state. Upon receiving the state of another replica, the receiver merges its local state with the received state, using a function that deterministically computes the join (least upper bound) of both states.

Unlike operation-based CRDTs, state-based CRDTs do not require that each update be sent to all other replicas. It is only needed that each replica synchronizes sufficiently often with a few other replicas. Eventual consistency is ensured as long as the synchronization graph is connected [2]. Shipping entire states between replicas can yield a major communication overhead, though. Delta-state CRDTs (or delta CRDTs for short) have been proposed as a means to reduce this overhead by shipping only partial information about the sender's state (typically, a representation of the effect of recent update operations on the state) [3], [4].

CRDTs can be used in distributed systems in which nodes may crash and network partitions may occur. Replicas will however ultimately converge, provided crashed nodes eventually recover, and partitions heal. Opportunistic networks (OppNets) are typically networks that exhibit such characteristics. An OppNet is a network whose nodes are mostly mobile, and that operates solely by exploiting transient direct radio contacts between pairs of nodes [5].

Many message forwarding protocols have been designed specifically to operate in OppNets, based on the "store, carry, and forward" principle: each mobile node can serve as a "data mule" for messages it has either produced itself or received recently, storing these messages in a local cache, and carrying them for a while before they can be forwarded to other nodes. Implementing operation-based CRDTs based on this model is rather straightforward. Each operation performed on a replica can be embedded in a message, which is then broadcast using for example an epidemic forwarding protocol. But broadcasting many small messages network-wide using the store, carry, and forward approach is resource consuming, since each node is expected to store as many messages as possible in its local cache, for as long as possible, in order to maximize their propagation in the network.

In this paper we investigate the implementation of state-based CRDTs (more specifically, delta-state based CRDTs) in OppNets. Unlike operation-based CRDTs, state-based CRDTs do not require broadcasting each update network-wide. Instead each contact between two nodes can be exploited as an opportunity for these nodes to synchronize pair-wise. Since the messages are exchanged only between neighbor nodes while they are in radio contact, they are not meant to propagate network-wide, so no forwarding protocol is required and there is thus no need for each node to maintain a message cache. The only information that needs to be maintained over time is already contained in each replica's state, and this is sufficient to synchronize replicas. State-based synchronization requires that entire states be exchanged between replicas, though. For large CRDTs this approach can yield significant communication overhead. In this paper we focus on delta-state-based synchronization, which consists in shipping only partial states (called deltas) between replicas whenever possible [3], [6], [7].

II. System model

As a general rule, in an OppNet, interactions are based solely on unplanned and transient pair-wise radio contacts between neighbor devices. So any opportunistic interaction protocol must tolerate communication disruptions. The algorithm defined in the next section is meant to run on top of a basic communication layer, whose characteristics are detailed below. This communication layer only allows a mobile device to exchange messages with its direct neighbors. We therefore do not assume that a network-wide message routing or dissemination protocol is implemented in the network.

Event *new_neighbor()* is raised by the communication layer whenever a radio contact is established with a new neighbor, and function *current_neighbors()* can be called to get a list of the current neighbors. The way neighbor discovery is actually performed in the communication layer depends on the characteristics of the underlying transmission technology (e.g., Bluetooth or Wi-Fi in ad hoc mode).

Function *send()* is used to send a message to a neighbor. This transmission may fail for different reasons, for example if the targeted neighbor has just moved out of transmission range. In any case function *send()* does not return a status, since we do not assume that transmission failures can be detected.

Event *receive()* is raised by the networking layer when a new message has been received from a neighbor. Corrupted messages, if any, are assumed to be discarded by the communication layer.

III. A Delta-state-based-synchronization algorithm

A synchronization algorithm designed for OppNets must use radio contacts between mobile nodes as opportunities for these nodes to synchronize the CRDT replicas they hold. A pure state-based synchronization protocol can be sufficient for CRDTs that don't grow much, such as counters and registers, but for container-like CRDTs that can aggregate large amounts of data, such as sets or maps, it is preferable to rely on a synchronization protocol that only ships partial information about a replica's state whenever possible. The algorithm we present below ships either digests, deltas (partial states), or full states depending on circumstances in order to achieve the synchronization of replicas. This algorithm is presented in two flavors: one for causal CRDTs (whose implementation requires maintaining some kind of causal context in a replica's metadata), and another one for non-causal CRDTs, for it appears that the very same algorithm could not be used for both kinds of CRDTs.

A number of functions must be defined in order to process the replica on each node. The signatures of these functions are presented in Alg.1. How these functions are implemented depends on the actual kind of CRDT considered.

Function *merge()* must be used to merge the state of the local replica with the (full or delta) state received from a neighbor. Function *get_digest()* must return the digest of a replica's state. This digest is assumed to summarize this state in a very compact form, so its transmission should be far less expensive than that of the full state. For non-causal CRDTs (e.g., GO-Counters, GO-Sets), the digest may typically be defined as a simple hash code. For causal CRDTs, the digest can be defined as a version vector. Such a digest makes it possible to determine what is missing in each replica or, more formally, what part of a replica's state would be required to strictly inflate the other replica's state. Function *generate_delta()* will be executed whenever an update (more formally, a mutator) is applied to the local replica. This function returns a delta that expresses the effect of the mutation. This delta can be sent to another replica, to be merged there with its own local state. Function *get_missing()* compares two replica states s_1 and s_2, and determines what part of s_2 would be required to strictly inflate s_1. This function returns a delta state, which captures the data required to inflate s_1 accordingly, or $\perp$ (bottom) if there is no way to inflate s_1. Note that unless s_1 and s_2 are equal, *get_missing*(s_1, s_2) and *get_missing*(s_2, s_1) will always return different results. Function *get_missing()* can also be used when the digest of another replica has been received. When this function is implemented for a non-causal CRDT (whose digest is only defined as a hash code), it either returns $\perp$ (bottom) if both digests are equal, or s_2 if they are different. For a causal CRDT (whose digest is defined as a version vector), the function determines based on d_1 and s_2 what part of s_2 would strictly inflate the other replica.

Any CRDT for which these functions are implemented is actually a Δ-CRDT [4], that is, a CRDT whose replication can be obtained by propagating a delta (Δ) of the current state that is missing in another replica.

Algorithm 1 Functions for delta state-based synchronization

```
def ID_T: String        // or MACaddr, or IMEI, etc.
def DIGEST_T: xxx   // Hashcode or Version_Vector

static ID_T self.id ↩ oppnet_id()
REPLICA_T self.state ↩ ⊥

function merge(REPLICA_T t1, REPLICA_T t2): REPLICA_T
function get_digest(REPLICA_T t): DIGEST_T
function generate_delta(MUTATOR m): REPLICA_T
function get_missing(REPLICA_T s1, REPLICA_T s2): REPLICA_T
function get_missing(DIGEST_T d1, REPLICA_T s2): REPLICA_T
```

The delta-state-based synchronization algorithm we propose is presented in Alg.2. Red code applies only for a non-causal CRDT, and blue code applies only for a causal CRDT.

When two nodes get into contact, one of the nodes sends its current digest to the peer node (lines 01–02). When a host receives a digest from a neighbor (line 03), function *get_missing()* is invoked to compare the received digest to the local digest, and determine if part of the local state can be sent to the neighbor.

The way function *get_missing()* behaves depends on whether the CRDT considered relies on causal context or not. For a non-causal CRDT, whose digest is only a hash code, comparing the received and local hash codes only allows to determine if they are equal or different (lines 37–39). If they are equal, then the

Algorithm 2 Delta-state-based (Δ-SB) synchronization algorithm [Red code: non-causal CRDT, blue code: causal CRDT]

```
01  upon new_neighbor(neigh_id) do
02    if (self.id < neigh_id) then send(neigh_id, self.digest)

03  upon receive(neigh_id, neigh_digest) do
04    $D \hookleftarrow$ get_missing(neigh_digest, self.state)
05    if ($D \neq \perp$) then send(neigh_id, $D$)
06    if (neigh_digest after self.digest) then
07      send(neigh_id, self.digest)
08    fi

09  function disseminate(targets, output):
10    forall id in targets do
11      send(id, output)
12    done

13  function process_input(neigh_id, input)
14    // input may be either a full state or a $\Delta$
15    $\Delta_{in} \hookleftarrow$ get_missing(input, self.state)
16    if ($\Delta_{in} \neq \perp$) then
17       self.state $\hookleftarrow$ merge(self.state, $\Delta_{in}$)
18       self.digest $\hookleftarrow$ get_digest(self.state)
19       targets = current_neighbors() \ neigh_id
20       disseminate(targets, $\Delta_{in}$)
21    fi

22  upon receive(neigh_id, neigh_delta) do
23    process_input(neigh_id, neigh_delta)

24  upon receive(neigh_id, neigh_state) do
25    neigh_digest $\hookleftarrow$ get_digest(neigh_state)
26    $\Delta_{out}$ $\hookleftarrow$ get_missing(neigh_digest, self.state)
27    if ($\Delta_{out} \neq \perp$) then send(neigh_id, $\Delta_{out}$)
28    process_input(neigh_id, neigh_state)

29  upon update(m) do
30    self.digest $\hookleftarrow$ get_digest(self.state)
31    $\Delta_{out}$ $\hookleftarrow$ generate_delta(m)
32    disseminate(current_neighbors(), $\Delta_{out}$)

33  function get_missing(CRDT t1, CRDT t2): $M$
34    $M \hookleftarrow \{ m \in$ t2.state, $m \notin$ t1.state $\}$
35    $M \hookleftarrow$ get_missing(t1.state.digest, t2)

36  function get_missing(DIGEST_T d1, CRDT t2): $M$
37    if (d1$\neq$ t2.state.digest)
38    then $M \hookleftarrow$ t2.state
39    else $M \hookleftarrow \perp$
40    $M \hookleftarrow \{ m \in$ t2.state, $m$.timestamp > d1 $\}$
```

state of the remote host is presumed to be equal to the local state, in which case function *get_missing()* returns $\perp$ (bottom) and nothing is sent to the peer host. If both digests are different, then there is no way to determine exactly what part of the local state is missing on the remote peer, so function *get_missing()* returns the full state of the local replica, which is then sent to the remote peer. For a causal CRDT, function *get_missing()* can determine exactly what is missing in the remote host, and return a delta accordingly (line 40). Besides, if the local version vector is late compared to that of the peer node, then the local digest must be sent to the peer as well (lines 06–08).

Upon receiving a full state (line 28), function *get_missing()* is invoked to compare this state (whose digest is assumed to be embedded in the state's metadata) to the local digest, and thus determine if part of the local state can be sent back to the peer host (line 27). Function *process_input()* is then invoked to determine what part of the received state can be merged with the local state. Function *get_missing()* is therefore invoked again (line 15), but this time it is to determine what can be gained on the local host, rather than what is missing on the remote host. The actual gain, if any, is merged with the local state, and function *disseminate()* is invoked to send this data to all the current neighbors of the local host, except the one from which new data has just been received (thus avoiding the retro-propagation of information between replicas).

Note that a host may be connected to several peers simultaneously. In a connected component of the graph, information can therefore disseminate rapidly, rather than propagate only when new contacts occur. Relaying what has just been received from one neighbor to all other neighbors makes sense in an OppNet, where contacts are transient and can be broken at any time: forwarding deltas transitively is a way to speed up their dissemination. A side-effect of this approach is that a host may receive the same input several times from distinct neighbors. This is the reason why function *get_missing()* is systematically invoked when processing an input (line 15), so as to discard any redundant information.

Upon receiving a delta state (line 22), function *process_input()* is invoked directly, since there is no way for the receiver to determine what may be missing on its neighbor based on a delta state: this is only possible when receiving either a digest or full state.

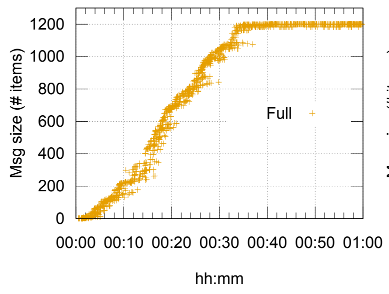
Finally, whenever an update operation occurs on the local replica (line 29), a delta is generated based on the operation (formally, the mutator) m that has been applied locally, and this delta is sent immediately to all the current peers of the local host. Note that the local digest is adjusted whenever the local state is changed, that is, when an update is applied locally (line 30), or when the local state is merged with another state (line 18).

IV. Experimentation

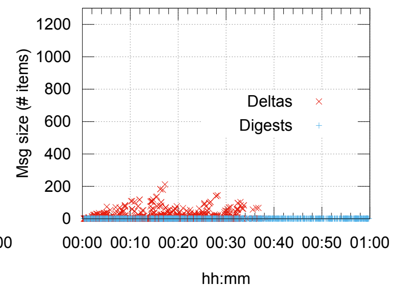
We describe in the following one of the experiments we conducted with the LEPTON emulation platform [8] in order to observe how our delta-state-based synchronization algorithm-can perform in realistic conditions.

Mobility and application scenario: We consider a population of 20 nodes moving in a $200m \times 200m$ area, according to a Levy walk model. The node speed varies between 1 and 2 m/s, the relation between a flight length l (0 to 100 m) and its duration Δt_f is $\Delta t_f = k.l^{1-\rho}$, with $k = 30.55$ and $\rho = 0.89$. The duration of each pause is chosen randomly (with a uniform distribution) between 0 and 10 s, the radio range is of 30 m.

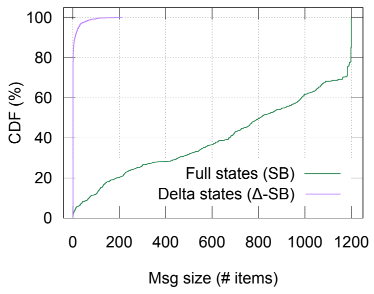
The CRDT involved in the application scenario is an AW-Set (Add Wins Set), a distributed set to which items can either be added or removed [2]. An AW-Set is a causal CRDT, and its digest can be expressed as a version vector. This makes it possible to determine which events in a replica's state occurred before, after, or concurrently with the events captured in another replica's state (whose version vector is known). The application



(a) SB algorithm



(b) Δ-SB algorithm



(c) CDF of message size

Metrics	**SB**	**Δ-SB**
# digests	-	1201
# full states	1415	-
# delta states	-	4736
Total	1415	5937

(d) Number of messages

Figure 1. AW-Set synchronization

timeline is split in two phases. During phase I, a new item is added by each node to the AW-Set every minute and is removed 90 seconds later. Phase I continues for 30 minutes, afterwards all nodes enter phase II, during which they do not update their local replica anymore. Phase II is meant to verify that all replicas converge eventually, and that the network traffic observed between neighbor nodes varies accordingly.

Synchronization algorithms: We implemented a pure state-based (SB) algorithm, as well as the Delta State-Based (Δ-SB) synchronization algorithm presented in Section III. The SB algorithm is meant to serve as a baseline. It works as follows: whenever a new contact is established between two nodes, one of these nodes sends its full state to the peer node. Upon receiving this input, the receiver compares it with its own local state, and if both states are different it sends its own state to the peer node. Both received states are of course merged with the local state.

Results: Figure 1 presents the results observed when running the AW-Set scenario. In this figure the size of a message exchanged between two nodes is expressed in terms of items contained in the message's payload. This is because an AW-Set can be used to store all kinds of items (e.g., numerical values, character strings, structured types). In any case the size of a message transporting a full state or delta state is roughly proportional to the number of items in this state, including the associated metadata if needed. The drawback of using the SB algorithm is apparent: the size of the messages exchanged by neighbor nodes grows rapidly and reaches a plateau once all replicas have converged (Fig. 1.a), whereas with the Δ-SB algorithm (Fig. 1.b), only digests and deltas are shipped. Since the digests exchanged by neighbor nodes are version vectors, it is possible to avoid shipping full states altogether. The table in Fig. 1.d and the CDF distributions presented in Fig. 1.c confirm that although the number of messages exchanged with Δ-SB is larger than with SB, these messages are a lot smaller, so the overall communication load is reduced significantly with the delta-state-based approach.

V. Conclusion

In this paper we have addressed the problem of synchronizing CRDT (Conflict-free Replicated Data Type) replicas in an opportunistic network (OppNet), leveraging transient contacts between mobile nodes to synchronize the replicas maintained on these nodes. The synchronization algorithm we have proposed uses a delta-state-based approach, using messages containing either state digests, delta states, or full states in order to mitigate the overhead of always shipping full states, as is commonly achieved in pure state-based synchronization protocols. This makes it effective for the synchronization of container-like CRDTs such as sets, lists, maps, graphs, etc. Experimental results produced by running this algorithm to synchronize Add Wins Sets in an emulated opportunistic networking setting confirm that it outperforms a pure state-based synchronization algorithm, while ensuring the same convergence of all replicas.

Funding

This work was supported by the French ANR (Agence Nationale de la Recherche) under grant number ANR-16-CE25-0005-02.

References

[1] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A Comprehensive Study of Convergent and Commutative Replicated Data Types," INRIA, Tech. Rep. 7506, Jan. 2011.

[2] N. Preguiça, "Conflict-free Replicated Data Types: an Overview," *Arxiv Preprint* https://arxiv.org/abs/1806.10254, 2018.

[3] P. S. Almeida, A. Shoker, and C. Baquero, "Efficient State-Based CRDTs by Delta-Mutation," in *Networked Systems*. Springer, 2015, pp. 62–76.

[4] A. van der Linde, J. a. Leitão, and N. Preguiça, "Δ-CRDTs: Making δ-CRDTs delta-based," in *2nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC 2016)*. London, United Kingdom: ACM, Apr. 2016.

[5] L. Pelusi, A. Passarella, and M. Conti, "Opportunistic Networking: Data Forwarding in Disconnected Mobile Ad Hoc Networks," *IEEE Communications Magazine*, vol. 44, no. 11, pp. 134–141, Nov. 2006.

[6] P. S. Almeida, A. Shoker, and C. Baquero, "Delta state replicated data types," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 162–173, 2018.

[7] V. Enes, P. S. Almeida, C. Baquero, and J. a. Leitão, "Efficient Synchronization of State-Based CRDTs," in *35th International Conference on Data Engineering (ICDE'19)*. Paris, France: IEEE, Apr. 2019, pp. 148–159.

[8] A. Sánchez-Carmona, F. Guidec, P. Launay, Y. Mahéo, and S. Robles, "Filling in the missing link between simulation and application in opportunistic networking," *Journal of Systems and Software*, vol. 142, pp. 57–72, Aug. 2018.