Future Generation Computer Systems 141 (2023) 755–767

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

PS-CRDTs: CRDTs in highly volatile environments

António Barreto, Hervé Paulino *, João A. Silva, Nuno Preguiça

NOVA Laboratory for Computer Science and Informatics (NOVA LINCS), Computer Science Department, NOVA School of Science & Technology, NOVA University Lisbon, Portugal

ARTICLE INFO

Article history:
Received 24 June 2022
Received in revised form 4 November 2022
Accepted 9 December 2022
Available online 21 December 2022

Keywords:
CRDTs
Consistency
Mobile computing
Publish/subscribe
Edge computing
Replication

ABSTRACT

The implementation of collaborative applications in highly volatile environments, such as the ones composed of mobile devices, requires low coordination mechanisms. The replication without coordination semantics of Conflict-Free Replicated Data Types (CRDTs) makes them a natural solution for these execution contexts. However, the current CRDT models require each replica to know all other replicas beforehand or to discover them on-the-fly. Such solutions are not compatible with the dynamic ingress and egress of nodes in volatile environments. To cope with this limitation, we propose the Publish/Subscribe Conflict-Free Replicated Data Type (PS-CRDT) model that combines CRDTs with the publish/subscribe interaction model, and, with that, enable the spatial and temporal decoupling of update propagation. We implemented PS-CRDTs in *Thyme*, a reactive storage system for mobile edge computing. Our experimental results show that PS-CRDTs require less communication than other CRDT-based solutions in volatile environments.

© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

Smartphones (and smart devices, in general), currently play an important role in peoples' everyday life, being used for everything, from leisure to work-related activities. To meet the users' expectations concerning availability and latency, while coping with the ever-increasing number of users and data, mobile-tailored applications and services, such as online gaming, social networks or collaborative editing, are increasingly adopting the edge computing paradigm.

Efficient mobile data management ultimately requires replication techniques [1]. In this context, when faced with the restrictions stated in the CAP theorem [2] and the poor or intermittent connectivity of mobile environments (which often cause temporary disconnections), the choice between compromising availability or relaxing consistency usually falls on the latter. However, relaxed consistency raises the challenge of how to handle and resolve data inconsistencies and conflicts.

Conflict-Free Replicated Data Types (CRDTs) [3,4] are replicated objects that can be concurrently modified without expensive synchronization, while guaranteeing the eventual convergence of all replicas. They, hence, present themselves as a natural solution to achieve relaxed consistency with low coordination in distributed environments. In fact, CRDTs have shown to be suitable for storage systems, such as Riak[1] and AntidoteDB,[2] collaborative applications [5–7] and peer-to-peer services [8].

Fig. 1 showcases an *edge computing* scenario, on which multiple (potentially mobile) devices create and share mutable data, in the form of CRDT objects. This data is stored in a cloud infrastructure that runs a CRDT-based database and is made available to the devices via a set of edge servers that cache some of the database's contents. In order to allow for disconnected operation [9], CRDT replicas may be installed on the devices themselves, synchronizing with the remainder replicas periodically, whenever a network connection is available. Accordingly, in this scenario, a CRDT object may have replicas in the cloud database, edge servers and mobile devices. To improve efficiency, replica synchronization can be carried out hierarchically, *i.e.*, co-located mobile devices and the local edge server can synchronize the ongoing modifications over the shared objects among themselves. The local edge server may then compute syntheses of these modifications and propagate them to the centralized database and all interested peer edge servers, enabling system-wide synchronization. The synchronization at the co-located mobile device level can also be applied to scenarios without a centralized database, and even without an edge server, being the data local to the devices sharing it. Example application fields of such solution are local multiplayer games, such as a chess or a card game, on which users play without centralized servers, or IoT scenarios, where users interact with IoT devices only based on locally available information.

* Corresponding author.
E-mail address: herve.paulino@fct.unl.pt (H. Paulino).

1 http://basho.com/products/riak-kv

2 https://www.antidotedb.eu

https://doi.org/10.1016/j.future.2022.12.013
0167-739X/© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

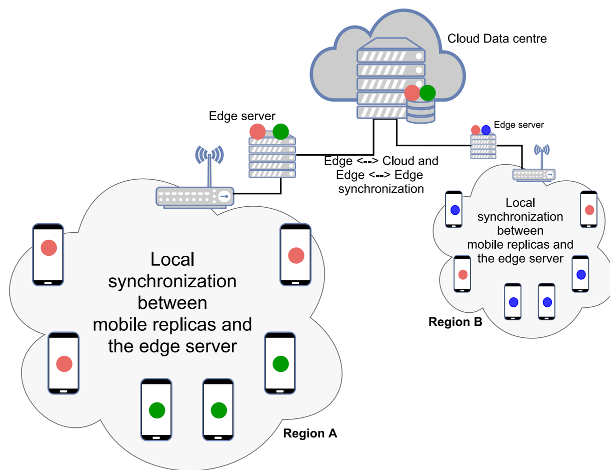


Fig. 1. Example of the use of local replica synchronization. The circles represent replicas. Red and green are stored in the cloud database, being the former replicated in more than one region and the latter replicated in a single one. The blue item is local to the devices of region B, being cached at the edge.

Current CRDT models cannot be efficiently used in the aforementioned scenarios because they require replicas to have knowledge of the peers they must synchronize with, before emitting an update. In static contexts, this information is known beforehand but, when such is not the case, active updating, flooding or some other out-of-band mechanism must be used [10]. In highly volatile mobile environments, as the ones comprising mobile devices, the inclusion of this *a priori* knowledge requirement imposes unacceptable communication costs. The alternative is to centralize the synchronization in some node, like the edge server or another node elected for the job. Each node becomes then aware of only two replicas: its own and the server's, inducing a solution that requires a central server, with all the known limitations concerning fault tolerance, load balance, performance, and so forth.

To address this limitation, we propose the Publish/Subscribe Conflict-Free Replicated Data Type (PS-CRDT) model. PS-CRDTs combine CRDTs with the publish/subscribe (P/S) interaction paradigm, leveraging on the latter to bring the spatial and temporal decoupling of CRDTs update propagation. Basically, we use the P/S system as the medium for propagating and sharing updates, and define the update propagation pattern for shared CRDT objects. With this, CRDT update dissemination is completely decoupled from update reception, enabling the use of CRDTs in highly volatile scenarios where there is no knowledge of all replicas. We also propose PS-CRDT-tailored versions of the main CRDT synchronization models (*state*-, *operation*- and *delta-based*), focusing on the adaptations needed to meet the requirements of each model.

To assess the practicality of our proposal, we implemented the proposed PS-CRDT synchronization models in the context of *Thyme* [11,12], a reactive storage system for mobile edge computing, and developed several CRDTs (*e.g.*, counters, sets and maps). The addition of CRDTs to *Thyme* provides a mean to use the P/S pattern to share and update mutable data in mobile edge environments. We used our implementation to conduct a comprehensive set of simulations, whose results show that PS-CRDTs require less bandwidth than other CRDT-based solutions to synchronize replicas in volatile environments.

In summary, our main contributions are:

1. the proposal of PS-CRDTs, a novel CRDT model that combines CRDTs with the P/S interaction paradigm, and the specification of three PS-CRDT-tailored synchronization models (Section 3);
2. the implementation of PS-CRDTs and the proposed three synchronization models atop *Thyme* (Section 4); and
3. a comprehensive experimental evaluation, through simulation, that compares our *Thyme* PS-CRDT against other solutions, such as Δ-CRDTs and CRDT-based centralized solutions (Section 5).

2. Background and related work

In this section we provide background on CRDTs, including the latest developments on the field, and also look at existent solutions for weak consistency in mobile environments.

2.1. CRDTs

CRDTs are a family of replicated data structures designed for highly available systems. They enable replicas to be updated independently and concurrently without coordination. CRDTs come in two (main) flavors, both providing strong eventual consistency [4]: *state-based* and *operation-based*.

State-based CRDTs synchronize replicas by periodically exchanging their local state, along with the necessary metadata information. When received, a state is merged with the local one by a commutative, associative and idempotent function, defined in the data type itself. These are generally the simplest type of CRDT to support, since they only require some kind of gossip protocol from the communication substrate. Their main drawback is the considerable communication overhead imposed by the propagation of the entire state (resulting from every know update, both local or remote) to every other replica. The work in [13] addresses the efficient dissemination of state-based CRDTs in large open networks. The work shares affinities with ours, in the sense that it also addresses dynamic environments. It is, however, directed to large global networks, while ours focuses on co-located nodes, and only addresses state-based synchronization, while we also support other synchronization models.

Operation-based (Op-based) CRDTs synchronize replicas by propagating state update operations, rather than the state itself. The process follows a two-step procedure: function *prepare* executes at the source to generate the operation (*op*) to be disseminated, and a commutative *effect* function executes at the destination to apply *op* over the local state. Op-based CRDTs are generally more efficient in terms of communication (i.e., bandwidth) but require stronger guarantees from the communication substrate: operations must be delivered with exactly-once semantics and obey the causal order.

Op-based CRDTs are usually simpler to implement than state-based. The designs are, however, compatible in the sense that they can be implemented atop each other. Pure op-based CRDTs [14] are a variant of op-based that limits state introspection in the *prepare* stage, with the goal of simplifying datatype implementation. The approach clearly separates op-based from state-based solutions, thus breaking the aforementioned compatibility.

Lastly, *Delta-based CRDTs* [10,15] combine aspects from both the state- and op-based designs. They are optimized state-based solutions on which only the changes made since the last synchronization moment (the delta) are propagated to the other replicas. The entire state is only communicated when there is a large enough amount of changes to justify it. Of the two, the $\Delta - based$ solution [10] is the one that better fits the type of environments we are targeting.

The work in [16] addresses the optimization of state transfers in delta-based CRDTs by: 1. avoiding the back propagation (BP) of data to the source node and 2. removing redundant state (RR) in delta propagation, *i.e.*, not sending the same information to the same node more than once.

2.2. Weak consistency in mobile environments

Throughout the years, weak consistency in mobile environments has mostly been addressed with the purpose of enabling disconnected execution. Systems such as Bayou [17], Rover [18], IceCube [19] and Telex [20], among others, leverage replication (mostly in the scope of caching) to allow clients to use remote services even when disconnected from the servers. Upon reconnection, the local and remote states are reconciled, being different strategies proposed for the purpose. SwiftCloud [21] follows the same philosophy, but is closer to our work by being a CRDT-based geo-replicated cloud storage system. To allow for disconnected execution, clients may cache CRDTs. Transactions (that never fail) are first executed and committed on the client side, and later propagated to the data centers. There is no inter-client interaction, clients always communicate with the closest data center.

PathStore [22] is a hierarchical storage system tailored for edge environments that provides eventual consistency guarantees. Replica updates are applied at each node of the hierarchy and propagated upward. There is, however, no local synchronization among the devices below the edge servers. All synchronization is performed by the local edge server. The work was extended in [23] to support session guarantees.

The topic of maintaining consistency between replicas in edge-cloud environments is addressed in other works, such as [24] that ensures strong consistency by leveraging on the Fast-Paxos algorithm. Other recent works have focused on end-to-end data consistency for mobile applications. Of these, we highlight Simba [25] that provides a high-level data abstraction with tunable consistency semantics. There are also works, such as Legion [8], that aim to reduce latency by promoting local replication and interaction as an alternative to continuous communication to centralized services. These works do not, however, address the volatility of networks of mobile devices.

3. The PS-CRDT Model

The PS-CRDT model brings the spatial and temporal decoupling of publish/subscribe interaction to CRDTs, enabling update dissemination to occur without the source knowing who, and where, the recipients are. We leverage the publish/subscribe model to notify all interested replicas that an update is available. Upon reception of such notification, it is up to the subscribing replicas to interact with the source to transfer the actual update.

We are considering an asynchronous system model composed of (potentially mobile) devices, that we refer to as *nodes*. Nodes have globally unique identifiers, no mobility restrictions, and may enter and/or leave the system at any time, *i.e.*, their absence may be permanent or temporary. A node's exit may be caused by a crash, for instance, due to battery depletion. When in the presence of such failures, we assume the classical crash-stop model.

CRDT objects are replicated among multiple nodes, being the composition of this node set variable over time. Objects are also globally unequivocally identifiable, and the identifier of a particular object *obj*, that we refer to as id_{obj}, must be known beforehand by all the nodes that wish to disseminate and/or receive updates to that object's state. Lastly, no imposition is made on the mechanism for the retrieval of object identifier information, leaving it to concrete implementations (we shall provide one in Section 4).

Concerning the P/S system in use, it must support topic-based subscriptions and also the persistence of both publications and subscriptions. Moreover, the P/S broker must be directly or indirectly accessible to every node, be it via a network infrastructure and/or device-to-device communication. No constraints are imposed on the broker's implementation, which may be centralized or distributed, and be internal (supported by the nodes themselves) or an external service.

Information that updates have been issued on a object *obj* is disseminated through *publish* operations on the P/S topic bound to id_{obj}, and received via notifications triggered whenever there is a match between publications and subscriptions on that same topic. Over time, nodes may overlap the roles of publisher and subscriber, publishing and subscribing to updates. To guarantee the uniqueness of the PS-CRDT topics, the function that maps object identifiers into P/S topics must naturally be injective. The model defines, thus, three main operations:

publish *(updt, topic_of_*id_{obj}*)* – disseminates the existence (the metadata) of a new update *updt* on object with identifier id_{obj}.

subscribe *(topic_of_*id_{obj}*)* – subscribes to updates addressed to object with identifier id_{obj}.

handleUpdate *(updt, topic_of_*id_{obj}*, handler)* – defines the *handler* to execute on the reception of an update to object with identifier id_{obj}.

Fig. 2 presents a scenario where six nodes subscribe to the updates on a given object (Fig. 2(a)), being that, short after, two of these nodes disconnect momentarily from the network (Fig. 2(b)). As is illustrated in Fig. 2(c), the broker is responsible for storing the metadata of published updates ($updt_m$), which includes the list of nodes (sources) that have the update, and relaying these to the subscribers currently reachable. Upon the metadata's reception, each notified node may download the update itself (whose contents will depend on the CRDT type and state) from one of the aforementioned sources (Fig. 2(d)). Once downloaded, the update is passed as argument to an invocation of operation *handleUpdate* so that it may be applied over the CRDT's state.

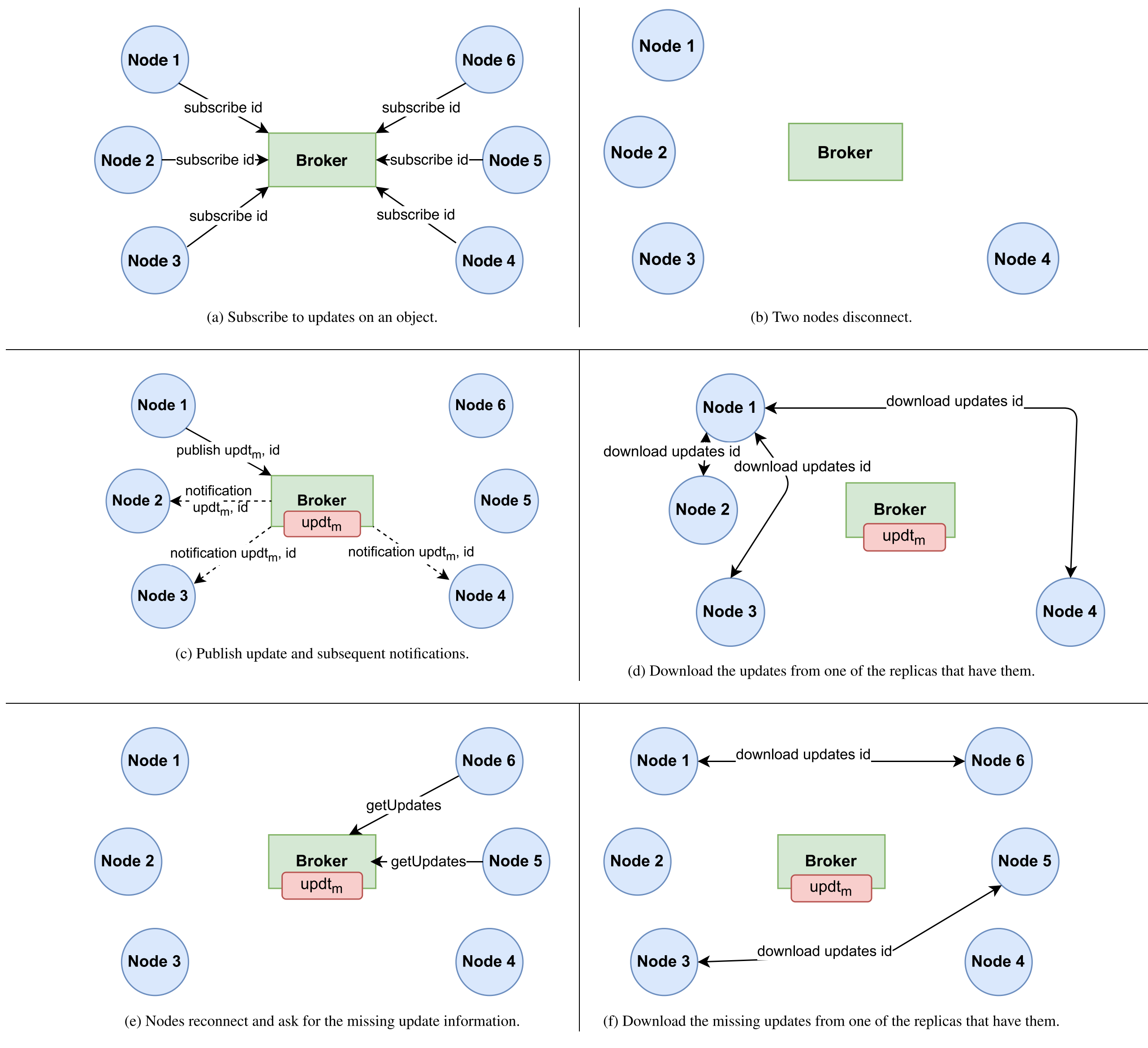
The persistence of the publications (the updates' metadata) provides the means for the nodes that lost updates to, proactively, recover this information and ensure the convergence of the replicas they hold. Fig. 2(e) illustrates the use of this mechanism by nodes 5 and 6, that have reconnected to the network. They may now obtain the list of missing updates from the broker and use it to reconcile the local state of their objects with the remainder replicas by downloading all missing updates (Fig. 2(f)).

The solutions needed to support the persistence of publications, the delivery of updates and the recovery of lost updates must be tailored to the synchronization model in use. The rest of this section discusses solutions for the state-, operation- and delta-based models.

3.1. State-based synchronization

Support for state-based synchronization in the PS-CRDTs model is fairly straightforward, but not very efficient in low throughput unreliable environments, such as mobile networks. Update dissemination must carry the entire state of the object, along with version information, so that states may be totally ordered. A simple solution to handle this data on the broker side is to queue all updates' metadata and, for each, broadcast a notification to the topic's subscribers. The logic for coping with versioned states is then delegated on the receiver nodes.

Since there is no guarantee that the order of the broker's queue is equivalent to the one induced by the state version information, a metadata object may not be deleted as soon as the respective notification is broadcast. Naturally, this limitation can be easily overcome by placing the version state information in the metadata (rather than on the update itself) and defining



(a) Subscribe to updates on an object.

(b) Two nodes disconnect.

(c) Publish update and subsequent notifications.

(d) Download the updates from one of the replicas that have them.

(e) Nodes reconnect and ask for the missing update information.

(f) Download the missing updates from one of the replicas that have them.

Fig. 2. Publishing and subscribing updates in the PS-CRDT model. $updt_m$ denotes the metadata of the update and *id* denotes the object's id.

this version state information as the queue's ordering criterion. In fact, with this criterion, the queue is no longer required: only the most up-to-date state (the one with the latest version) needs to be stored. Many P/S services allow for the definition of a ordering criterion, *e.g.*, Google Cloud PubSub.[3] The elimination of the queue may require a tailor-made implementation. Also, note that the receiver's update handler must still cope with versioned states because there is no order guarantee in the delivery of notifications.

Even with this broker optimization, state-based synchronization still places a heavy communication burden for several CRDT types, as the entire state is sent on each update download. Moreover, in dynamic systems, where the number of nodes holding replicas likely vary over time, storing the version information requires dynamic structures that will have to feature one entry per node that ever contributed to the object's current state. Given that this version information has to be sent in every state update publication, state-based synchronization is not a good solution for volatile systems nor for environments with non-reliable communication.

To tackle this issue, we devised the *Local-state-based* alternative synchronization model, that we refer to as **Lstate-based synchronization**. This synchronization model builds only on the publication of the contributions made by each individual node. Accordingly, the payload of a Lstate CRDT is reduced to the portion of the object's state that results from the modifications made at the source. The modifications received from other nodes are now stored in a second data structure that we refer to as *global state*. As in the state-based design, the *merge* operation receives two arguments. However, these are now the replica's global state and a state (that includes the payload) received from another node. The result is a new global state.

Algorithms 1 and 2 provide, respectively, simple state-based and Lstate-based implementations of a counter CRDT proposed in [3,4]. In Algorithm 1, to support an unknown number of replicas, the type of the *payload* was modified from the original array of integers to a map from replica identifiers to integer values.

3 https://cloud.google.com/pubsub/docs/ordering

Algorithm 1 State-based Counter.

```
1: payload map⟨id, int⟩ valP            ▷ Map of replica ids to their
   contributions

2: query value () : int
3:   return ∑_{r∈dom(valP)} valP(r)
4: update inc (int n)
5:   r ← repId()                       ▷ repId: get the local replica id
6:   valP(r) ← valP(r) + n             ▷ valP(r) = 0, if r ∉ dom(valP)
7: merge (X, Y) : payload Z
8:   for r ∈ X.valP.keys ∪ Y.valP.keys
9:     Z.valP(r) ← max(X.valP(r),Y.valP(r))
```

Algorithm 2 Lstate-based Counter.

```
1: globalState map⟨id, int⟩ valP        ▷ Map of replica ids to their
   contributions
2: payload int count                        ▷ Initial value: 0

3: query value () : int
4:   return ∑_{r∈dom(valP)} valP(r) + count
5: update inc (int n)
6:   count ← count + n
7: merge (X, Y) : globalState Z
8:   r ← Y.repId()                      ▷ repId: get the replica id
9:   Z.valP ← X.valP \ {r}
10:  Z.valP(r) = max(X.valP(r),Y.count)          ▷ valP(r) = 0,
                                          ▷ if r ∉ dom(valP)
```

Hence, the state of the counter, which is also the payload to be sent in every update dissemination, is given by the *valP* map that stores the sum of all contributions per replica. The value of the counter is then obtained by summing the elements of *valP* (lines 2 and 3). Incrementing the counter, adds the increment to the entry of the local replica in *valP* (lines 4 to 6). Lastly, to merge the local state with one received from a remote replica entails merging the received *payload* (a *valP* map) with the local map (lines 7 to 9). The result is a new map that projects every key r found in either on the input maps (*X.valP* and *Y.valP*) to *max(X.valP(r), Y.valP(r))*, assuming $valP(r) = 0$, if $r \notin \mathrm{dom}(valP)$.

In the Lstate implementation (Algorithm 2), the counter's state is decomposed into the *payload* (to be sent in every update dissemination) and the *globalState*. The payload becomes a simple integer (*count*) that stores the increments performed by the local replica, being the *valP* map now declared as *globalState* and storing only the sum of the increments received by each of the remainder replicas. To obtain the counter's value is to add the sum all elements of *valP* to *count*. To increment the counter is simply to add the increment to *count*. The merging conciliates a received payload with the replicas global state, producing a new global state. In the particular case of the counter, it only impacts on the entry regarding the source replica r.

3.1.1. Convergence

In Lstate, the eventual convergence of a replica is only guaranteed if its state contains the final[4] update from every other contributor. This requirement demands an *at-least-once* delivery guarantee of all notifications of these final updates. Otherwise, if a node does not receive one of such notifications, it will forever miss a contribution needed to compute the object's latest state. The remainder publications do not require delivery guarantees, as they will ultimately be subsumed by the latest update metadata publication from the same node. The order of the notifications of update metadata from the same source must however be guaranteed, so that the receiving nodes may always apply the update bound from the latest publication. In sum, updates, other than a node's final one, may be loss but not processed out of order.

A node that lost notifications, either by entering the system later than others or by being absent for a period of time, must be able to retrieve the publications that have at-least-once delivery guarantees. Only the updates received by the broker are considered. A node that is not able to inform the broker of its updates will forever diverge from the remainder replicas, as its contributions will be unknown to the system. The situation must be reported on every failed publication attempt, so that measures can be taken.

3.1.2. Complementary issues

Following our discussion on convergence, successful publish operations must be acknowledged by the broker, while unsuccessful ones must be detected and reported at client-side. With this information at hand, a node will only publish an update's metadata after it has acknowledged the success of the previous. On the broker's side, the efficient support of Lstate-based synchronization requires a custom-made storage. An option is to use a *multi-value register* with an entry per node that ever published an update. This entry will store only the node's most recent publication. No version information is required since nodes publish their updates' metadata in order, and only after the previous have been acknowledged.

3.2. Operation-based synchronization

Operation-based synchronization can also be implemented in the PS-CRDT model. An update will now carry the metadata of the operation(s) to be applied at every replica, while the update's metadata will only contain the publisher's identifier, the identifier of the last update received from a peer (to establish a chain of causality), the list of nodes containing the update (initially only the publisher) and other implementation specific information. The causal ordering requirements of op-based synchronization can be easily guaranteed in the presence of a centralizing entity (the broker). The latter simply has to sort the updates' metadata by their order of arrival. The assurance of the causality chain is delegated on the replicas.

Note that this sorting does not implement a total ordering on the updates. The application of a local update does not require coordination with the broker, and hence there are strong consistency guarantees. Consider an example with replicas A, B and C of some object, all having received no updates yet, represented by *last update received = 0*. If all of the replicas apply an update concurrently, they will publish the following metadata: $(node, 0, \{node\})$, with *node* equal to A, B or C, depending on the source. Consider, now, that the broker ordered the updates as follows:

$$[1 \mapsto (A, 0, \{A\}), 2 \mapsto (B, 0, \{B\}), 3 \mapsto (C, 0, \{C\})]$$

that we abbreviate to $[1A0, 2B0, 3C0]$. Replica A will likely apply the operations in that same order,[5] but replica B will apply them in order $[B0, 1A0, 3C0]$, because $B0$ was applied at source first and published later (local updates do not have sequence number, because the replicas have no knowledge of the number assigned to their updates). Furthermore, suppose that replica C receives

[4] The last update that will ever be published by a node on the object.

[5] As long as causality is preserved, notifications can be received out of order.

Algorithm 3 Retrieving missing updates.

```
1: function getMissingUpdates(
     lastUpdateApplied,       ▷ The number of the last update applied
     lastUpdateReceived,      ▷ The number of the last update received
     broker,                  ▷ Reference to contact the P/S broker
     object,                  ▷ Local replica of the CRDT object
     updateTopic,             ▷ The update topic for the CRDT object
     replicas,                ▷ Set of locally known replicas of the object
     time)                    ▷ The time given for the operation to conclude
2:     status ← MISSING
3:     updates ← ∅
4:     while status ≠ ALL ∧ hasNotExpired(time) do
5:         (status, updates) ← broker.getUpdates(updateTopic, lastUpdateApplied, lastUpdateReceived)
6:         if status ≠ ALL then
7:             (lastUpdateApplied, object) ← replicas.getRandom().getNewObjectVersion(updateTopic, lastUpdateApplied)
8:             if lastUpdateApplied ≥ lastUpdateReceived then
9:                 status ← ALL
10:                lastUpdateReceived ← lastUpdateApplied
11:    applyUpdates(object, updates)
12:    return status
```

update $1A0$ and applies a local update before receiving $2B0$. This local update depends on the already received update with identifier 1 and, thus, will have metadata $(C, 1, \{C\})$. Assuming no other updates, replica C will apply these in order $[C0, 1A0, C1, 2B0]$, while the remainder will only apply $4C1$ after the other 3.

Besides the broker's ordering, notifications must be delivered according to the *exactly-once* semantics, so that no notification is lost nor is processed more than once. In order to guarantee such semantics to late entering and temporary absent nodes, lost updates must be retrievable long after the notifications have been broadcast. To that end, the broker must keep a log of the updates it receives. Whenever a node asks for an update that has been evicted from the broker's storage, due to memory management policies, this node must be able to obtain a copy of a more recent version of the CRDT object from a peer, and converge to the latest state from there.

Algorithm 3 presents the function executed when a node detects that a notification is missing, due for instance to a temporary network disconnection. Variable *lastUpdateApplied* holds the identifier of the last update received and applied. It is used in line 4 to query the broker for the missing updates. The *getUpdates* operation receives an additional argument, *lastUpdateReceived*, that conveys the last update received[6]. If the broker is able to satisfy the query, *i.e.*, the metadata of all the demanded updates may be found in its storage, these are sent back to the queering node, along with a flag indicating the success of the operation (ALL). Otherwise, no updates are returned and the flag is set to MISSING. In this latter case, the node must contact a random peer, from the set of locally known replicas of the object, and obtain a more recent version of the object (line 6). The actual realization of this mechanism is an implementation detail (we will discuss an approach in Section 4), but the resulting data must always include the object and the identifier of the last update applied on it. At reception, if this update information, assigned to *lastUpdateApplied*, is still lower than the last received update, it means that some updates are still missing and the broker must be contacted once again. The function executes until a given timeout expires, and returns the status that has been reached: ALL or MISSING.

[6] The last update received may differ from the last one applied because some may be lost in-between. To force the query when these values are the same a wildcard may be used to denote the retrieval of all updates.

3.2.1. Convergence

Contrary to state-based synchronization, receiving the last update of each contributor is not enough for an op-based PS-CRDT to converge. Each replica must also receive all updates by an order that preserves causality. Given the ordering established by the metadata, when a replica receives an update notification, it has the information needed to maintain the chain of causality. Hence, it just needs to ensure that the new update is only applied (on the CRDT's state) after the ones on which it depends on have been applied.

Whenever a notification is lost, the broker must be contacted following Algorithm 3. When the metadata of the demanded updates is still stored in the broker's log, it is successfully received and the causally necessary updates are successfully downloaded, convergence is once again guaranteed. Otherwise, convergence is only guaranteed if there is a replica accessible with all the updates that have been discarded by the broker. It may also happen that, after receiving a notification, the update is nowhere to be found. None of the sources are within reach. Again, convergence is only guaranteed if there is a replica accessible with that update applied. Algorithm 3 executes once more.

3.2.2. Complementary issues

Operation-based synchronization in PS-CRDT also requires a custom storage that stores publications according to some total order. This order must be encoded on the notifications, for it to be preserved all the way to the receivers.

3.3. Lδ-based synchronization

Delta-based synchronization can also be implemented in the PS-CRDT model. We can trivially support it on top of op-based and offer a new synchronization model, on which each node publishes deltas between its own contributions to the object's state. For instance, if, since the last update, a node has added two elements to a set and removed another, the update to be propagated must only include these two new elements and the element to be removed. This delta model is, in fact, semantically equivalent to an Lstate-based approach on which only the delta between consecutive state updates are published. Thus, we baptized it as **Lδ-based synchronization**.

To implement Lδ-based on top of op-based we simply need to map each type of state modification to an operation. For instance, *add* and *remove* a set of elements for sets, *increment* and *decrement* n values for counters.

4. An implementation in *Thyme*

We implemented the three proposed PS-CRDT synchronization models atop of *Thyme* [11,12], a reactive storage system that enables co-located mobile devices to store and disseminate data among them, without having to resort to external services. The P/S system is thus decentralized and supported by the devices themselves, being inter-device communication achievable either by using a Wi-Fi infrastructure or by *ad-hoc* networking protocols. We used this implementation to develop a framework for turn-based local multiplayer games [26] and a distributed snake game.

In this section we begin by giving a brief introduction to *Thyme* and then proceed to explain how PS-CRDTs have been added to the system.

4.1. A pinch of Thyme

Thyme combines a storage interface with a topic-based P/S abstraction. It follows a data-centric storage approach that uses a key–value substrate built on top of a cluster-based distributed hash table (DHT). Clusters act as virtual P/S brokers, being responsible for matching subscriptions against published content. For that purpose, they store both subscriptions and publication metadata. The actual published data is kept on the source node, and on replicas that arise over time, either by explicitly downloading the content or via proactive replication mechanisms (when configured). Communication to a cluster is directed to a randomly chosen member that becomes responsible for performing the action on the cluster's behalf. This may include cluster-wide communication, *e.g.*, when receiving a subscription or a publication, to update the cluster's global state. A concise description of *Thyme*'s main operations follows:

publish *(item, tags, description)* – publish a data item, with associated metadata containing, among others, a small *description* (*e.g.*, a photo thumbnail) and a *set of tags* related with the content. The metadata is indexed by all the tags, *i.e.*, stored in all the clusters resultant from hashing each tag replicate the metadata.

Upon reception, a publish operation is matched against the existing subscriptions. If matches are found, the interested subscribers are notified with a message containing a set of locations from where the data may be downloaded from.

subscribe *(tags, filters)* – a subscription comprises the *query* defining which tags are relevant, and a set of *filters* to be applied over the set of publications that match against the query. Hashing each of the query's literals determines the clusters where to send that part of the query. Fig. 3 illustrates a publish and a subscribe operation.

Subscriptions in *Thyme* may target data that has been previously published. Time-aware filters may be used to defined the time period to consider. Accordingly, upon reception, a subscription operation is also matched against the publications and may trigger the emission of notifications.

download *(metadata)* – When a user receives a notification, by inspecting the item's description (such as a photo thumbnail), he/she may decide to download the item or not. If the decision is to download, *Thyme* uses the list of received locations to find a source node from where to download the data.

All operations in *Thyme* are asynchronous and so require the definition of an handler to be executed when the operation concludes, either success or unsuccessfully.

Thyme supports multiple tag namespaces across applications or even within a single application, *e.g.*, to support multiple shared to-do lists, and assumes that nodes' clocks are synchronized (with a negligible skew). Due to the unreliable nature of wireless communication mediums, the system notifies subscribers of all relevant published data as completely and faithfully as possible, *i.e.*, missing some notifications is permitted because applications are not expected to be mission-critical.

4.2. PS-CRDTs in Thyme

For application developers, *Thyme* CRDTs expose the usual interface for the development of state- and op-based solutions: function *merge* for the former, and *prepare* and *effect* for the latter [4]. Updates may be locally buffered and made public in batches to optimize node ↔ broker interaction in environments where communication is expensive. To that end, the API is complemented with function *save* that allows programmers to control when a (batch of) local modifications is ready to be published.

A requirement for sharing updates under the PS-CRDT model is to have the CRDT object's global identifier accessible to all nodes, *i.e.*, establish publisher–subscriber relationships between the nodes that wish to publish and/or receive updates. We leverage *Thyme* to share this identifier. More precisely, we use *Thyme* to share the actual CRDT objects via publications and subscriptions on tags (as the *cake* object is shared in Fig. 3). After downloading an object, a node becomes the host of a replica of that same object. *Thyme*'s globally unique object identifier is then internally used as the object's update topic. Fig. 4 illustrates the publishing, reception and download of updates.

The API does not expose the update topic. Thus, in order to subscribe to updates on an object, the programmer must use method ***subscribeUpdate****(objectId, handler)*, where *objectId* is the identifier of the PS-CRDT object and *handler* is the behavior to execute when a new update is received and applied on the local replica, *i.e.*, when operation *handleUpdate* (described in Section 3) internally concludes its execution. Update publication (operation *publish* in Section 3) is automatically triggered by the system, by periodically checking if a new batch of local modifications is ready to be published.

4.2.1. Handling lost updates

Convergence in Lstate-based synchronization requires at-least-once delivery guarantee for the notification of the last update ever published by a node. The requirements are stricter in the op-based counterpart, with notifications having to be delivered exactly-once and in causal order. Given that, in *Thyme*, there is no generic way of knowing if a given publication will be the last ever made by a node, we implemented a mechanism to enable nodes to detect if any notification was lost.

To support such mechanism and, at the same time, provide for the causal order delivery, we use a sequencing scheme similar to the one found in blockchain [27,28]. Here, each update notification message contains an hash of the previous one. On reception, a node compares the hash included in the notification message against the last one received (concerning a given sequence). If the hashes do not match, they become the parameters of the *getUpdates* request to the broker in line 4 of Algorithm 3. To cope with late entries, disconnections, or simply the loss of the last notifications for a considerable amount of time, the interval between two *getUpdates* requests may not exceed a configurable time limit. When this limit expires, a new request is sent to the broker, asking for all updates from the last one received.

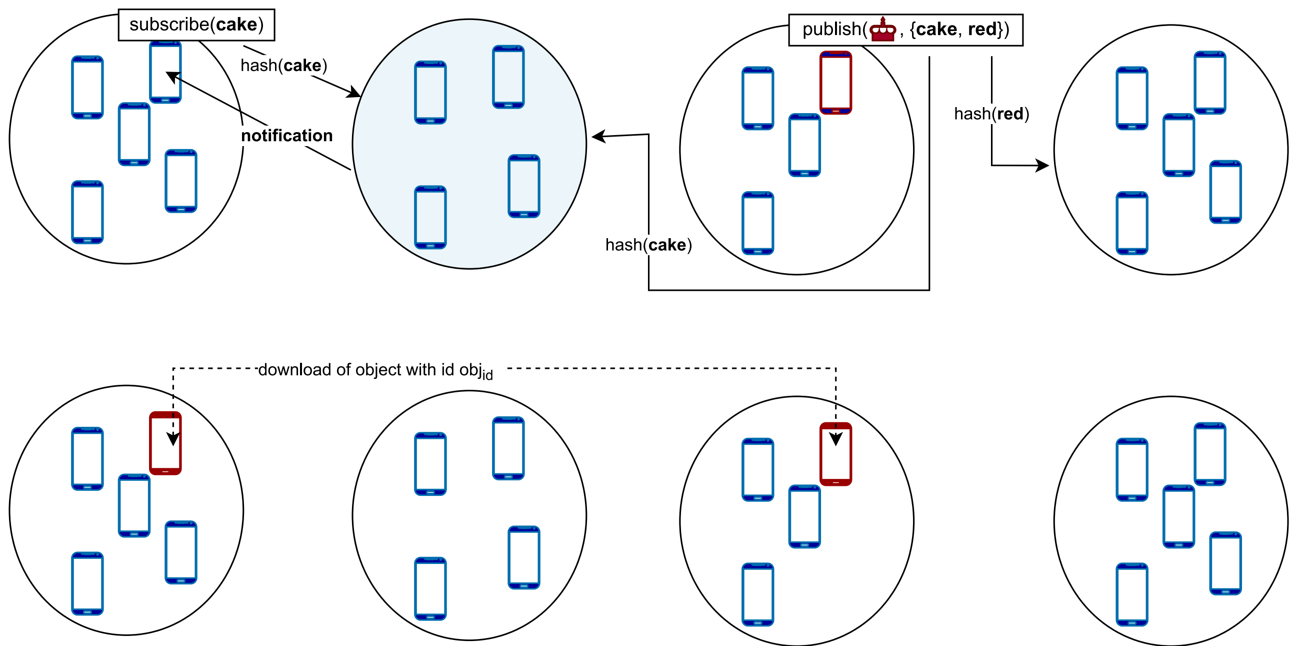


Fig. 3. Sharing a data item in *Thyme*. The network features 18 nodes grouped into 4 clusters. The hashing of the tags used in both publications and subscriptions determines the clusters responsible for managing both the metadata of the published objects and the subscriptions' data. Once a publication and a subscription match, a notification is sent to the subscribing node (figure above) and a download operation may be issued (figure below). The red nodes illustrate the replicas of the published *cake* object.

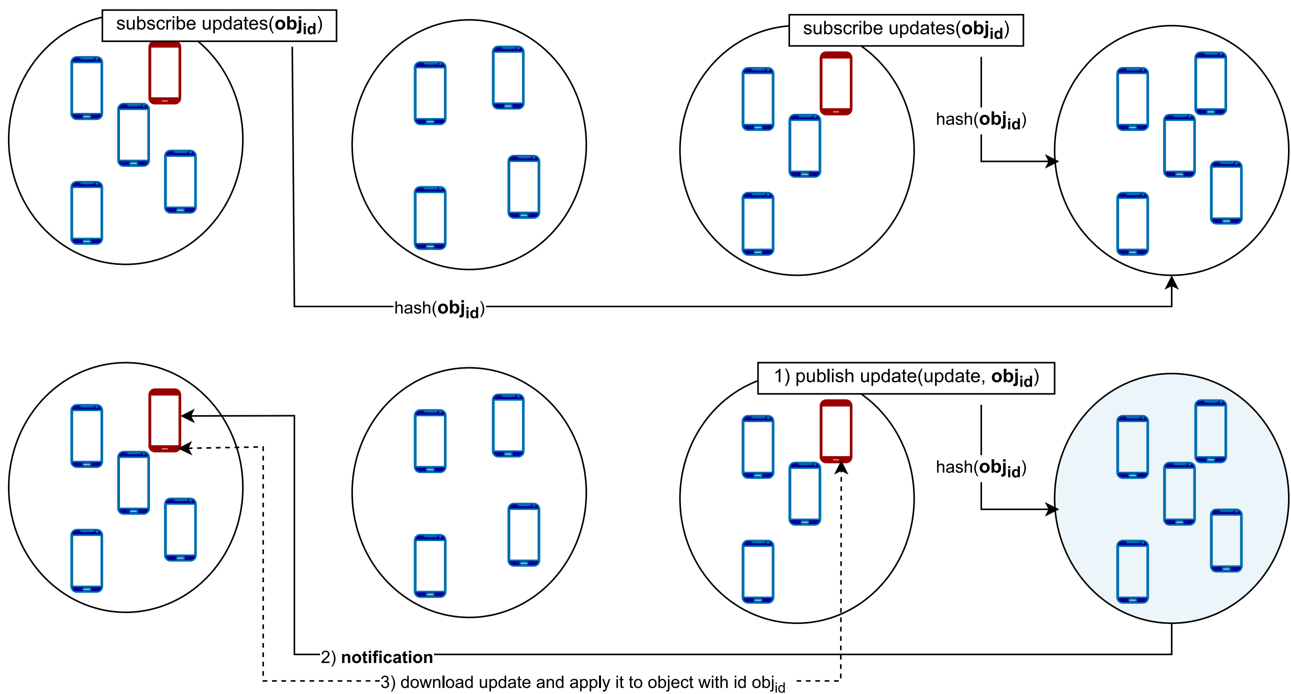


Fig. 4. Publishing & downloading updates with *Thyme*. All nodes may subscribe updates to the PS-CRDTs they hold. When one of them issues an update, the other(s) are notified, download the update and apply it to their local replica.

In Lstate, concurrent updates do not need to be ordered. Ergo, the sequencing is applied to each pair ⟨source node, tag⟩. In op-based, concurrent update ordering is also not needed, but publications must be ordered. This raises a challenge to the cluster-based structure of *Thyme*, as each cluster acts as a virtual broker for a set of topics/tags. Communication with a cluster is directed randomly to one of the nodes currently composing it. Accordingly, two concurrent publications to the same tag can be directed to two different nodes in a cluster. This renders cluster-wide sequencing impossible without cluster-wide coordination. Our current solution elects a cluster-leader per CRDT object, responsible for sequencing and emitting the object's notifications. To meet the exactly-once requirements, if a notification message is received more than once, it is simply discarded. Hash information is also included in the reply to a *getNewObjectVersion* request. Besides the object's state, such requests include the hash of the last update applied, so that Algorithm 3 can be iteratively executed.

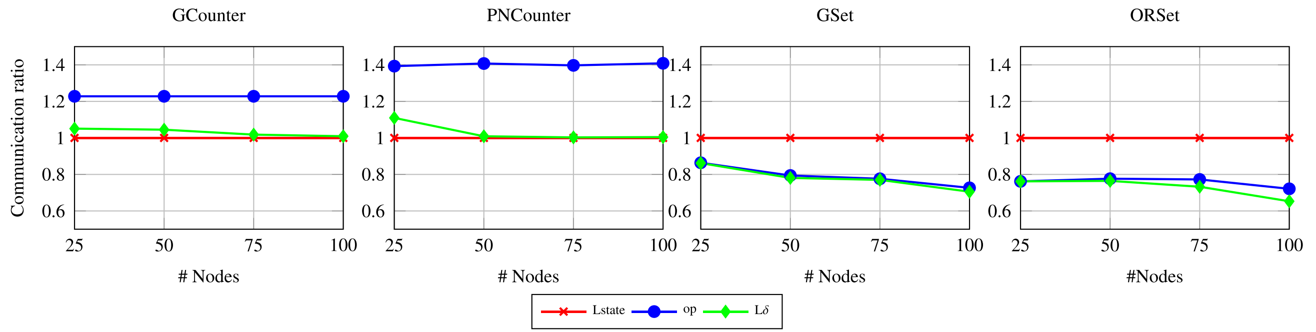


Fig. 5. Communication wrt. Lstate for known implementations of counters and sets.

4.2.2. The broker

The broker configuration depends on the synchronization model in place. That is, a multi-value register for Lstate-based and list for op-based. This configuration information is encoded in the id_{obj}'s tag by prefixing it to the object identifier, such as: *sync_model* :: id_{obj}. Upon reception of the first message (publication or subscription) on a tag with such structure, the broker extracts the *sync_model* component, and uses it to configure the type of storage needed to manage the publications on the referred tag.

5. Evaluation

In this section we present a set of experiments that evaluate implementation, with the aim of answering the following questions:

1. How does it scale with the increase of the number of nodes?
2. How sensitive are PS-CRDTs to churn? and
3. How do PS-CRDTs compare against Δ-based CRDTs and CRDT-based centralized solutions in volatile environments?

5.1. Experimental setup

In order to assess the behavior of our approach without hardware limitations, we resorted to the emulation of the mobile nodes. To that end, we made use of the trace-based emulation framework implemented for *Thyme*. This framework allows for *Thyme*'s non-graphical Android code to execute unaltered. User interaction is encoded in the input trace, with commands *Publish*, *Subscribe* and *Save*, among others. The inter-node communication layer was reworked to support the logical dissemination of messages between the emulated nodes. Moreover, the simulation allows for the definition of the network's speed and message queuing delay, and, to simulate churn, the trace may include commands for the ingress and egress of nodes.

Each node begins its execution by subscribing to a set of CRDT objects, which are subsequently published by a set of selected nodes that also automatically subscribe to the objects' updates (with the *subscribeUpdate* operation). The remainder, once notified of the objects' publications, download the object and subscribe to the updates as well.

5.2. Scalability

To analyze the scalability of our implementation of the proposed PS-CRDT synchronization models we resort to two metrics: the communication load imposed on the network and the memory consumption on the broker.

5.2.1. Network communication

Regarding the communication load, we conducted experiments with 25, 50, 75 and 100 nodes, in a scenario without churn and where all nodes join at the beginning of the simulation. Each node applies the same number of updates to a single shared PS-CRDT object. Fig. 5 depicts the network traffic per node for Grows-only (G) and Positive–Negative (PN) counters and, Grows-only (G) and Observed-Remove (OR) sets. Almost all depicted solutions scale linearly with the increase of the number of nodes and update operations. The exception are Lstate sets which tend to grow superlinearly when reaching the 100 nodes.

Concerning bandwidth, the Lstate and Lδ versions are more efficient. This is due to the fact that the state published in an update only carries the number of increments (and decrements) performed, i.e., it is small when compared with the propagation of lists of operations. With 100 nodes, this reflects in a 19% and 29% reductions for the GCounter and the PNCounter, respectively. The tables turn when analyzing sets, with the Lstate version generating more traffic than the op and Lδ counterparts. The state to propagate grows with the size of the set. Ergo, as more nodes participate, more elements are added to the set, and the communication load increases. Lδ perform even better than op-based, since they also only send the elements that have been added or removed from the set, since the last broadcast delta, with the added optimization of not sending operations that cancel each other out, such as the addition and subsequent removal of an element.

In conclusion, PS-CRDTs follow the norm that state-based CRDT solutions perform better for data types with small states, while op-based solutions perform better for the remainder. The use of the Lstate model tips the balance a little more to the side of state-based approaches by requiring fewer bytes to encode the object's state. This is even more evident in the Lδ approach that reduces the state (to propagate) to the one that is still unknown to the system.

5.2.2. Memory requirements

The broker only stores the metadata of the updates and the subscriptions made by the nodes. Therefore, for op-based PS-CRDTs the broker's memory consumption is bounded by the size of the operation log. For a log of o operations and a maximum of n nodes subscribing to updates on the object, the upper bound can be written as

$$mem_{op}(o, n) = l_0 + o \times sizeof_{pub}(n) + s_0 + n \times sizeof_{sub}$$

where l_0 denotes the amount of memory needed to create the empty log, $sizeof_{pub}(n)$ denotes the memory needed to store a publication replicated by (at most) n nodes, s_0 denotes the memory needed to create the empty subscription storage and $sizeof_{sub}$ denotes the memory needed to store a subscription.

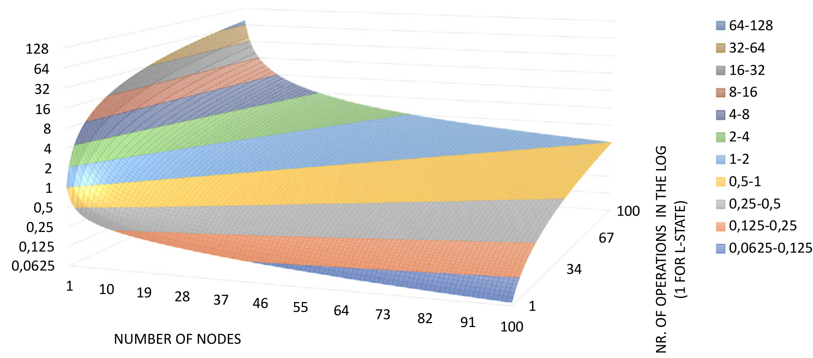


Fig. 6. Memory required by the broker. Ratio between op and Lstate (log2n scale).

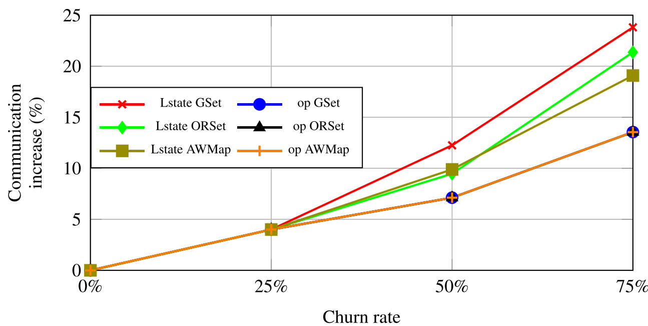


Fig. 7. Communication increase in the presence of churn (100 nodes).

Table 1
Update download: op GSet.

Churn	# Ops	Updates	getUpdates	getNewObjVersion
0%	689	689 (100%)	0 (0%)	0 (0%)
25%	742	665 (90%)	48 (6%)	30 (4%)
50%	823	646 (78%)	93 (11%)	84 (10%)
75%	916	644 (70%)	157 (17%)	115 (13%)

Table 2
Update download: Lstate GSet.

Churn	# Ops	Updates	getUpdates	getNewObjVersion
0%	685	685 (100%)	0 (0%)	0 (0%)
25%	740	677 (91%)	63 (9%)	0 (0%)
50%	817	711 (87%)	106 (13%)	0 (0%)
75%	909	730 (80%)	179 (20%)	0 (0%)

Concerning Lstate, the broker needs to store the last state published by every known replica. The required storage capacity needed is thus given by:

$$mem_{Lstate}(n) = p_0 + n \times sizeof_{pub}(n) + s_0 + n \times sizeof_{sub}$$

where p_0 denotes the space needed to allocate an empty multi-value register. In *Thyme*, when it comes to updates, we have that $sizeof_{pub}(n) \approx 96 + 4 \times n$ bytes and that $sizeof_{sub} \approx 32$ bytes. So, the memory required is low, peaking at ≈ 53 Kbytes for 100 nodes for both models. However, to provide a better understanding of their relative memory requirements, Fig. 6 depicts the ratio $\frac{mem_{op}(o,n)}{mem_{Lstate}(n)}$ varying both o and $n \in [1, 100]$.

For a small number of nodes, we assume that the log of operations in op-based PS-CRDTs (and Lδ consequently) is likely to surpass the number of nodes. In such scenario, Lstate solutions are more memory efficient. In scenarios with higher number of nodes, it all comes down to the update rate. A high update rate benefits from a log with more capacity, which means a trade-off between the memory needed by the log and the network traffic required to transfer the whole object when the metadata on a requested update is no longer available. Low update rates will likely keep the log small and thus be more efficient that the Lstate alternative.

5.3. Sensitivity to churn

To study the impact of node churn, we set up a scenario with 100 nodes and varied the percentage of nodes that temporary leave the network. The disconnection period is randomly defined and causes a node to loose from 5 to 20 updates. To recover this lost information, and allow for the convergence of the replicas, Lstate-based implementations only need to query the broker for lost updates (*getUpdates*), whist op- and Lδ-based versions must execute Algorithm 3, which includes prompting other nodes for a more recent version of the object's state (*getNewObjectVersion*).

Tables 1 and 2 break down the type and number of operations required to keep the shared CRDT object's state up to date for op- and Lstate-based GSets, respectively. In the churn-free scenarios, all updates were delivered and, thus, no *getUpdate* nor *getNewObjectVersion* operations were needed. Naturally, the number of lost updates increases with the churn rate. In the op-based version, the broker's storage was configured to store up to 10 operations. Thus, convergence from a state *older* that the last 10 operations requires the download of a new version of the object. This does not happen in the Lstate-based GSet because the broker's storage always has the last update published by each node.

Fig. 7 depicts the gradual increase in network traffic, for several CRDTs (including the Add-Wins (AW) Map), when in the presence of churn. A churn rate of 25% implies ≈4% increase in traffic for all studied CRDTs. From then onward, CRDTs with fast-growing states are more sensitive to churn, since the size of the updates, and specially, of the full versions to transfer is bigger. However, note that, even for a 75% churn rate, the traffic increase was only from ≈13% to ≈23%.

5.4. Comparison against Δ-CRDTs

Prior to our work, the Δ-CRDT model [10] was the best suited one for volatile environments. We hence choose it as a baseline for the evaluation of PS-CRDT. To be able to fairly compare both models, we had to implement Δ-CRDTs atop of *Thyme*. Δ-CRDTs compute the minimal delta that needs to be propagated to another node since their previous communication round. This delta is computed from locally stored information about the causal context between those nodes (commonly implemented with vector clocks). We replicate this behavior by sending

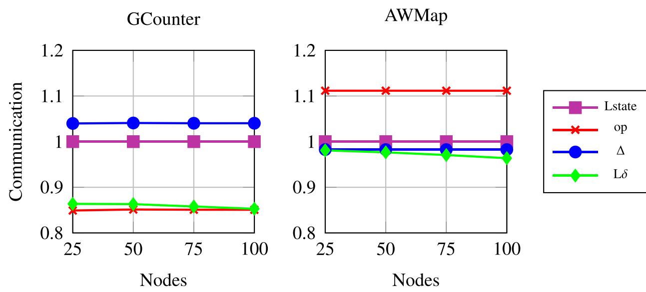


Fig. 8. Communication ratio of PS-CRDTs wrt. Δ-CRDTs, varying the number of nodes.

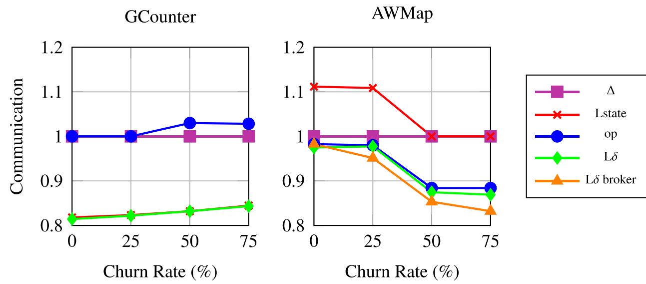


Fig. 9. Communication ratio of PS-CRDTs wrt. Δ-CRDTs, varying the churn rate.

a *getDelta* request containing the version vector of the requesting node. On reception, the target node computes the minimal delta by comparing the received vector with its own, checking for missing updates. The delta is subsequently shipped back to the requester. In order to enable direct node-to-node communication (N2N), particularly the sending of messages to random nodes required by Δ-CRDTs, we use a special *Thyme* tag on which all nodes publish their network address.

5.4.1. Static scenarios

To compare the PS-CRDT and Δ-CRDT models in both ends of the *state size growth in time* spectrum, we developed two Δ-CRDTs: GCounter and Add-Wins Map (AWMap). Fig. 8 depicts the network traffic for both the PS-CRDT and the Δ-CRDT models. For GCounter, Lstate generates ≈20% less network traffic than Δ. This is justified by the fact that, in Δ, delta messages carry all the missing modifications, while the payload of update messages in Lstate is a single integer. This behavior can be extrapolated to mostly all Lstate CRDTs with small payloads. The difference dims as the payload increases, due to the dilution of the delta messages' total in the overall amount of data to communicate. We may observe this fact in the chart for AWMap, where Δ outperforms Lstate. Moreover, we have the op-based solution performing marginally better than Δ, due to the similar size (in this case) of the delta and operation payloads. The overall better solution is Lδ that is able to perform close to Lstate, in scenarios on which state-based solutions are better, and close (even better) than op-based, in scenarios on which op-based solutions are more efficient.

These results show that, even in static scenarios, PS-CRDTs are competitive against the Δ approach. Lstate and Lδ reveal to be better than Δ for state-based CRDTs with small payloads, while op-based and Lδ show to be on par with Δ for state-based CRDTs with big payloads.

5.4.2. Sensitivity to churn

Fig. 9 presents the comparison results in the presence of churn, with the same conditions of Section 5.3. For GCounter, the Δ alternative scales worse than Lstate because of the overhead brought by failed communication attempts. N2N is more sensitive to churn, since the local knowledge of the network may be outdated. The overhead of Δ peaks at 27%. Once again, this is less noticeable in the AWMap case, where the overhead of failed communication is also diluted in the overall communication. Yet, the alternatives with better results (op and Lδ) also make use of N2N to obtain the newer versions of the objects. If we remove this communication, *i.e.*, if we assume that the broker's log is large enough to hold all the updates needed by the nodes reentering the network (line *Lδ broker* in the chart), than the Lδ-based approach behaves even better than the Δ-based. Naturally, late arriving nodes will have always to execute Algorithm 3 to obtain a replica that is able to converge with the remainder.

The benefits of PS-CRDTs increase when in the presence of churn, once again with Lδ leading the gains. The best results appear when the broker has a log large enough to accommodate all updates requested by the replicas. Once again we fall in the memory-network consumption trade-off of Section 5.2.2.

5.5. Comparison against centralized CRDT solutions

Lastly, we analyze the overhead of having a CRDT-based centralized solution versus PS-CRDTs. The centralized solution requires the server to synchronize with each replica individually, much like what is done in solutions that delegate all synchronization on the edge. Accordingly, the server(s) receive all updates and relay them to the remainder replicas (with possible update fusion). Conversely, in *Thyme*-PS-CRDT, the broker node(s) only receive publications carrying the updates' metadata and trigger notifications comprising both the metadata and the location(s) of the data.

Fig. 10 presents the communication performed by the servers (the possible bottlenecks of both approaches) to handle an update, when varying the size of the update (32 bytes to 10 K) and the number of replicas (1 to 100). The values presented are the ratio between the centralized and the Lδ solutions. We may observe that, as soon as the size of the updates surpasses the size of a publication, and subsequent notifications, the overhead of the centralized solution is very high (peeking at ≈318). The overhead is more prominent when the number of replicas is small; what we expect to be the more common scenario. This happens because the size of a notification is directly proportional to the number of locations from where the data may be downloaded from. In this experiment, the number of update locations was (in average) half the number of replicas.

6. Conclusions

CRDTs are an increasingly popular approach to avoid expensive synchronization in distributed replicated systems. Their applicability is however limited when targeting volatile systems. In this paper, we proposed PS-CRDTs, a new approach to the dissemination of updates between replicas that enable the source to propagate its modifications to a CRDT's state without knowing who is at the other end. Having bandwidth limitations in mind, we developed the Lstate and the Lδ-based solutions. The latter has proved to be one that best performs across the board. However, when in the context of data structures that do not grow in size with the number of replicas or operations, Lstate provides an equally efficient solution.

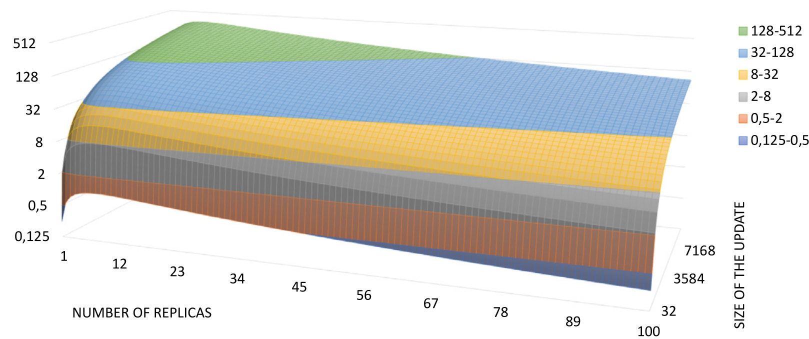


Fig. 10. Communication overhead on the server(s) of a CRDT-based centralized solution vs. Lδ PS-CRDTs (log2n scale).

CRediT authorship contribution statement

António Barreto: Conceptualization, Software, Validation, Investigation, Data curation, Writing – original draft, Visualization. **Hervé Paulino:** Conceptualization, Software, Investigation, Data curation, Writing – original draft, Writing – review & editing, Supervision, Funding acquisition. **João A. Silva:** Conceptualization, Software, Data curation, Writing – review & editing, Visualization. **Nuno Preguiça:** Conceptualization, Writing – review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

This work was partially supported by *Fundação para a Ciência e a Tecnologia, Portugal*, through project *DeDuCe* (PTDC/CCI-COM/32166/2017) and the *NOVA LINCS, Portugal* research center (UIDB/04516/2020).

References

[1] D.B. Terry, Replicated Data Management for Mobile Computing, Morgan & Claypool Publishers, 2008.
[2] E.A. Brewer, A certain freedom: thoughts on the CAP theorem, in: A.W. Richa, R. Guerraoui (Eds.), Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010, ACM, 2010, p. 335, http://dx.doi.org/10.1145/1835698.1835701.
[3] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, A comprehensive study of Convergent and Commutative Replicated Data Types, Tech. Rep. RR-7506, INRIA, 2011, p. 50, URL https://hal.inria.fr/inria-00555588.
[4] N.M. Preguiça, Conflict-free replicated data types: An overview, 2018, CoRR arXiv:1806.10254, URL http://arxiv.org/abs/1806.10254.
[5] B. Nédelec, P. Molli, A. Mostéfaoui, E. Desmontils, LSEQ: an adaptive structure for sequences in distributed collaborative editing, in: S. Marinai, K. Marriott (Eds.), ACM Symposium on Document Engineering 2013, DocEng '13, Florence, Italy, September 10-13, 2013, ACM, 2013, pp. 37–46, http://dx.doi.org/10.1145/2494266.2494278.
[6] N.M. Preguiça, J.M. Marquès, M. Shapiro, M. Letia, A commutative replicated data type for cooperative editing, in: 29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada, IEEE Computer Society, 2009, pp. 395–403, http://dx.doi.org/10.1109/ICDCS.2009.20.
[7] X. Lv, F. He, Y. Cheng, Y. Wu, A novel CRDT-based synchronization method for real-time collaborative CAD systems, Adv. Eng. Inform. 38 (2018) 381–391, http://dx.doi.org/10.1016/j.aei.2018.08.008.
[8] A. van der Linde, P. Fouto, J. Leitão, N.M. Preguiça, S.J. Castiñeira, A. Bieniusa, Legion: Enriching internet services with peer-to-peer interactions, in: R. Barrett, R. Cummings, E. Agichtein, E. Gabrilovich (Eds.), Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017, ACM, 2017, pp. 283–292, http://dx.doi.org/10.1145/3038912.3052673.
[9] J.J. Kistler, Disconnected Operation in a Distributed File System, in: Lecture Notes in Computer Science, Vol. 1002, Springer, 1995, http://dx.doi.org/10.1007/3-540-60627-0.
[10] A. van der Linde, J. Leitão, N.M. Preguiça, Δ-CRDTs: making Δ-CRDTs delta-based, in: P. Alvaro, A. Bessani (Eds.), Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016, ACM, 2016, pp. 12:1–12:4, http://dx.doi.org/10.1145/2911151.2911163.
[11] J.A. Silva, F. Cerqueira, H. Paulino, J.M. Lourenço, J. Leitão, N.M. Preguiça, It's about Thyme: On the design and implementation of a time-aware reactive storage system for pervasive edge computing environments, Future Gener. Comput. Syst. 118 (2021) 14–36, http://dx.doi.org/10.1016/j.future.2020.12.008.
[12] J.A. Silva, P. Vieira, H. Paulino, Data storage and sharing for mobile devices in multi-region edge networks, in: 21st IEEE International Symposium on "a World of Wireless, Mobile and Multimedia Networks", WoWMoM 2020, Cork, Ireland, August 31 - September 3, 2020, IEEE, 2020, pp. 40–49, http://dx.doi.org/10.1109/WoWMoM49955.2020.00021.
[13] A. Auvolat, F. Taïani, Merkle search trees: Efficient state-based CRDTs in open networks, in: 38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 1-4, 2019, IEEE, 2019, pp. 221–230, http://dx.doi.org/10.1109/SRDS47363.2019.00032.
[14] C. Baquero, P.S. Almeida, A. Shoker, Making operation-based CRDTs operation-based, in: K. Magoutis, P.R. Pietzuch (Eds.), Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held As Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings, in: Lecture Notes in Computer Science, Vol. 8460, Springer, 2014, pp. 126–140, http://dx.doi.org/10.1007/978-3-662-43352-2_11.
[15] P.S. Almeida, A. Shoker, C. Baquero, Efficient state-based CRDTs by delta-mutation, in: A. Bouajjani, H. Fauconnier (Eds.), Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers, in: Lecture Notes in Computer Science, Vol. 9466, Springer, 2015, pp. 62–76, http://dx.doi.org/10.1007/978-3-319-26850-7_5.
[16] V. Enes, P.S. Almeida, C. Baquero, J. Leitão, Efficient synchronization of state-based CRDTs, in: 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019, IEEE, 2019, pp. 148–159, http://dx.doi.org/10.1109/ICDE.2019.00022.
[17] D.B. Terry, M. Theimer, K. Petersen, A.J. Demers, M. Spreitzer, C. Hauser, Managing update conflicts in bayou, a weakly connected replicated storage system, in: M.B. Jones (Ed.), Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995, ACM, 1995, pp. 172–183, http://dx.doi.org/10.1145/224056.224070.
[18] A.D. Joseph, A.F. deLespinasse, J.A. Tauber, D.K. Gifford, M.F. Kaashoek, Rover: A toolkit for mobile information access, in: M.B. Jones (Ed.), Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995, ACM, 1995, pp. 156–171, http://dx.doi.org/10.1145/224056.224069.

[19] N.M. Preguiça, M. Shapiro, C. Matheson, Semantics-based reconciliation for collaborative and mobile environments, in: R. Meersman, Z. Tari, D.C. Schmidt (Eds.), On the Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003, in: Lecture Notes in Computer Science, Vol. 2888, Springer, 2003, pp. 38–55, http://dx.doi.org/10.1007/978-3-540-39964-3_5.

[20] L. Benmouffok, J. Busca, J.M. Marquès, M. Shapiro, P. Sutra, G. Tsoukalas, Telex: Principled system support for write-sharing in collaborative applications, 2008, CoRR arXiv:0805.4680, URL http://arxiv.org/abs/0805.4680.

[21] M. Zawirski, N.M. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, M. Shapiro, Write fast, read in the past: Causal consistency for client-side applications, in: R. Lea, S. Gopalakrishnan, E. Tilevich, A.L. Murphy, M. Blackstock (Eds.), Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, December 07 - 11, 2015, ACM, 2015, pp. 75–87, http://dx.doi.org/10.1145/2814576.2814733.

[22] S.H. Mortazavi, M. Salehe, C.S. Gomes, C. Phillips, E. de Lara, Cloudpath: a multi-tier cloud computing framework, in: J. Zhang, M. Chiang, B.M. Maggs (Eds.), Proceedings of the Second ACM/IEEE Symposium on Edge Computing, San Jose / Silicon Valley, SEC 2017, CA, USA, October 12-14, 2017, 2017, pp. 20:1–20:13.

[23] S.H. Mortazavi, B. Balasubramanian, E. de Lara, S.P. Narayanan, Toward session consistency for the edge, in: I. Ahmad, S. Sundararaman (Eds.), USENIX Workshop on Hot Topics in Edge Computing, HotEdge 2018, Boston, MA, July 10, 2018, USENIX Association, 2018.

[24] C. Li, C. Wang, H. Tang, Y. Luo, Scalable and dynamic replica consistency maintenance for edge-cloud system, Future Gener. Comput. Syst. 101 (2019) 590–604.

[25] D.J. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H.V. Madhyastha, C. Ungureanu, Simba: tunable end-to-end data consistency for mobile apps, in: L. Réveillère, T. Harris, M. Herlihy (Eds.), Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015, ACM, 2015, pp. 7:1–7:16, http://dx.doi.org/10.1145/2741948.2741974.

[26] S. Marreiros, A framework for turn-based local multiplayer games, (Master's thesis), NOVA School Science & Tecnology, 2020, URL https://run.unl.pt/bitstream/10362/120491/1/Marreiros_2020.pdf.

[27] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, 2009, URL http://www.bitcoin.org/bitcoin.pdf.

[28] Z. Zheng, S. Xie, H. Dai, X. Chen, H. Wang, Blockchain challenges and opportunities: a survey, Int. J. Web Grid Serv. 14 (4) (2018) 352–375.

António Barreto received his B.Sc. and M.Sc. degrees in computer engineering from NOVA School of Science and Technology, NOVA University of Lisbon (FCT/UNL), in 2018 and 2019, respectively. His research interests include edge computing, and data storage and dissemination.

Hervé Paulino is an Associate Professor at the Computer Science Department of NOVA University of Lisbon. He is also an integrated researcher at NOVA LINCS, and a collaborator member of the Research Center for Research in Advanced Computing Systems (CRACS) from Universidade do Porto. He received his Ph.D. in computer science from NOVA University of Lisbon in 2006, in the area of mobile agent computing. Currently, his research interests center on the fields of data-centric concurrency, parallel programming, with particular interest in heterogeneous systems, and availability in largescale systems.

João A. Silva received his Ph.D. degree in computer engineering from NOVA School of Science and Technology in 2021. He is currently as a Software Engineer at Sensei, a Portuguese electronic retail technology company. His research interests include edge computing, and data storage and dissemination, with special emphasis on mobile and wireless environments.

Nuno Preguiça is an Associate Professor with Habilitation in the Department of Computer Science from NOVA School of Science and Technology, NOVA University of Lisbon (FCT/UNL), and leads the Computer Systems group at NOVA Laboratory for Computer Science and Informatics (NOVA LINCS). His research interests are focused on the problems of replicated data management and processing of large amounts of information in distributed systems and mobile computing settings.