Improving the Reactivity of Pure Operation-Based CRDTs

Jim Bauwens
Software Languages Lab
Vrije Universiteit Brussel
Belgium
jim.bauwens@vub.be

Elisa Gonzalez Boix
Software Languages Lab
Vrije Universiteit Brussel
Belgium
egonzale@vub.be

Abstract

Modern distributed applications increasingly replicate data to guarantee both high availability of the system and an optimal user experience. Conflict-Free Replicated Data Types (CRDTs) are a family of data types specially designed for highly available systems which guarantee some form of eventual consistency. In this paper, we show that the reliance on reliable causal broadcasting (RCB) middleware by existing CRDT frameworks may lead to less reactive CRDTs which in turn hampers user experience. We propose a solution that improves the reactivity of CRDTs built on an RCB middleware by reifying buffered operations. We apply our strategy to the pure operation-based CRDT framework, allowing for reactive pure operation-based CRDTs. We describe our approach as a formal extension to the framework and implement it in Flec, an extensible and open CRDT framework. The extension is then used to create new Add-Wins and Remove-Wins sets which exhibit higher reactivity.

CCS Concepts: • **Software and its engineering** → **Consistency**; **Synchronization**; **Middleware**; **Reflective middleware**; • **Computer systems organization** → **Distributed architectures**.

Keywords: Replication, CRDTs, Middleware, Reflection, WebAssembly, Eventual consistency

ACM Reference Format:
Jim Bauwens and Elisa Gonzalez Boix. 2021. Improving the Reactivity of Pure Operation-Based CRDTs. In *8th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC'21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3447865.3457968

1 Introduction

Many CRDT implementations rely on Reliable Causal Broadcasting [5] (RCB), ensuring both causal ordering and reliable delivery [2, 10] of operations. In fact, RCB is commonly understood to be a requirement for operation-based CRDTs [1, 2, 9, 10]. The main benefit is that causal ordering reduces complexity in the implementation of operation-based CRDTs, as the handling of non-concurrent operations becomes trivial.

In the case of concurrent operations, some causal meta-data will still be required to ensure the commutativity of operations, which can lead to unbounded memory growth. But even here, an RCB middleware may help: Pure operation-based CRDTs [2], a framework for constructing operation-based CRDTs makes additional use of the RCB middleware for tackling meta-data removal. The framework builds a partially ordered log (PO-Log) of operations by piggybacking on the causality information available in the middleware (following the happened before relation [8]). When operations are causally stable (i.e. operations for which no new concurrent operations can occur) or become redundant by newer operations, they are compacted or removed from the log.

While the benefits of using an RCB middleware are large, it is not always desirable to fully rely on causal ordering as it may hamper the reactivity of operation-based CRDTs. When operations arrive out of causal order (e.g. before other operations that happened before), they are buffered by the RCB middleware until all causal predecessors arrive. Since the happened before relation does not always imply an actual dependency between operations, operations may be buffered by the RCB middleware needlessly. The result of this is a less responsive CRDT, where replicas may have to wait for unrelated updates of other replicas before they can apply already received updates. This, in turn, will hamper user experience as applications may suffer from unnecessary delays. In the case of pure op-based CRDTs, waiting may additionally impact the removal of redundant log entries, leading to higher memory consumption.

Table 1. A sequence of operations applied to several replicas implementing an add-wins pure operation-based set CRDT. The last remove does not have any immediate effect on A, as A is waiting for B before it will apply any other operation from C.

Operation	SET A	SET B	SET C
	{}	{}	{}
SET C :: Add (X)			
SET C :: Add (Y)			
	{X,Y}	{X,Y}	{X,Y}
SET B :: Add(Z)			
	{X,Y}	{X,Y,Z}	{X,Y,Z}
SET C :: Remove(X)			
	{X,Y}	{Y,Z}	{Y,Z}

Table 2. A sequence of operations applied to several replicas implementing a classic OR-Set CRDT. The last remove is applied immediately on set A.

Operation	SET A	SET B	SET C
	{}	{}	{}
SET C :: Add (X)			
SET C :: Add (Y)			
	{X,Y}	{X,Y}	{X,Y}
SET B :: Add(Z)			
	{X,Y}	{X,Y,Z}	{X,Y,Z}
SET C :: Remove(X)			
	{Y}	{Y,Z}	{Y,Z}

2 Causal Ordering and its Impact on Reactivity

To demonstrate the implications of delayed operations, consider a sequence of operations that is applied to three set replicas named A, B, and C. Figure 1 visualises the connectivity between the replicas. Black lines denote bidirectional connectivity between replicas and dotted lines temporal network failures. The figure shows a scenario in which updates are not propagated between replicas A and B.

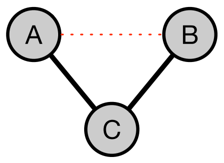

Figure 1. Network connectivity between set replicas

Assume that the replicas host two different set CRDTs, both with add-wins semantics: a pure operation-based Add-Wins set[2] and an operation-based OR-Set CRDT that utilises tombstones[4]. Contrary to pure operation-based Add-Wins CRDTs, OR-Set CRDTs do not require an RCB middleware, as tombstones and unique identifiers are used for tracking causality. Table 1 shows the operations applied to the pure op-based Add-Wins set, while table 2 shows the operations applied to the OR-Set CRDT. We apply the operations to the replicas in the following order:

1. As a first step, the elements X and Y are added to replica C. This update is propagated to all other sets and their state is updated.
2. Following this, the element Z is added to set B. This update is only sent to set C, as there is a disconnection between set A and B.
3. Finally set C applies the remove of item X, which will be observed by set A and B, as set C is connected to both other replicas.

In the case of the OR-Set, the item will be immediately removed on all replicas. This differs however for the Add-Wins set, where the operation will not be applied to set A. The reasoning behind this is simple: the Add-Wins pure op-based implementation set relies on the RCB middleware. The middleware will buffer the operation as it can derive from the causality information that it received along with the operation (from set C) that A has not yet received one or more operations from B. In practice, this means that only after the connectivity issue between A and B is resolved, and set A receives and applies the `Add(Z)` operation from B, the remove operation from set C will be applied.

While the OR-Set has a clear benefit in terms of reactivity when compared to the Add-Wins set, the implementation is very ad-hoc and has a higher complexity: unique identifiers need to be generated for every operation and tombstones have to be kept to make sure that remove operations commute. In the following section we propose a solution that brings higher reactivity to operation-based CRDTs frameworks without having to rely on ad-hoc techniques.

3 Improving the Reactivity of CRDTs in an RCB-based Approach

To improve the reactivity of operation-based CRDTs utilising RCB in a generic (non ad-hoc) way, we propose that the buffer of the RCB middleware where these messages are held is made accessible (i.e. reified) to CRDT implementors as part of the framework interface. In the context of a pure operation-based CRDT, this buffer can then be used to construct an *incomplete* partially ordered log. We say it is incomplete because the buffer can contain gaps of missing causal dependencies. The incomplete log can then complement the existing partially-ordered log (PO-Log) and compacted sequential state to represent the full CRDT state. Furthermore, entries in the main PO-Log and the compacted state can be made redundant by entries from the incomplete log. However, entries in the incomplete log cannot be made redundant as long as they have not yet been moved to the main PO-Log as concurrent operations that might be affected by the operation may yet arrive.

This approach has, aside from increasing the reactivity of CRDTs, some additional benefits in terms of memory management. Since entries from the incomplete log can cause entries from the main log to become redundant, it can be used for decreasing memory consumption whenever intermediate disconnections are common. When disconnections are common it may be hard to determine causal stability as not all replicas will be responsive. By observing the incomplete log, potentially redundant information can already be removed from the main log even if causal dependencies are missing.

In the next section, we detail our approach as an extension to the pure operation-based CRDT framework in the form of a formal specification. We illustrate the applicability of the approach by applying it to Add-Wins and Remove-Wins sets. Following this, we describe the implementation of our strategy and the extended sets in our own TypeScript CRDT framework.

3.1 Extending Semantic Log Compaction with incomplete PO-Log Support

Pure operation-based CRDTs utilise a mechanism named *causal redundancy* to prune operations from the log whenever they become causally redundant. Baquero et al. define this mechanism by means of binary redundancy relations (typically named R and $\mathrm{R\_}$). We extend this mechanism with a new binary relationship R_β that defines the relation between log entries that live in the RCB buffer (e.g. the incomplete PO-Log segment) and the non-buffered main log. The relation defines which operations from the non-buffered log become redundant when new entries arrive in the RCB buffer.

Algorithm 1: Distributed algorithm (for a replica i) showing the interaction between the RCB middleware and the pure operation-based CRDT framework

```
state: $s_i := \emptyset$
on $operation_i(o)$ :
    $\text{broadcast}_i(o)$
on $deliver_i(t, o)$ :
    $s_i := (s_i \setminus \{(t', o') \mid \forall (t', o') \in$
      $s_i \cdot (t', o')\ \mathrm{R\_}\ (t, o)\}) \cup \{(t, o) \mid (t, o)\ \not\mathrm{R}\ s_i\}$
on $buffer_i(t, o)$ :
    $s_i := s_i \setminus \{(t', o') \mid \forall (t', o') \in s_i \cdot (t', o')\ \mathrm{R}_\beta\ (t, o)\}$
on $stable_i(t)$ :
    $\text{stabilize}_i(t, s_i)[(\perp, o)/(t, o)]$
```

The interactions between the RCB layer and the pure operation-based framework are listed in algorithm 1, which builds on the original algorithm by Baquero et al.. It describes the interactions that arriving operations have on the state of a CRDT replica. In the case of a pure operation-based CRDT, the state (s_i) is initially an empty log. When an operation is applied locally, it will be broadcasted to all replicas. Depending on whether other causally dependent operations have yet to arrive `deliver` or `buffer` may be invoked on the receiving replicas. These operations, in turn, use the R, $\mathrm{R\_}$ and R_β relations to check what log entries become redundant (and modify the state accordingly) and if the new operation is redundant itself. If the new operation has no missing causal dependencies and is not redundant, it will be added to the log (along with its logical timestamp). Whenever a particular timestamp is causally stable, `stable` will be invoked and the pure op-based framework will compact stable operations that are returned by the `stabilize` function (which is data type dependent).

The algorithm only describes the interaction between the RCB middleware and pure operation-based framework. We will now describe how actual CRDTs are build on top of this, and in what way users can interact with them.

Table 3. Modified semantics for the Add-Wins pure-op set, supporting incomplete PO-Logs (based on approach in [2])

Framework	$(t, o)\ \mathrm{R}\ s$	=	$\text{op}(o) = (\text{clear} \vee \text{remove})$
	$(t', o')\ \mathrm{R\_}\ (t, o)$	=	$t' < t \wedge (\text{op}(o) = \text{clear} \vee \text{arg}(o) = \text{arg}(o'))$
	$(t, o)\ \mathrm{R}_\beta\ (t_\beta, o_\beta)$	=	$t < t_\beta \wedge (\text{op}(o_\beta) = \text{clear} \vee \text{arg}(o_\beta) = \text{arg}(o))$
	$\text{stabilize}(t, s)$	=	s
User	$\text{toList}(s, s_\beta)$	=	$\{v \mid (\_, [\text{op}=add, \text{arg}=v]) \in s\} \cup \{v \mid (\_, [\text{op}=add, \text{arg}=v]) \in s_\beta\}$
	$\text{add}(e)$	=	$\text{operation}([\text{op}=\text{add}, \text{arg}=e])$
	$\text{remove}(e)$	=	$\text{operation}([\text{op}=\text{remove}, \text{arg}=e])$

3.2 Reactive Pure Operation-Based Sets

Table 3 shows a modified reference implementation for the pure op-based Add-Wins set (AW-Set) CRDT using our approach. The table is grouped as follows: (1) functions that are used by the pure operation-based framework that dictate the interaction between new operations and entries in the log, and (2) procedures that can be invoked by the user for state serialisation or mutations.

In the case of the AW-Set, R_β is equivalent to $\mathrm{R\_}$, i.e. a (causally) older operation is redundant if it shares the same arguments with a newer operation, or if the newer operation is a clear operation. As R_β also encodes semantics for newer operations (albeit for buffered operations), it will typically be equivalent to $\mathrm{R\_}$ for most data types.

Table 4. Modified semantics for the RW-Wins pure-op set, supporting incomplete PO-Logs (based on approach in [2, 4])

Framework	$(t, o)\ \mathrm{R}\ s$	=	$\mathrm{op}(o) = \mathrm{clear} \lor (\mathrm{op}(o)=\mathrm{add} \land \exists(t', [\mathrm{op}=\mathrm{remove}, \mathrm{arg}=\mathrm{arg}(o)]) \in s \cdot t \sim t')$
	$(t', o')\ \mathrm{R\_}\ (t, o)$	=	$(t' < t \land ((\mathrm{op}(o) = \mathrm{clear} \land \mathrm{op}(t') = \mathrm{add}) \lor \mathrm{arg}(o) = \mathrm{arg}(o')) \lor (t \sim t' \land \mathrm{op}(o) = \mathrm{remove} \land \mathrm{op}(o') = \mathrm{add} \land \mathrm{arg}(o) = \mathrm{arg}(o'))$
	$(t, o)\ \mathrm{R}_\beta\ (t_\beta, o_\beta)$	=	$\mathrm{R\_}$
	$\mathrm{stabilize}_i(t, s)^*$	=	$\{(t', o) \mid \forall(t', o) \in s \cdot t \neq t'\} \cup \{\forall(\bot, [\mathrm{op}=\mathrm{add},\mathrm{arg}=e]) \mid (t', [\mathrm{op}=\mathrm{add},\mathrm{arg}=e]) \in s \cdot t = t'\}$
User	$\mathrm{toList}(s, s_\beta)$	=	$\{v \mid (\_, [\mathrm{op}=add,\mathrm{arg}=v]) \in s\} \cup \{v \mid (t, [\mathrm{op}=\mathrm{add},\mathrm{arg}=v]) \in s_\beta \land \forall(t', [\mathrm{op}=\mathrm{remove},\mathrm{arg}=v]) \in s_\beta \cdot t' < t\}$
	$\mathrm{add}(e)$	=	$\mathrm{operation}([\mathrm{op}=\mathrm{add}, \mathrm{arg}=e])$
	$\mathrm{remove}(e)$	=	$\mathrm{operation}([\mathrm{op}=\mathrm{remove}, \mathrm{arg}=e])$

$(o' \sim o)$ denote concurrent operations
*We assume that stabilize is only called for a timestamp when all concurrent operations are stable as well.

The `toList`[1] operation is extended to take the incomplete PO-Log segment into account (s_β, the incomplete PO-Log, is passed as an extra dependency). The fully evaluated state is the union of both the main and the incomplete PO-Log segments. Finally, `add` and `remove` are mutator procedures that will cause the corresponding operation to be broadcasted to all replicas (following the definition in algorithm 1).

Table 4 shows a modified reference implementation for the pure-op Remove-Wins set (AW-Set) CRDT using our approach. Again, R_β is equivalent to $\mathrm{R\_}$. However, unlike the AW-Set version, `toList` cannot simply take all the adds from s_β operation as it needs to account for possible removes that may invalidate the add. All adds that contain a concurrent or newer remove for a particular element will be filtered out.

4 Implementation

We implemented the proposed approach in our operation-based framework Flec [3, 4], an extensible programming framework for CRDTs written in TypeScript. Flec incorporates the concepts of ambient-oriented programming [6, 7], to discover and communicate with replicas in a distributed dynamic network. Among other things, it has support for pure-operation based CRDTs and RCB for causal delivery. In this section, we describe the modifications to Flec that are required to improve the reactivity of pure-operation based CRDTs.

Flec provides an open programming interface that allows for the implementation of pure-operation based CRDTs, using the following constructs:

- **isRedundantByOperation**: Encodes the $\mathbf{R\_}$ (or R_0, R_1) binary relation(s) (i.e. do existing log entries become redundant by a new operation).
- **isRedundantByLog**: Encodes the $\mathbf{R}$ binary relation (i.e. is a new operation redundant by an already existing log entry).
- **setEntryStable**: Perform an action when an operation becomes stable.
- **removeEntry**: Perform an action when a particular item is removed from the log (for example if it was marked redundant by `isRedundantByOperation`).
- **newOperation**: Perform an action when a new operation arrives in the log.

To build an actual (new) CRDT type, developers have to implement these methods, following the semantics of the datatype. For our reactivity extension we provide two new constructs:

- **isRedundantByBufferedOperation**: Encodes the $\mathbf{R}_\beta$ binary relation (i.e. do existing log entries become redundant by a new buffered operation).
- **newBufferedOperation**: Perform an action when a new operation arrives in the incomplete log segment.

```
onBufferedOperation(clock: VectorClock, op: O, args: any[]) :
     POLogEntry<O> {
    const entry = new POLogEntry<O>(clock, op, args);
    this.newBufferedOperation(entry);

    for (let i=this.log.length-1; i>=0; i--) {
        let e = this.log[i];
        if (this.isRedundantByBufferedOperation(e, entry,
              false)) {
            this.removeEntry( this.log[i], entry );
            delete this.log[i];
        }
    }
    this.log = this.log.filter(e =>
       typeof e !== "undefined");

    return entry;
}
```

Listing 1. A code extension to the pure op-based framework in Flec that enables reification of buffer data.

Listing 1 shows the code used to enable this reactive functionality in the pure operation-based layer of Flec. By overriding the `onBufferedOperation` method, the framework can hook into the RCB middleware. As a first step a `POLogEntry` object, an object that represents a log entry, is created with the operation data. This allows us to represent buffered RCB data as log entries. These entries are then exposed to the `newBufferedOperation`, `isRedundantByBufferedOperation` and `removeEntry` hooks, which can be used to implement

[1]In the original pure operation-based paper by Baquero et al. this would be the `eval(elems, ...)` operation.

the actual pure operation-based CRDT logic. Finally, the entry object is returned to the RCB layer, so that successive accesses to the RCB layer can immediately provide proper objects in pure op-based entry format.

4.1 Implementing Reactive Sets in Flec

In this section, we describe the implementation of Add-Wins and Remove-Wins sets using the Flec extensions described above. Listing 2 shows the implementation of an Add-Wins set, following from the proposed semantics in table 3.

```
enum SetOperation {
    Add, Clear, Remove
}
type SetEntry = POLogEntry<SetOperation>;

export class AWSet extends POLog<SetOperation> {
    ...

    isRedundantByOperation(existing: SetEntry, new_: SetEntry
        , isRedundant: boolean) {
        return existing.precedes(new_) && ( new_.is(
             SetOperation.Clear) ||
         existing.hasSameArgAs(new_) );
    }

    isRedundantByBufferedOperation =
         this.isRedundantByOperation;

    isRedundantByLog(entry: SetEntry) {
        return entry.is(SetOperation.Remove) ||
               entry.is(SetOperation.Clear);
    }

    public toList() {
        let list = {};

        this.getLog().forEach(entry =>
            list[entry.args[0]] = true);

        this.getBufferedLog().forEach(entry => {
            if (entry.is(SetOperation.Add))
               list[entry.args[0]] = true});

        return Object.keys(list);
    }

    add(e)   { this.performOp(SetOperation.Add,    [e]); }
    remove(e){ this.performOp(SetOperation.Remove, [e]); }
    clear(e) { this.performOp(SetOperation.Clear,  [e]); }
}
```

Listing 2. AW-Set implementation in Flec

The implementations of isRedundantByLog and isRedundantByOperation are a 1-on-1 mapping with the described semantics for the R and R_ relations. isRedundantByBufferedOperation, which implements the R_β relation, is set to point to the method of isRedundantByOperation as its semantics are equivalent. Finally, in toList, the main log (denoted by s) and the incomplete (buffered) log (denoted by s_β) are combined to determine the full state of the data structure.

Listing 3 shows the implementation for the RW-Set, following the semantics in Table 4. The approach is similar to the Add-Wins set, aside from that the RW-Set has some mechanisms for pruning causally stable operations. When an entry from the log becomes stable, setEntryStable (line 26-35) will remove it from the log and place it in a small compacted set (implemented as a dictionary). removeEntry, newOperation and newBufferedOperation (line 45-47) make sure that this local compacted set stays up to date when the log changes (using the cleanCompactEntries helper method). The toList method (line 44-60) is also a bit more complex for the RW-Set, as it has to take the compacted set, the main log and the incomplete (buffered) log into account while respecting the semantics of the data structure.

```
1  export class RWSet extends POLog<SetOperation> {
2      ...
3
4      isRedundantByOperation(e1: SetEntry, e2: SetEntry,
           isRedundant: boolean) : boolean {
5          return (e1.precedes(e2)
6                  && ((e1.is(SetOperation.Add) && e2.is(
                         SetOperation.Clear)) || e1.hasSameArgAs
                         (e2)))
7
8                  ||
9
10                (e1.isConcurrent(e2)
11                 && e1.is(SetOperation.Add) && e2.is(
                         SetOperation.Remove) && e1.hasSameArgAs
                         (e2));
12     }
13
14     isRedundantByLog(entry : SetEntry ) {
15         return entry.is(SetOperation.Clear) || (entry.is(
                SetOperation.Add) &&
16             !!this.log.find(e => e.is(SetOperation.Remove) &&
                                                   e.hasSameArgAs(
                    entry) &&
                    e.isConcurrent(entry)));
17     }
18
19     isRedundantByBufferedOperation =
           this.isRedundantByOperation;
20
21     setEntryStable(entry : SetEntry) : boolean {
22         let element;
23
24         if (entry.is(SetOperation.Add)) {
25             const element = entry.args[0];
26             this.compact[element] = true;
27         }
28
29         return true;
30     }
31
32     cleanCompactEntries(entry: SetEntry) {
33         if(!entry.is(SetOperation.Clear)) {
34             let element = entry.args[0];
35             delete this.compact[element];
36         } else {
37             this.compact = {};
38         }
39     }
40     newOperation          = this.cleanCompactEntries
41     newBufferedOperation  = this.cleanCompactEntries;
42     removeEntry           = this.cleanCompactEntries
43
44     toList() {
45         let list = {...this.compact};
46
47         this.getLog().forEach(entry =>
48             list[entry.args[0]] = true);
49
50         let sb = this.getBufferedLog();
51         sb.forEach(entry => {
52             if (entry.is(SetOperation.Add) &&
53                   !sb.find(e => e.is(SetOperation.Remove) &&
54                      (e.isConcurrent(entry) || e.follows(
                          entry))))
55                 list[entry.args[0]] = true});
56         // this has exponential complexity, can be reduced to
57         // linear complexity by using a dictionary
58
59         return Object.keys(list);
60     }
61
62     add(e)   { this.performOp(SetOperation.Add,   [e]); }
63     remove(e){ this.performOp(SetOperation.Remove,[e]); }
64     clear(e) { this.performOp(SetOperation.Clear, [e]); }
65
66 }
```

Listing 3. RW-Set implementation in Flec

5 Conclusion

Conflict-Free Replicated Data Types (CRDTs) are powerful tools to replicate data in a distributed system as they guarantee that eventually, all replicas end up in the same state. In this paper, we show how causal ordering, used by many CRDT frameworks, is not always desirable and may lead to less reactive CRDTs. We introduce a technique that circumvents the problem by the reification of the operation buffer, which can potentially be used by any CRDT framework relying on reliable causal broadcasting. We apply our approach to the pure operation-based CRDT framework and explain how it leads to improved reactivity and potentially lowered resource usage. Finally, we describe and implement reactive Add-Wins and Remove-Wins sets on top of the extended pure operation-based CRDT framework.

Discussion and future work. The R_β relation only tackles the redundancy of operations in the main log. It might be interesting to explore if operations from the buffer could be made redundant before they are moved out. This would introduce an extra improvement on the reactivity of the CRDTs. Implementing this is a bit more involved, however, as removing elements within the RCB middleware can lead to subtle problems. Additionally, unlike for the main log, items do not arrive in causal order. This means that extra causal bookkeeping may be needed to track operations - possibly undoing the extra benefit.

Right now developers can decide to include or ignore particular entries from the buffer when computing the state of the CRDT (such as done in the `toList` function of the RW-Set). With this, theoretically, the developer could still achieve the same reactivity that could be gained by having a relation that defines if buffer entries are redundant. Future work will include exploring what method of expression allows for the least complexity.

References

[1] C. Baquero, P. S. Almeida, and A. Shoker. 2014. Making Operation-Based CRDTs Operation-Based. In *Distributed Applications and Interoperable Systems*, Kostas Magoutis and Peter Pietzuch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 126–140.

[2] C. Baquero, P. S. Almeida, and A. Shoker. 2017. Pure Operation-Based Replicated Data Types. *CoRR* abs/1710.04469 (2017). arXiv:1710.04469

[3] J. Bauwens and E. Gonzalez Boix. 2020. Flec: A Versatile Programming Framework for Eventually Consistent Systems. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20)*. Association for Computing Machinery, New York, NY, USA, Article 12, 4 pages. https://doi.org/10.1145/3380787.3393685

[4] J. Bauwens and E. Gonzalez Boix. 2020. *From Causality to Stability: Understanding and Reducing Meta-Data in CRDTs*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/3426182.3426183

[5] K. P. Birman and T. A. Joseph. 1987. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.* 5, 1 (Jan. 1987), 47–76. https://doi.org/10.1145/7351.7478

[6] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix., J. Dedecker, and W. De Meuter. 2007. AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. In *XXVI International Conference of the Chilean Society of Computer Science (SCCC'07)*. Iquique, Chile, 3–12. https://doi.org/10.1109/SCCC.2007.12

[7] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. 2006. Ambient-Oriented Programming in AmbientTalk. In *ECOOP 2006 – Object-Oriented Programming*, Dave Thomas (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 230–254.

[8] L. Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[9] N. Preguiça. 2018. Conflict-free Replicated Data Types: An Overview. arXiv:cs.DC/1806.10254

[10] M. Shapiro, N Preguiça, C. Baquero, and M. Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Technical Report 7506. INRIA.