Nested Pure Operation-Based CRDTs

Jim Bauwens ✉
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Elisa Gonzalez Boix ✉
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Abstract

Modern distributed applications increasingly replicate data to guarantee high availability and optimal user experience. Conflict-free Replicated Data Types (CRDTs) are a family of data types specially designed for highly available systems that guarantee some form of eventual consistency. Designing CRDTs is very difficult because it requires devising designs that guarantee convergence in the presence of conflicting operations. Even though design patterns and structured frameworks have emerged to aid developers with this problem, they mostly focus on statically structured data; nesting and dynamically changing the structure of a CRDT remains to be an open issue.

This paper explores support for nested CRDTs in a structured and systematic way. To this end, we define an approach for building nested CRDTs based on the work of pure operation-based CRDTs, resulting in *nested pure operation-based CRDTs*. We add constructs to control the nesting of CRDTs into a pure operation-based CRDT framework and show how several well-known CRDT designs can be defined in our framework. We provide an implementation of nested pure operation-based CRDTs as an extension to the Flec, an existing TypeScript-based framework for pure operation-based CRDTs. We validate our approach, 1) by implementing a portfolio of nested data structures, 2) by implementing and verifying our approach in the VeriFx language, and 3) by implementing a real-world application scenario and comparing its network usage against an implementation in the closest related work, Automerge. We show that the framework is general enough to nest well-known CRDT designs like maps and lists, and its performance in terms of network traffic is comparable to the state of the art.

2012 ACM Subject Classification Software and its engineering → Consistency; Computer systems organization → Distributed architectures; Software and its engineering → Synchronization; Software and its engineering → Middleware; Software and its engineering → Reflective middleware

Keywords and phrases CRDTs, replication, pure operation-based CRDTs, composition, nesting

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding Fonds Wetenschappelijk Onderzoek - Vlaanderen: FWOSB90

1 Introduction

To ease the development of geo-distributed applications, much research has studied the concept of *replicated data types* (RDTs). An RDT exposes to programmers an interface akin to that of a sequential data type while incorporating mechanisms to keep data consistent across replicas [9, 22, 14]. Conflict-Free Replicated Data Types [22, 21, 19] (CRDTs) are the most well-known family of replicated data types. CRDTs guarantee strong eventual consistency (SEC) [22] that adds to eventual consistency the guarantee of *state convergence*, i.e. if two replicas of the data type have received the same updates, they will be in the same state. This implies that replicas converge without synchronisation or conflicts because they reach the same state as soon as they have observed the same operations.

Designing new RDTs that guarantee convergence is a complex task. Only for data types for which all operations commute (e.g., counters), one can easily construct a CRDT (since regardless of the ordering in which operations are applied, the resulting state will be equivalent). A common approach to designing CRDTs is to use causal ordering for non-

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:26
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

concurrent operations and handle conflicts between non-commutative concurrent operations [19, 13, 3]. Many current designs handle those conflicts in an ad-hoc way crafted for each data type, often relying on specific meta-data to track causality and relations. For example, some CRDT designs (e.g., OR-Set) use *tombstones* to ensure that removal operations commute [22]. However, for many CRDT types, this meta-data grows unboundedly. Moreover, it is very difficult to modify existing designs (e.g., add operations to the data type, or modify the design to work with different networking assumptions). Pure operation-based CRDTs [3] aim to solve those issues and propose an approach for building operation-based CRDTs based on a Partial Ordered Log (PO-Log) of operations. The approach exposes causal information from the underlying communication middleware which can be used to enable the removal of redundant meta-data. While pure operation-based CRDTs provide a structured framework to build CRDTs, it is designed to build CRDTs for flat data structures.

In this work, we focus on the issues raised by composing CRDTs, e.g., when CRDTs are nested or more than one CRDT is combined into a new one. Composing CRDTs is non-trivial, as the convergence property of a CRDT design is made to hold for a *single* CRDT but does not necessarily hold when several CRDTs are composed into a new one. Recent work has explored what concurrency semantics can be utilised for composing designs [19] and several specific implementations exist [15, 16, 18]. Existing approaches, however, mainly follow a state-based design, in which any information on applied operations is lost during the merging process. This may result in non-sensible designs for nested CRDTs and hampers the development of CRDTs where the operation history needs to be used to improve the merging algorithm. For example, recent work [25] explores the design of a distributed file system CRDT that uses nested structures for storing filesystem metadata. They argue that to properly support authentication primitives, all semantically related authentication information needs to be combined and considered in the merging semantics.

Operation-based techniques, on the other hand, are better suited for replicating nested data structures as information on applied operations can be used to determine the optimal ordering for concurrent operations. In the context of nested structures, this means that it is less complex to relate different operations or even separate them when deciding what nested semantics for non-commutative concurrent operations are needed. To the best of our knowledge, no uniform (structured) approach exists for designing and implementing nested CRDTs, where CRDT designers can easily coordinate the interaction between nested structures, as part of the concurrency semantics of the replicated structure. In this paper, we introduce a general approach to nesting and composing pure operation-based CRDTs and propose a framework for implementing pure operation-based nested CRDTs. For this, we extend the pure operation-based CRDT framework [3] with support for nested CRDT structures. We implement our approach by extending an existing pure operation-based CRDT framework written in TypeScript called Flec [4], where we develop a portfolio of nested data structures. We demonstrate the correctness of our approach using a VeriFx implementation where we verify that the structures always remain strong eventually consistent. Finally, we implement a distributed file system based on Vanakieva et al. [25] to assess the performance of our approach in comparison to a state-of-the-art JSON CRDT implementation, Automerge [15].

To summarise, we introduce the following contributions:

- A general approach for the design and implementation of nested CRDTs, building on the work of pure operation-based CRDTs.
- A full-fledged TypeScript implementation of our approach which includes a portfolio of existing and novel pure operation-based CRDTs.

- A validation of the correctness of our nested pure operation-based framework and a portfolio of CRDTs built on this framework.
- A performance evaluation showing that our approach has reduced network usage when compared to Automerge [15].

2 Background

In this section, we provide the necessary background to understand the contributions of this work. Baquero et al. [2] introduced the pure operation-based framework for designing CRDTs in a structured way while avoiding performance issues related to the unbound growth of meta-data. They build on the idea of relying on Reliable Causal Broadcast [8] (RCB) middleware to ensure causal ordering for non-concurrent operations (along with reliable delivery) [22, 2]. Instead of manually encoding causality information as meta-data to operations, the framework exposes causality information stored within the RCB middleware to CRDT implementors. More concretely, the framework employs a partially ordered log of operations (PO-Log) constructed with the causality information of the underlying RCB middleware. The state of the data structure can be computed by observing this log, and the log can be compacted to ensure that memory does not grow unboundedly. Figure 1 shows an example of a PO-Log of an Add-Wins (AW-Set) set replica (in a system of three replicas). It contains four `add` operations, which form the state *{A,B,C}*, depicted in grey. Three of these operations include causality information from the underlying RCB middleware, i.e. they carry a vector clock.

Algorithm 1 shows the distributed algorithm describing the interaction between the RCB middleware and the pure operation-based CRDT framework. Each replica contains has a particular state (s_i for replica i), representing its PO-Log. The *operation(o)* method is called by client applications (e.g. by a CRDT implementation using the pure operation-based framework) when an operation o should be applied. It ensures that operations are broadcasted to other replicas and annotated with a logical timestamp on delivery (t in the algorithm description). It does this by invoking the *broadcast* method from the RCB layer, which broadcasts the operation with the associated timestamp meta-data to all other replicas. On delivery of these operations (and after all causal dependencies are met), the RCB layer will invoke the *deliver(t, o)* method from the pure operation-based framework, where the log (s_i) will be modified if needed.

The framework introduces the concept of *causal redundancy* to keep the log compact. The idea is that a particular operation may make existing operations in the log redundant, or that the arriving operation may be redundant itself. Rules for this can be defined by using two binary redundancy relations, R and R__. R__ defines whether an arriving operation

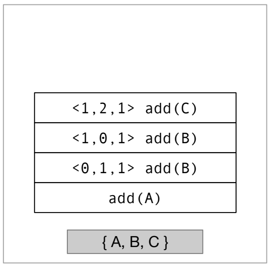

Figure 1 The internal state of an AW-Set. One operation is causally stable, and as such does not contain a timestamp. Together, the operations form the state {A,B,C}.

makes existing entries in the log redundant, and R defines if a newly arriving operation should be stored in the log. The definitions for these relations need to be provided by the concrete CRDT implementation. The framework can also determine when operations are *causally stable*, i.e., they have been observed on all replicas, and trim causal information for their log entries. Since new operations can never be concurrent with causally stable operations, their causal meta-data (such as timestamps) is thus no longer needed. The RCB layer can determine causal stability by comparing the vector clocks of incoming messages and decide whether a particular timestamp must have been observed by all nodes. Whenever a particular timestamp is causally stable, the `stable` function will be invoked by the RCB layer, and the framework will compact stable operations that are returned by the `stabilize` function. It does this by replacing (removing) the associated timestamp with the bottom (null) element. This can also be seen in Figure 1, where the `add(A)` operation has been stripped from causality information. Similarly to the redundancy relations, the `stabilize` function has to be provided by any CRDT implementation built on the framework.

Algorithm 1 (Simplified) distributed algorithm for a replica i showing the interaction between the RCB middleware and the pure op-based CRDT framework.

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

Table 1 shows the implementation for an AW-Set CRDT in the pure operation-based framework. The table is grouped as follows: (1) functions that are used by the framework and 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.

The R relation for the add-wins set defines that the `clear` and `remove` operations will never be stored in the log. $\mathrm{R}\_\_$, on the other hand, defines that an arriving operation o will make any stored operations (in the log) redundant if and only if the stored operation o' causally happened before the arriving operation (i.e $t' < t$) and the arriving operation is acting on the same set element, or the arriving operation is a `clear` (i.e., which removes all happened-before elements). For example, a `remove(X)` will make a previous `add(X)` redundant; and a `clear` operation will remove all previous log entries. The combination of both rules ensures that `add` operations will always 'win' from concurrent operations. The implementation of `stabilize` defines that all causally stable operations will be stripped from their timestamps (to preserve memory consumption). Additionally, the log will only contain distinct `add` operations at any point in time. To query the state, a map function can extract each element from these operations (as shown in the `toList` function) and serialise it into an actual set data structure.

Figure 2 illustrates the internal state and the PO-Log of the AW-Set depicted in Figure 1 after receiving a `remove(B)` operation (depicted in the a. box) and after the operation has been applied (depicted in the b. box). Initially, the log consists of an operation which is causally stable (the `add(a)`), and three other operations which are not yet stable. Looking at the vector clocks, we can observe that the log has two concurrent operations, both of which add element B. When the arriving `remove(B)` is checked against these stored operations, both

Table 1 Semantics for the add-wins pure-op set, based on the approach in [3].

Pure	$(t, o)\ \mathrm{R}\ s$	$=$	$\mathrm{op}(o) = (\mathrm{clear} \vee \mathrm{remove})$
	$(t', o')\ \mathrm{R}\_\ (t, o)$	$=$	$t' < t \wedge (\mathrm{op}(o) = \mathrm{clear} \vee \mathrm{arg}(o\ ) = \mathrm{arg}(o'))$
	$\mathrm{stabilize}(t, s)$	$=$	s
User	$\mathrm{toList}(s)$	$=$	$\{v \mid (\_, [\mathrm{op}{=}\mathit{add}, \mathrm{arg}{=}v]) \in s\}$
	$\mathrm{add}(e)$	$=$	$\mathrm{operation}([\mathrm{op}{=}\mathrm{add},\ \mathrm{arg}{=}e])$
	$\mathrm{remove}(e)$	$=$	$\mathrm{operation}([\mathrm{op}{=}\mathrm{remove},\ \mathrm{arg}{=}e])$

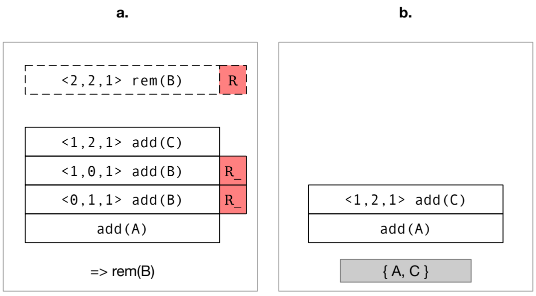

Figure 2 The internal states of an AW-Set, after receiving a remove (rem) operation, and after the operation has been applied.

previous `add(B)` operations will be marked as redundant by the $\mathrm{R}\_$ relation (as the operations have the same key, and are causal predecessors). Additionally, the arriving operation itself is immediately marked as redundant by the R relation of the AW-Set semantics (all `remove` and `clear` operations are immediately redundant) and as such, it will not be added to the log. The box denoted by *b.* shows the final result of applying `remove(B)`: no entries for adding element B remain, and the removal operation itself was not added to the log. Thus, the replica state becomes $\{A,\ C\}$.

3 Nesting Pure Operation-Based CRDTs

Currently, it is not possible to reason about nested structures within the pure operation-based CRDT framework. Redundancy relations only work on a flat level, and any logic to traverse hierarchical/nested structures would have to be manually bolted on top of the framework in an ad-hoc way.

As there is no native support for this functionality, nested designs built with the current framework require developers to store nested operations in a flattened form in the main log. To evaluate and apply the contents of the log, developers would need to either fully combine the logic of the nested and main top-level CRDT or encode the nested CRDT semantics in the query functions. In the former case, the redundancy relations and query functions would have to manage all concurrency rules for all needed nested strategies. This greatly complicates the design of such structures and makes them more prone to errors. In the latter case, only the query functions would need to be touched, but they would have to implement all redundancy logic from scratch. A programmer could delegate operations to separate components for the nested CRDTs, but in the end, this implies a reimplementation of the delivery of operations in the query function logic while this should be kept in the framework.

In this work, we propose a novel nested pure operation-based CRDT framework that enables the systematic construction of nested data structures building on the ideas of Baquero et al [2]. We explore a framework that allows developers to combine and nest existing pure operation-based CRDTs and provides constructs for the development of novel CRDTs. In particular, we focus on designs where nested structures can dynamically change at runtime, i.e., data structures that grow and shrink during the lifetime of an application, such as maps and lists, where values can be CRDTs. Our approach offers developers novel framework constructs to define the relationship between parent and child CRDT. The framework then handles all replication aspects regarding the delivery of operations in the data-structure hierarchy, ensuring that causal ordering is respected and that nested children are recursively reset when needed. In the following section, we will focus on the CRDT framework level and detail our extensions to pure operation-based CRDTs to support nesting.

3.1 Extending the Pure Operation-Based Framework

In this work, we model a nested data structure as a nested hierarchy where children can be identified by a particular key and deeply nested children by an absolute path (list of keys) relative to the topmost data structure (the root CRDT). To support nested data structures, we introduce three extensions to the pure operation-based framework:

- An internal data structure to keep track of nested CRDTs (i.e., the *children* of a CRDT).
- An update propagation mechanism for nested CRDTs that delivers the applied operations ensuring that the concurrency semantics of parent data structures are upheld.
- A reset mechanism for nested CRDT operations that ensures that the concurrency semantics of children's data structures are upheld.

Each of these extensions is essential to ensure the correctness of replicated data types. In the following sections, we elaborate on them and motivate why they are needed.

3.1.1 Keeping Track of Nested Data Structures

Objects or data structures that have nested children typically refer to children by some key. Our approach assumes that children have a unique identifier by which they can be accessed (i.e., queried and updated). As nested children can also contain other nested elements, an absolute path can be constructed to identify a particular nested data structure, starting from the root (top-most) data structure.

At the implementation level, a CRDT developer can decide in what manner key lookup works by providing an implementation of a particular handler function (`getChild`) that is used for lookup. The framework then provides a mechanism that allows absolute paths on a replicated structure to identify nested data structures that need to be queried or updated.

3.1.2 Updating Individual Nested CRDTs

When an operation needs to be applied to a nested child, the concurrency semantics of parent data structures must be upheld. Operations cannot just be immediately applied to the nested structure alone, as concurrent operations can be applied to the parent node which affects the key which points to the nested structure. For example, with a hash map, an entry could be concurrently updated, while it is being removed.

In our approach, when an update is applied to a particular child element, we will first issue special update operations to every parent node. These update operations signal the parent CRDTs that a nested operation is going to be applied and that it should be compared to existing log entries using redundancy relations. For example, when building an update-wins replicated hash map, it is important to ensure that update operations win over remove operations (on the same key). At times, the update operation itself may be immediately redundant, and as such, there is no need to propagate the operation further to a nested child.

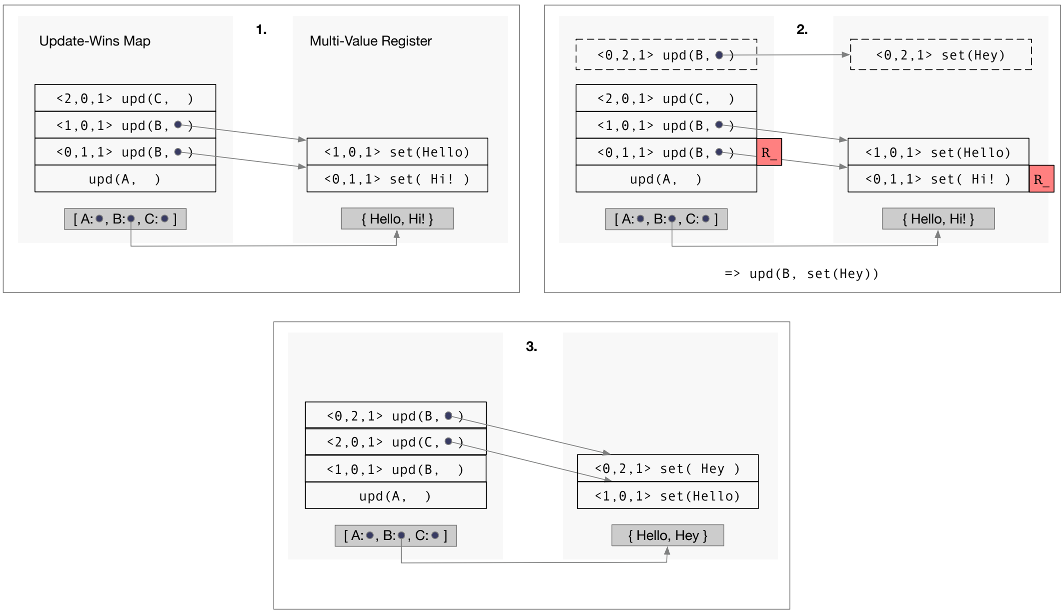

Figure 3 Three stages of the internal state of a hash-map with update-wins semantics containing nested Multi-Value registers: 1) initial state, 2) arrival of an update (upd) operation, and 3) final state after applying the operation.

To illustrate how an update is applied in our approach, consider Figure 3 showing a hash map with update-wins semantics containing nested Multi-Value registers in three different stages. A Multi-Value register (MV-Register) [22] is a replicated register that, when faced with concurrent updates, will store all concurrent values. Updates that (causally) follow will replace previous values. This is in contrast to other replicated registers, for example, the Last-Writer-Wins (LWW) CRDT register [22] that always keeps a single value. When faced with concurrent updates, an LWW-Register will use an arbitrary method for picking a single update (such as picking the update from the replica with the highest network id). The first box (denoted by 1) shows the internal state and the PO-Log for the hash map and the register associated with the key 'B'. As explained, every update applied to the nested register has an associated update in the parent log. In this case, two concurrent updates were applied to the nested register, resulting in the state `{Hello, Hi!}`.

The second box shows the state when an `update(B, set(Hey))` is applied to the hash map. This update has a timestamp (<0,2,1>) which is concurrent with some operations (<2,0,1>, <1, 0, 1>), but causally follows others (<0, 1, 1>, ..). The update itself is applied to the hash map, making one of the existing update entries redundant, i.e., the one with vector clock <0,1,1>, as it concerns the same key and has a non-concurrent timestamp. As the update operation itself is not redundant, its nested operation can be applied to the nested register. The `set(Hey)` is then applied to the nested register, making also one set operation redundant in the register, i.e., the one with vector clock <0,1,1>. Note that there is another pair of concurrent operations in both the map and register that will not be made redundant, and thus are kept in the log. The third box shows the state and the log after applying `update(B, set(Hey))` resulting in the updated state `{Hello, Hey}`.

3.1.3 Maintaining Consistency of Children by Targeted Causal Resets

Applying redundancy checks on update operations ensures that the concurrency semantics of parents are upheld. However, they do not ensure that the concurrency semantics of children are upheld. In fact, the update mechanism ensures that redundancy relations are respected at each level of the CRDT, but these redundancy checks never cross hierarchical boundaries. This is problematic if a particular key is removed, but the remove operation is concurrent with one or more, but not all, previously applied operations (for example, remove operation c is concurrent with b, operation b is concurrent with a, but operation c causally follows operation a). This means that a key and associated child cannot be removed completely, as the child received some redundant operations (by the removal, e.g., operation a) and others that are not redundant (e.g., operation b).

To solve this issue, we introduce a novel nested redundancy relation R_n that allows nested children to be reset to a particular logical timestamp (inclusive or exclusive of concurrent operations). Using this relation, redundancy rules can be implemented that define hierarchical relations between log entries.

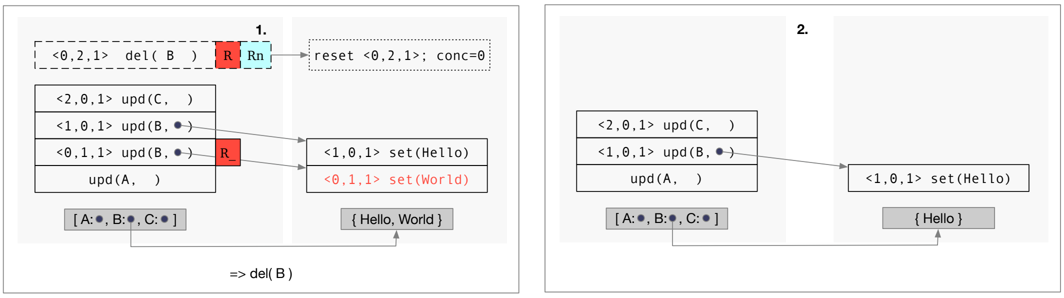

Figure 4 Example of a nested redundancy relation that selectively resets nested children, triggered by the deletion of a key. As the arriving delete (del) operation is concurrent with an update (upd) that arrived earlier, the nested child needs to be partially reset.

Figure 4 illustrates the use of the R_n relation in an update-wins hash map containing nested Multi-Value registers. The first box (denoted by 1) shows the internal state and the PO-Log for the hash map, and the register associated with the key 'B' when a `delete(B)` operation arrives. As this operation is concurrent with one of the earlier updates in the map, and the map follows update-wins semantics, the key itself cannot be removed. The entry with a preceding vector clock <0,1,1>, however, will be marked redundant by the existing $R\_\_$ relation. At this point, the register associated with key B has partially redundant data, and as such needs to be updated to respect the remove operation. To this end, the R_n relation can be used to reset all operations *in the nested register* that are previous to the delete operation. In the case of the example, the set of the value 'Hi!' (denoted in red in the figure) will be made redundant and removed from the register log. The second box shows the state and the log after applying the `delete(B)` operation in which all redundant operations are removed from the entire hierarchy, and the state of the register is updated to `{Hello}`.

In the following section, we provide a more formal specification of our approach and extensions to the pure operation-based framework and describe example implementations for update-wins and delete-wins hash maps.

3.2 Formalised Semantics for Extended Functionality

We now describe our approach as an extension of the formal model of a pure operation-based CRDTs framework (cf. Section 2). Algorithm 2 describes the distributed algorithm for our novel nested pure operation-based framework specifying the interaction between the RCB middleware and the framework. The original Algorithm 1 used the i variable to denote a particular replica. In our extended model, Algorithm 2 compounds this with a list variable p, which denotes the path to the CRDT, relative to its parent. The top-most data structure is denoted as *root*. For example, $\{root, bob, favourite\_colours\}$ could be a path that refers to a favourite_colours object associated with the key 'bob' in a map.

Compared to the original pure operation-based design, Algorithm 2 features new primitives for broadcasting and delivering nested operations:

- **broadcast_nested**$_{i,p}(o)$: broadcasts nested operations ensuring that the operation will be delivered to all replicas (reliably and in causal order). In our design, a broadcast can only be triggered from the top-most data structure, as such p will always be *root*.
- **deliver_nested**$_{i,p}(t, o)$: called when an operation o is delivered (e.g. after it was previously broadcasted) on a replica i at path p with causal clock t.
- **nested_operation**$_i(p, o)$: called when a nested operation o needs to be applied at path p.

Recall from Section 3.1.2 that when an operation is applied to a nested child, at each level of the parent hierarchy an `update` operation needs to be applied so that all redundancy rules can be activated. In the algorithm, the implementation of `nested_operation` ensures that an operation is packaged in an `update` operation and broadcasted using `broadcast_nested`. These broadcasted operations are received by the top-level data structure (root) using `deliver_nested`. `deliver_nested` will then try to deliver the operation to the child data structure specified by the path. At each level of the path, it will apply the `update` operation, check if the operation is not redundant, and if not, recursively descend into the hierarchy until the path only consists of one final child. It will then apply the actual operation to the last nested data structure using the non-nested `deliver` callback. Our approach extends the original `deliver` function with our novel nested redundancy relation: an implementation can use R_n to select what timestamps should become redundant for which nested children. Children are then (recursively) reset using the `reset` function, which takes a timestamp t and a variable *conc* that denotes whether the reset is exclusive (only entries that happened-before) or exclusive (including all concurrent entries).

In the following section, we explore how an actual nested CRDT can be built using our proposed extensions.

3.3 Nested Pure Operation-Based Maps

In this section, we illustrate our framework by describing the design of two novel nested map CRDTs: an update-wins map (UW-Map) and a remove-wins map (RW-Map).

Table 2 shows the semantics for the update-wins map (UW-Map) in our pure operation-based framework which were informally described in the examples in Section 3.1. The design of the UW-Map CRDT is inspired by the add-wins Set CRDT [3, 5], with some modifications to take care of its nested nature [19]. The R relation for the UW-Map defines that `delete` operations will never be stored in the log (i.e., they are immediately redundant). They will, however, make any existing operation in the log redundant if they happened before ($R\_$). This ensures that keys can be deleted. Note that the $R\_$ relation also makes `update`

Algorithm 2 Distributed algorithm (for a replica i) showing the interaction between the RCB middleware and the pure operation-based CRDT framework.

```
**state:** $s_{i,p} := \emptyset$
**state:** $children_{i,p}$
**on** $operation_i(o)$ **:**
    $broadcast_{i,root}(o)$
**on** $nested\_operation_i(p, o)$ **:**
    $broadcast\_nested_{i,root}(update(p, o))$
**on** $deliver\_nested_{i,p}(t, update((child, \emptyset), o)$ **:**
    $deliver_{i,p}(t, update(child))$
    $deliver_{n,child}(t, o)$ if $(t, update(child)) \cancel{\mathrm{R}}\ s_{i,p}$
**on** $deliver\_nested_{i,p}(t, update((child, p), o))$ *if* $p \neq \emptyset$ **:**
    $deliver_{i,p}(t, update(child))$
    $deliver\_nested_{n,child}(t, update(p, o))$ if
      $(t, update(child)) \cancel{\mathrm{R}}\ s_{i,p}$
**on** $deliver_i(t, o)$ **:**
    $s_{i,p} := (s_{i,p} \setminus \{(t', o') \mid \forall (t', o') \in s_{i,p} \cdot\ (t', o')\ \mathrm{R}_{\_}\ (t, o)\}) \cup \{(t, o) \mid (t, o) \cancel{\mathrm{R}}\ s_{i,p}\}$
    $reset_{i,child}(t, 0) \mid \forall child \in children_{i,p} \cdot (child, 0)\ \mathrm{R}_n\ (t, o)$
    $reset_{i,child}(t, 1) \mid \forall child \in children_{i,p} \cdot (child, 1)\ \mathrm{R}_n\ (t, o)$
**on** $stable_{i,p}(t)$ **:**
    $s_{i,p} :=\ stabilize_{i,p}(t, s_{i,p})[(\perp, o)/(t, o)]$
    $stable_{i,child}(t) \mid \forall child \in children_{i,p}$
**on** $reset_{i,p}(t, conc)$ **:**
    $s_{i,p} := s_{i,p} \setminus \{(t', o') \mid \forall (t', o') \in s_{i,p} \cdot\ ((t' \prec t) \vee (conc \neq 0 \wedge t' \parallel_c t))\}$
    $reset_{i,child}(t, conc) \mid \forall child \in children_{i,p}$
```

operations with the same key that happened before be redundant. This makes the data structure a bit more efficient. Finally, the R_n relation for UW-Map defines that all nested operations that happened before any delete need to be recursively reset (i.e. removed). As this remove should be exclusive, i.e., no concurrent entries should be removed, we additionally encode that *conc* should be zero.

Table 2 Update-wins pure operation-based map, with support for nested CRDTs.

Framework	$(t, o)\ \mathrm{R}\ s$	$=$	$\mathrm{op}(o) = \mathrm{delete}$
	$(t', o')\ \mathrm{R}_{\_}\ (t, o)$	$=$	$t' \prec t \wedge \mathrm{arg}(o\ ) = \mathrm{arg}(o')$
	$(child, conc)\ \mathrm{R}_n\ (t, o)$	$=$	$conc = 0 \wedge \mathrm{op}(o) = \mathrm{delete} \wedge \mathrm{arg}(o) = child$
	$\mathrm{stabilize}(t, s)$	$=$	s
User	$\mathrm{update}(p,\ o)$	$=$	nested_operation ([op=update, arg=[p, o]])
	$\mathrm{delete}(c)$	$=$	operation([op=delete, arg=e])

An alternative to update-wins is ensuring that delete operations are ordered after concurrent updates, leading to a map with remove-wins semantics. Note that there are different ways to implement a CRDT from a sequential data type as there is no one solution for dealing with concurrent updates. Nevertheless, it is important to offer different variants to the end-user, as some concurrent semantics may be preferred over others in particular applications.

Table 3 shows the implementation of such a remove-wins map (RW-Map) in our framework. It is structured similarly to the AW-Map but has some additional complexity as the log needs to retain all delete operations until they are causally stable. The R_n relation encodes that all previous or concurrent nested updates need to be removed (to ensure remove-wins

Table 3 Remove-wins pure operation-based map, with support for nested CRDTs.

Framework	$(t, o)\ \mathrm{R}\ s$	$=$	$\mathrm{op}(o) = \mathrm{update}$ $\wedge\ (\exists\ (t', o') \in s {\cdot} arg(o) = arg('o) \wedge op(o') = delete \wedge t \parallel_c t')$
	$(t', o')\ \mathrm{R\_\_}\ (t, o)$	$=$	$t' \prec t \wedge \mathrm{arg}(o\ ) = \mathrm{arg}(o') \wedge \mathrm{op}(o) = \mathrm{delete}$
	$(child, conc)\ \mathrm{R}_n\ (t, o)$	$=$	$\mathrm{op}(o) = \mathrm{delete} \wedge \mathrm{arg}(o) = child$
	$\mathrm{stabilize}(t, s)$	$=$	s
User	$\mathrm{update}(p,\ o)$	$=$	nested_operation ([op=update, arg=$[p, o]$])
	$\mathrm{delete}(c)$	$=$	operation([op=delete, arg=e])

semantics).

In this design of an RW-Map, in theory, `update` operations do not need to be stored in the log as these updates are stored in the nested children. However, only the last update operation for a particular child is kept (since previous update operations are removed from the log as they are redundant) As such, storing the `update` operations in the log can be useful to check if a particular child has a value, without having to query the nested children. When storing these entries poses a problem memory-wise, they can trivially be removed with no impact on the behaviour of the data type.

The implementation of these map CRDTs demonstrates that supporting nested structures can be tackled in a structured and easy way. Our framework handles all logic related to nesting and update propagation, aiming to provide an easy-to-use interface. Additionally, hierarchical redundancy rules can be encoded using the R_n relation, ensuring that concurrency semantics are upheld at any level.

3.4 Discussion

We believe that our approach simplifies the design of replicated nested CRDTs, and with it, we aim to reduce their implementation complexity. With the presented methodology, one can think of every CRDT with nesting support as a flat CRDT, which needs to support one additional operation, namely *update*. For example, a map is similar to a set of keys with an associated value. In a set, we can add and remove keys. Using some rules we can make the set add-wins or remove-wins, and with a bit of extra work, we can define how an *update* operation could be ordered against concurrent add and remove. This could be the core design of a Map. Our framework will make sure that every nested operation, e.g. a nested operation to a child of the map, is first represented as an *update* operation for the parent CRDT. The parent CRDT (e.g. the map) does not need to know anything about the nested content of this update, it is simply trying to make sure that this update will be properly ordered between the additions and removals of keys. This alone, however, is not enough to ensure convergence, i.e. that the algorithm is functional and correct. Depending on the arrival order of an update in combination with other concurrent operations, the associated nested operation may have been applied to some replicas and not to others. To ensure that the nested state converges, the algorithm sometimes might need to apply some cleanup procedures, which is precisely where the nested redundancy relation comes into play. In section 5.1 we formally prove that this is the case for our approach and our implemented designs.

4 Implementation

We implemented our novel nested pure operation-based approach in Flec [4, 6], an extensible programming framework and middleware for CRDTs written in TypeScript. Flec incorporates the concepts of ambient-oriented programming [10, 12], to discover and communicate with replicas in a distributed dynamic network. Since it has support for pure operation-based CRDTs and RCB for causal delivery, Flec is the ideal platform for implementing our approach. In this section, we describe the extensions and modifications to Flec that are required to support nested pure operation-based CRDTs.

4.1 Nesting in Flec

To support the implementation of pure operation-based CRDTs, Flec provides an open framework with the following operations:

- **isPrecedingOperationRedundant** and **isConcurrentOperationRedundant**: encode the $\mathbf{R}\_\_$ (or R_0, R_1) binary relation(s) defining if existing log entries become redundant by a new operation. Alternatively, **isRedundantByOperation** unifies both methods.
- **isArrivingOperationRedundant**: Encodes the $\mathbf{R}$ binary relation (i.e., is a new operation redundant by an already existing log entry).
- **onLogEntryStable**: performs an action when an operation becomes stable.
- **onRemoveLogEntry**: performs an action when a particular item is removed from the log (for example if it was marked redundant by `isRedundantByOperation`).
- **onAddLogEntry**: performs an action when a new operation arrives in the log.

To build an actual CRDT data type, developers have to implement these methods, following the semantics of the datatype. While `onLogEntryStable`, `onRemoveLogEntry`, and `onAddLogEntry` are not required to implement the CRDT semantics, they can help optimise a pure operation-based CRDT to use a native data structure for causally stable entries. The log, entries, and optional native data compacted structures can be queried using the following methods:

- **getLog**: gets all current log entries.
- **getState**: gets all current log entries, the compact native state, and the current logical timestamp for the replica.
- **getConcurrentEntries**: gets all concurrent log entries for an operation.

In this work, we extend the framework with the following new hooks and operations to implement nested pure operation-based designs:

- **setChildInitialiser**: is a method that will be used to initialise new children, using child-specific constructs (e.g. if you want children to be AW-Sets, the initialiser will return a new AW-Set).
- **doesChildNeedReset**: encodes the R_n binary relation (i.e., from what timestamps do children need a partial reset).
- **performNestedOp**: performs a nested operation and broadcasts it to other replicas.
- **addChild**: register a CRDT as a child to a parent, for a particular key.
- **resolveChild**: override the default internal child bookkeeping and instruct the framework on how to resolve a particular child CRDT based on a name (this will disable addChild).

4.2 Implementing Nested CRDTs in Flec

We now illustrate the extended Flec by means of the RW-Map CRDT described in Table 3. Listing 1 and Listing 2 show the core of the implementation of RW-Map CRDT in Flec. Lines 4 to 8 in Listing 1 define the CRDT constructor, which is used to initialise the `values` property that contains all nested children. Additionally, an initialiser can be specified that sets the initial (start) value for children. For example, if a map with a nested AW-Set is needed, the initializer will initialize a new AW-Set CRDT. Lines 14-16 in Listing 1 show the update function which can be used to apply nested operations on children (by CRDT client code). Any operation on a child is indicated by specifying a particular path, and the update to be applied. Using `performNestedOp` this operation will be propagated to the child and all replicas. The actual semantics can be seen in Listing 2 which shows the implementation of the redundancy relations and children referencing.

```
 1  export class RRWMap extends PureOpCRDT<MapOps> {
 2      values: Map<string, NestedCRDT>;
 3
 4      constructor(initializer: () => NestedCRDT) {
 5          super();
 6          this.values = new Map();
 7
 8          this.setChildInitialiser(initializer);
 9      }
10      ...
11      // User functions
12      ...
13
14      public update(path, ...args) {
15          this.performNestedOp("update", path, args);
16      }
17  }
```

Listing 1 The implementation of an RW-Map in Flec, using the described extensions (A).

Lines 20 to 22 in Listing 2 show the implementation of the `resolveChild` method which allows the underlying Flec framework to reference children, stored in the values property. The rest of the listing shows how the RW-Map implements redundancy relations to achieve remove-wins semantics: the RW-Map provides an implementation for `isPrecedingOperationRedundant` to implement the R__ relation: any operation in the log is redundant if it has happened before a newly arriving operation, and if they are acting upon the same child. It also implements `isArrivingOperationRedundant` to define the R relation: any arriving update is not applied if a concurrent delete is stored in the log. Finally, by providing an implementation for `doesChildNeedReset` we specify that when a delete arrives for a particular child, the child will be reset. The `reset_concurrent` flag is set to true to indicate that even concurrent updates to the child should become redundant.

```
 1  protected isPrecedingOperationRedundant(existing: MapEntry, arriving
        : MapEntry, isRedundant: boolean) {
 2      return arriving.isDelete() && existing.hasSameArgAs(arriving);
 3  }
 4
 5  protected isArrivingOperationRedundant(arriving: MapEntry) {
 6      const concurrentDeletes = this.getConcurrentEntries(arriving).
 7          filter(e => e.entry.isDelete() && e.entry.hasSameArgAs(
                arriving));
 8
 9      return concurrentDeletes.length > 1;
10  }
11
```

```
protected doesChildNeedReset(child, arriving: MapEntry) {
    return {
        condition        : arriving.isDelete() && arriving.args[0] ==
            child,
        reset_concurrent: true
    };
}

// Resolve child CRDTs
protected resolveChild(name: string) {
    return this.values.get(name);
}
```

Listing 2 The implementation of an RW-Map in Flec, using the described extensions (B).

5 Validation

To validate our work, we conduct three experiments. First, verify the correctness of our proposed framework and nested pure op-based maps. Secondly, we implement the concepts in a real programming framework and finally, we compare it to another framework featuring similar concepts.

5.1 Verification with VeriFx

In order to verify our approach, we have re-implemented the core of our nested pure operation-based CRDTs in VeriFx [11]. VeriFx is a programming language for replicated data types with automated proof capabilities that allow users to implement replicated data types in a high-level language and express correctness properties that are verified automatically. VeriFx internally uses an SMT theorem prover to search for counterexamples for each property that needs to be upheld. It also enables the transpilation of the data types to mainstream languages (e.g. Scala and JavaScript).

Correctness means that strong eventually consistent data types can be built with the framework and that they exhibit the *strong convergence* property which requires that replicas need to have received the same operations to be in the same state (regardless of the order in which the operations have been received). Shapiro et al. showed in [22] that operation-based CRDTs guarantee strong convergence if all concurrent operations commute. In our case, this implies checking the effects of all redundancy relations. Proving the correctness is, however, slightly trickier in our case, as we are dealing with a recursive design. SMT solvers, such as Z3 used by VeriFx, do not deal well with recursive and nested data structures, as they might not be able to find a solution in a finite time. To verify our approach, we thus combine VeriFx proofs with structural induction, which limits the recursion depth needed to verify our design:

- Base case: we implemented a 'perfect' resettable pure operation-based CRDT in VeriFX that can model both a flat CRDT or a CRDT containing children. The CRDT logs all operations in a single flattened log (e.g., one log for all potentially nested structures). Items in the log can be reset by a parent when requested. No redundancy rules are applied. This design ensures that we can represent a 'correct' nested structure (in terms of SMT assumptions) without needing a recursive model. We use a VeriFx proof to ensure convergence of this 'perfect' CRDT.
- Induction step: a particular nested CRDT can be implemented on top of our VeriFx implementation and set to use perfect nestable CRDTs as children. With this approach,

VeriFx can then be used to prove that our approach is correct for one level of nesting, for all pairs of operations.

By combining the base case and induction step, we prove using structural induction that our framework remains correct for any nestable structure.

```
proof FUWMap_update_update_converges {
  forall(map: FUWMap, k1:String, k2: String, t1: VersionVector, t2
      : VersionVector, o1: SimpleOp, o2: SimpleOp) {
    ( t1.concurrent(t2)  && map.children.contains(k1) && map.
        children.contains(k2) &&
                             map.polog.forall((e:TaggedOp[FMapOp])=>
                                 ((e.t.before(t1) || e.t.concurrent(t1
                                 ))
                             && (e.t.before(t2) || e.t.concurrent(t2)
                                 )))) =>: (

    map.update(t1, k1, o1).update(t2, k2, o2)
     ==
    map.update(t2, k2, o2).update(t1, k1, o1)
    )
  }
}
```

Listing 3 Convergence update-update.

```
proof FUWMap_update_delete_converges {
  forall(map: FUWMap, k1:String, k2: String, t1: VersionVector, t2
      : VersionVector, o1: SimpleOp) {
    (t1.concurrent(t2) && map.children.contains(k1) &&
     map.polog.forall((e:TaggedOp[FMapOp])=>((e.t.before(t1) || e.
         t.concurrent(t1)) && (e.t.before(t2) || e.t.concurrent(t2)
         )))) =>: (
      map.update(t1, k1, o1).delete(t2, k2)
       ==
      map.delete(t2, k2).update(t1, k1, o1)
    )
  }
}
```

Listing 4 Convergence update-delete.

```
proof FUWMap_delete_delete_converges {
  forall(map: FUWMap, k1:String, k2: String, t1: VersionVector, t2
      : VersionVector, o1: SimpleOp, o2: SimpleOp) {
  (t1.concurrent(t2) && map.children.contains(k1) && map.children.
      contains(k2) && map.polog.forall((e:TaggedOp[FMapOp])=>((e.t.
      before(t1) || e.t.concurrent(t1))
                           && (e.t.before(t2) || e.t.concurrent(t2))
                               ))) =>: {

    map.delete(t1, k1).delete(t2, k2) == map.delete(t2, k2).delete
        (t1, k1)
  }
  }
}
```

Listing 5 Convergence delete-delete.

As an example, Listings 3, 4, and 5 show the VeriFx proof logic that was used to check the behaviour of concurrent operations on an update-wins map implemented with our framework. We define that any pair of correct operations that are concurrent and applied to a correct state should commute. The operations and state are correct if the operations (causally)

CRDT	Semantics
UW-Map	Update-wins map where values can be CRDTs. Update win from concurrent deletes.
RW-Map	Remove-wins map where values can be CRDTs. Deletes win from concurrent updates.
RW-Map (mod)	Modular version of the remove-wins map that allows more efficient memory usage.
AW-Map	A variant of the update-wins Map where keys are managed by an add-wins set.
AW-Set	An add-wins set where values can be CRDTs.
DW-List	A delete-wins linked list where elements can be CRDTs.
ImmutableCRDT	A map with immutable keys, which behaves similarly to structs in C.

Table 4 Implemented nested CRDT types.

follow or are concurrent with all other operations that were applied previously to the state (e.g. everything in the log). For this definition, we assume the usage of RCB (which is the case with the pure operation-based CRDT framework), so that we know that everything in the log must be concurrent or happened-before. In other words, the logic encodes the correctness properties that should always hold in our framework, i.e. that if all operations on the map commute and the nested operations are applied to correct CRDTs (in our case, all nested operations are applied to a 'perfect' CRDT), that the map is correct.

We use the automatic VeriFx prover to verify these properties hold given the implemented designs. Internally, the VeriFx SMT engine will look for valid solutions that satisfy the negation of our definitions, it will search for any case where the correctness properties are violated. Since no counterexamples (valid solutions for the negation of properties) were found after exhausting all search options, we can then constitute that our framework model is valid according to the correctness properties.

Using this approach, we have verified our map designs, validating both the concurrency semantics of our proposed CRDTs and proving that our novel framework functions correctly. The benefit of our verification approach is that to validate the correctness of any nestable CRDT (built on our framework), one only needs to encode proofs for the operations on a flat level. All needed nesting aspects of the proof will automatically be inherited from our VeriFx implementation. The full source code for our VeriFx implementation, including proofs and implemented models, is included as an artifact.

5.2 Portfolio of Nested CRDTs in Flec

To show the flexibility and applicability of our approach, we have implemented several commonly used data structures as novel nested pure operation-based CRDTs in Flec, summarised in Table 4. As shown in the previous section, we have map implementations with update-wins and remove-wins semantics. Maps form the basis for many other data structures and thus are essential to any replication framework. They have been verified using their VeriFx-based implementations and have been used in more complex data structures since.

We have implemented two other maps: one modified map (based on the remove-wins map) that optimises some structures to have better memory resource usage, and another map where keys are managed by an add-wins set. Finally, we have a delete-wins list that can be used to store values in sequential order. Similarly to other sequential replicated structures such as RGAs [13], a linked list is used internally.

The source code for the update-wins map, remove-wins map and delete-wins list implementations can be found as part of the included artifact.

5.3 Use-Case: A Mixed CRDT-Based Distributed Filesystem

To validate our approach in a real-world application scenario, we implemented a distributed file system based on the work of [25] in our Flec implementation. This application is also used later in Section 5.4 to compare our approach to state-of-art.

Flec does not only support pure operation-based CRDTs, it has many general-purpose constructs for building any replicated data type. As such, it comes with a portfolio of (non-pure-op) general CRDTs. While our extensions to Flec were focused on pure operation-based CRDTs, part of the nesting support we added can also be used in conjunction with general non-pure operation-based CRDTs to develop real-world applications.

When composing (traditional) CRDTs, operations on a (parent) root node typically trigger several operations that will be applied to internal (nested) CRDTs. For a single operation, these sub-operations need to be applied atomically, they cannot be viewed as independent and should not automatically replicate to nested children of replicated CRDTs. This is in contrast with our main approach where an update is applied via a particular sub-path. To ensure compatibility with this approach in the framework, nested children can detect the context in which operations are applied. If a nested CRDT has a parent, and an operation is applied directly from that parent (and not via a nested update), the operation will not be broadcasted to other replicas. Instead, it is assumed that the (top-)parent operation will be broadcasted, resulting in the same nested update path on other replicas.

We now discuss the overall data structures and operations of the distributed file system. Listings 6–8 in the appendix show the core of the implementation. It has been modified to hide some minor boilerplate code, type definitions, and a lot of operation handling code, but it contains the essentials. Listing 6 shows the main body of the `DistributedFS` class, which implements the core functionality of the CRDT. By extending the `SimpleCRDT` class it automatically inherits all the distribution and CRDT functionality from Flec (along with our extensions). Lines 5-21 define the required data structures for the distributed file system that keep track of metadata for files, groups and users. To this end, we define three maps, and each map on its own contains records (in the form of ImmutableCRDT) containing other CRDTs for storing the metadata of particular files, groups and users. For example, the `files` data structure is defined using an RW-Map and contains filesystem meta-data related to access rights, ownership, and data content. The data types we use for the registers (*AccessRightF, UserID, ...*) are basic types constructed from primitive types such as numbers or strings and can be stored directly in the registers. *AccessRightF* is a numerical value that we index as a bit-vector to store our permission flags (similar to POSIX systems). We provide an additional TypeScript class, *AccessRight*, that provides a high-level abstraction to this bit-vector, but concretely we store numerical values in the CRDT register. Lines 24-28 define the `onLoaded` method which associates the aforementioned three maps with their parent CRDT. In line 30, the `setHandler` method defines all operation handlers which implement the semantics of the CRDT.

Listing 7 shows the implementation of the `CreateFile` operation in more detail. Listing 8 shows code that exposes some of the CRDT API to the local user, for performing some basic actions which are used by the `test` method in Listing 9 to show local usage of the file system functionality. Flec will ensure that all operations are properly replicated and distributed. In general, most of the code is similar to that of sequential data structures, and the API is not much more complex. This is in line with the goal of our framework: an easy-to-use interface for building CRDTs where developers can immediately benefit from a middleware that does all the heavy lifting.

5.4 Evaluation of Network Traffic in Comparison With Automerge

To compare our approach with state of the art, we implemented the same distributed filesystem in Automerge v1.0.1 [15] and evaluated the differences in network traffic between our Flec implementation and the Automerge implementation.

It is not possible to select the individual concurrency semantics for nested objects with Automerge, as is possible with our extension to Flec. As such, the implementation has a slight difference in concurrency semantics when compared to the original design [25] and our implementation. For example, while the distributed filesystem (DFS) specification describes update-wins concurrency semantics for the user list, the Automerge implementation uses remove-wins concurrency semantics. Functionality-wise, it has the same features. In fact, in our implementations, both the Automerge and Flec versions have the same API.

Automerge itself does not provide a network layer but instead provides an API that allows you to query (Automerge) documents for changes, and if any changes exist, you can propagate these over any networking channel that your application depends on. On the receiving end, you can insert these changes back into Automerge, which can merge the received information in the local state. Automerge itself uses a state-based approach, where only the required changes (deltas) are propagated instead of the full state, to conserve network bandwidth.

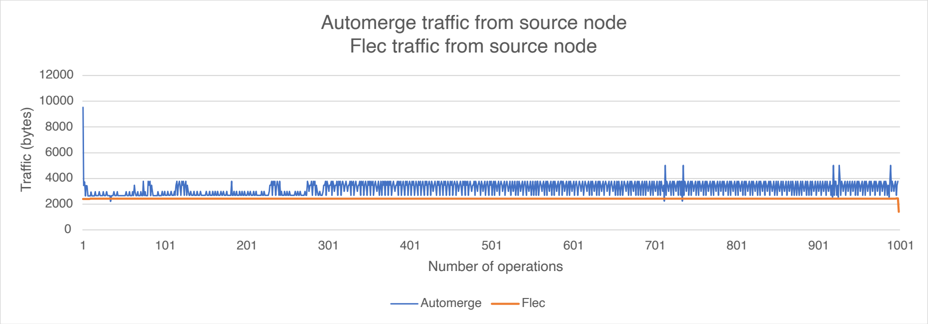

Figure 5 Network traffic (in bytes/op) originating from the source node for both Automerge and Flec. In every operation, a file is created and written.

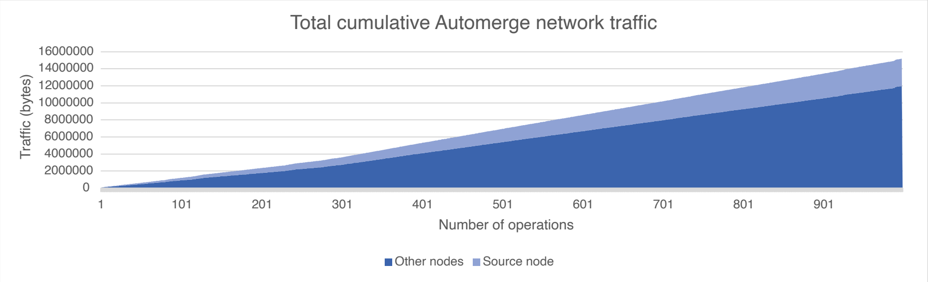

Figure 6 Total cumulative networking traffic (in bytes/op) from all nodes for Automerge. In every operation, a file is created and written.

For the experiments, we used a virtual network for both Automerge and Flec, which allows us to reproduce benchmarks and results with little non-determinism. We set up a system with 5 nodes (ad-hoc, peer-to-peer), and issue a thousand operations per experiment.

5.4.1 Experiment A: File Creation and Writing

For the first benchmark, each operation exists out of file creation and file modification. We applied these operations a thousand times to a deployed distributed file system, once using the Flec implementation and once with the Automerge implementation.

Figure 5 shows the network traffic originating from the source node (the node where the operations are applied), for both implementations. As both our approach and Automerge share the essential updates, the results are fairly stable and linear. Automerge will always send small updates containing the state delta (which means the newly modified file) and our extension to Flec sends the operations itself. While Automerge uses a binary representation for the update payload, the payload itself is still heavier than the non-optimized JSON payload used in Flec.

The visualisation hides some essential information, however. Automerge uses an additional protocol that allows replicas to propagate updates among each other. This means that not only the source node will share information, but also other nodes that received the new updates if they believe that other replicas may be missing information. Figure 6 highlights the additional traffic, showing that it makes up a significant portion of the total network traffic. In Flec updates are only sent directly from a source node to a destination node, and as such, there is no additional network usage.

5.4.2 Experiment B: User, Group, and File Creation, and Configuration

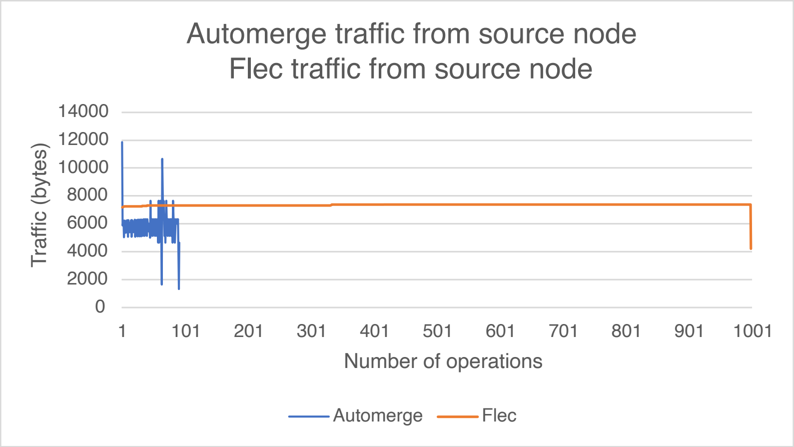

Figure 7 Network traffic (in bytes/op) originating from the source node for both Automerge and Flec. Every operation creates a new user, a new group, and a new file. The user is added to the group, and the file is created with the new user as the owner. Finally, the file is written.

For the second experiment, in each operation, we create a new user, and a new user group, add the user to the new group, create a new file (with the new user as owner), and write to this file. This extra complexity leads to some interesting results. As seen in Figure 7 the Automerge measurements stop at around ∼100 operations. This is because the additional gossip traffic starts growing exponentially (see Figure 9) and causes the entire system to halt. We are not exactly certain what causes this problem, but we did not observe this issue with

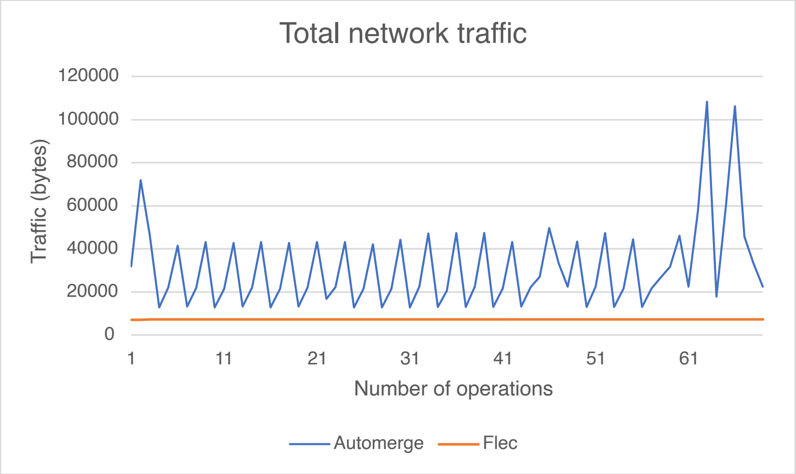

Figure 8 Total network traffic (in bytes/op) for both Automerge and Flec. Every operation creates a new user, a new group, and a new file. The user is added to the group, and the file is created with the new user as the owner. Finally, the file is written.

the previous experiment, only when we applied more complex operations. We believe that this is not correct behaviour from Automerge but have not been able to identify the root cause of the bug yet. The behaviour is consistent and reappears with each run. To be able to evaluate this example anyway, we will only focus on the initial measurements before the exponential explosion. Based on Figure 7 we can see that Automerge has a lower network overhead on the source node when compared to Flec. When looking at the total traffic, however (Figure 8), we can see that Automerge still utilizes more bandwidth. The reason for this is that as we are sending many operations, other replicas start propagating updates as well, resulting in the source node itself sending fewer updates (as it is relieved from work).

5.4.3 Experimental Evaluation: Conclusion

With this experimental evaluation, we showed that our approach is comparable to state-of-the-art CRDT frameworks, even though Flec and our extensions have not yet been optimised for non-experimental use. While additional optimisations can be applied to the pure operation-based CRDT framework and our nested framework extension, these results are promising and show that our approach is viable in real-world scenarios.

We now discuss some of the potential threats to the validity of our experimental evaluation and why our benchmark methodology and conclusions are not invalidated by these threats.

- T: The number of replicas used in our benchmarks (5) is potentially too low.
- The results of the experiments show that this number is fair, as it allows us to observe interesting differences between both benchmarked platforms. For example, in Figure 6., we can see that the total traffic generated by Automerge in experiment A quickly exceeds the traffic of our approach, but we can still compare results in a reasonable way.
- T: The chosen experiments are not realistic.
- The operations are tailored to induce complicated internal behaviour of the replicated data type, which we expect to also occur doing normal and realistic tasks. Of course, in a realistic setting such operations may not be applied repeatedly, but in the context of our evaluation we wanted to evaluate behaviour under repeated, continual usage

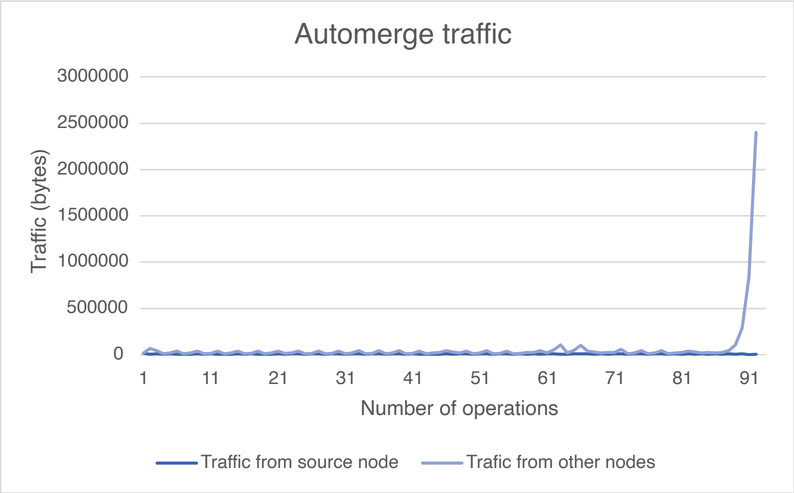

Figure 9 Total network traffic for Automerge for the previous experiment, highlighting an issue with exponential growth after a certain number of operations.

while testing many different parts of the CRDT framework as well. However, the total amount of operations used in the benchmarks could be achieved over a small period in a real deployment, and therefore it is important that a distributed filesystem system can handle such load. The operations used aim to use nesting to its full extent, in a realistic application case (a distributed file system). We, therefore, believe that the benchmarks are suitable for evaluating our approach.

- T: The benchmarks only compare results with one other related work.
- While comparing with extra platforms could improve the evaluation, we do not believe that this invalidates or diminishes our results. Automerge is a state-of-the-art framework for replicated data structures, with a lot of usages, and therefore a proper framework to compare against and evaluate whether our proposed approach has viability.

6 Related Work

The bulk of research in replicated data types has focused on devising a portfolio of conflict-free data structures such as counters, sets, and linked lists [22, 24, 20, 7, 21, 19]. However, the composition and nesting of CRDT have drawn little attention so far. The composition of replicated structures is possible in a few frameworks like Automerge [15] and Lasp [17]. While Automerge allows programmers to arbitrarily nest linked lists and maps in a document, it doesn't allow for much flexibility regarding the actual merging semantics. Lasp supports functional transformations over existing CRDTs provided in the language, which allows a composition to some extent. However, when the current portfolio of CRDTs falls short in those frameworks, developers need to design the desired nested data structure from scratch. This requires rethinking the data structure completely such that all operations commute and manually implement conflict resolution for concurrent non-commutative operations, which is hard and error-prone [22, 15, 1].

Weidner et al. [23] explore ways to compose and de-compose pure operation-based CRDTs. They introduce techniques for creating novel CRDTs based on existing (de-composed) CRDTs *with a static structure*. They do not aim to provide a solution for creating general nested data structures, but instead, propose constructs to define the semi-direct product of op-based

CRDTs. This means that instead of nesting and maintaining individual semantics, novel semantics are introduced to create a combination of several CRDTs, leading to an entirely new, non-nested CRDT. In our approach, nested data structures can change dynamically during runtime, using maps, lists, and sets.

Preguiça in [19] explains several possible nesting semantics for operation-based CRDTs. To support a wide variety of CRDTs as nested values in different settings, it will be necessary for the CRDTs to be able to partially reset themselves to an initial state before a particular timestamp. Typically, this means that this reset has to be recursive and that nested sub-CRDTs will need to be reset as well. Without a disciplined approach, combining ad-hoc CRDTs will be hard. The benefit of using a log-based approach, which we are proposing, is that such recursive resets can be supported at the framework level, in a unified way, without needing to modify the semantics of CRDTs.

Operation-based and state-based CRDTs are two approaches to guarantee SEC that share an equivalence to some extent. While both approaches can be emulated as each other [22], it depends on the application or system in use which approach might be more suitable. It is typically a tradeoff choice, between waiting for the right moment to make a state merge, or rather propagating operations continuously. It should be possible to emulate our approach (and pure operation-based CRDTs in general) as a state-based design, but making it efficient might be problematic as one would need to keep track of extra meta-data related to the applied operations (in order to maintain individual semantics between nested components). This information comes for free in an operation-based CRDT approach; as the operations themselves are directly propagated.

7 Conclusion

Conflict-Free Replicated Data Types (CRDTs) are useful programming 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 explore a structured approach for designing nested CRDTs based on the ideas of pure operation-based CRDTs. We propose a novel framework for building nested pure operation-based CRDTs and show how several common nested data structures can be designed and modelled in the framework. We validate our approach by extending an existing pure operation-based framework written in TypeScript, Flec, to include support for nested pure operation-based CRDTs and implement a portfolio of commonly nested data structures. This portfolio includes novel add-wins and remove-wins pure operation-based CRDTs, implemented following our framework. Additionally, we demonstrate the flexibility of the framework by implementing a distributed filesystem model using these techniques. We used an SMT-based implementation to verify the correctness of our approach. Finally, showed that our approach produces competitive results compared to Automerge, a state-of-the-art framework.

References

1 P. S. Almeida, A. Shoker, and C. Baquero. Efficient state-based crdts by delta-mutation. *CoRR*, abs/1410.2803, 2014. `arXiv:1410.2803`.

2 C. Baquero, P. S. Almeida, and A. Shoker. Making operation-based crdts operation-based. In Kostas Magoutis and Peter Pietzuch, editors, *Distributed Applications and Interoperable Systems*, pages 126–140, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

3 C. Baquero, P. S. Almeida, and A. Shoker. Pure operation-based replicated data types. *CoRR*, abs/1710.04469, 2017. `arXiv:1710.04469`.

4 J. Bauwens and E. Gonzalez Boix. 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, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3380787.3393685`.

5 Jim Bauwens and Elisa Gonzalez Boix. Improving the reactivity of pure operation-based crdts. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '21, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3447865.3457968`.

6 Jim Bauwens and Elisa Gonzalez Boix. From causality to stability: Understanding and reducing meta-data in crdts. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*, MPLR '20, page 3–14, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3426182.3426183`.

7 Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. *arXiv preprint arXiv:1210.3368*, 2012.

8 K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987. URL: `http://doi.acm.org/10.1145/7351.7478`, `doi:10.1145/7351.7478`.

9 S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, page 283–307, Berlin, Heidelberg, 2012. Springer-Verlag. `doi:10.1007/978-3-642-31057-7_14`.

10 T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix., J. Dedecker, and W. De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *XXVI International Conference of the Chilean Society of Computer Science (SCCC'07)*, pages 3–12, Iquique, Chile, 2007. `doi:10.1109/SCCC.2007.12`.

11 Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. Verifx: Correct replicated data types for the masses, 2022. URL: `https://arxiv.org/abs/2207.02502`, `doi:10.48550/ARXIV.2207.02502`.

12 J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in ambienttalk. In Dave Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, pages 230–254, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

13 R. Hyun-Gul, J. Myeongjae, K. Jin-Soo, and L. Joonwon. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354 – 368, 2011.

14 G. Kaki, S. Priya, KC Sivaramakrishnan, and S. Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. `doi:10.1145/3360580`.

15 M. Kleppmann and A. R. Beresford. A conflict-free replicated json datatype. *IEEE Transactions on Parallel & Distributed Systems*, 28(10):2733–2746, oct 2017. `doi:10.1109/TPDS.2017.2697382`.

16 R. Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM. `doi:10.1145/1900160.1900176`.

17 Christopher Meiklejohn and Peter Van Roy. Lasp: A Language for Distributed, Coordination-free Programming. In *17th Int. Symp. on Principles and Practice of Declarative Programming*, PPDP '15, pages 184–195, 2015.

18 Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. pages 39–49, 11 2016. `doi:10.1145/2957276.2957310`.

19 N. Preguiça. Conflict-free replicated data types: An overview, 2018. `arXiv:1806.10254`.

20 Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

21 M. Shapiro. Replicated Data Types. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia Of Database Systems*, volume Replicated Data Types, pages 1–5. Springer-Verlag, July 2017. URL: `https://hal.archives-ouvertes.fr/hal-01578910`, `doi:10.1007/978-1-4899-7993-3\_80813-1`.

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

23 Matthew Weidner, Heather Miller, and Christopher Meiklejohn. Composing and decomposing op-based CRDTs with semidirect products. August 2020. `arXiv:2004.04303`, `doi:10.1145/3408976`.

24 Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Trans. on Parallel and Distributed Systems*, 21(8):1162–1174, August 2010.

25 Elena Yanakieva, Michael Youssef, Ahmad Hussein Rezae, and Annette Bieniusa. Access control conflict resolution in distributed file systems using crdts. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '21, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3447865.3457970`.

A DFS Code Listings

This appendix contains code listings with portions from our distributed filesystem test implementation. A legend for the used types can be found in Table 5.

Class / Type	**Description**
RWWMap	Nested Remove-Wins Map CRDT.
RUWMap	Nested Update-Wins Map CRDT.
ImmutableCRDT	ImmutableCRDT map. Nested CRDT map that works as a C struct.
Register<T>	LLW-Register CRDT, containing a primitive value of type T.
AccessRightF	Alias of the 'Number' type, represents a bit vector with access flags.
AccessRight	Abstraction over AccessRightF, never stores in a CRDT, just used for easy modification of the access right bit vectors.
SimpleCRDT	Abstract CRDT class in Flec, for creating operation-based CRDTs.
GroupID / UserID / FileID	Aliases for strings that represent UUIDs.

Table 5 Legend for the TypeScript classes and types used in the DFS implementation.

```
export class DistributedFS extends SimpleCRDT<FSOperation> {
  handler: FSOperation;
  ...

  files = new RRWMap(t => new ImmutableCRDT({
    access_right_owner: new Register<AccessRightF>(),
    access_right_group: new Register<AccessRightF>(),
    access_right_other: new Register<AccessRightF>(),
    file_owner: new Register<UserID>(),
    file_group: new Register<GroupID>(),
    file_data: new Register<string>()
  }));

```

```
  groups = new RRWMap(t => new ImmutableCRDT({
    group_users: new AWSet(), // must be RW
    created: new Register<flag>()
  }));

  users = new RUWMap(t => new ImmutableCRDT({
    is_admin: new Register<flag>()
  }));
  ...

  onLoaded() {
    this.addChild("files", this.files);
    this.addChild("users", this.users);
    this.addChild("groups", this.groups);
  }

  setHandler() {
    const me = this;
    this.handler = {

      ChangeOwner(userId: UserID, newOwnerId: UserID, fileId: NodeID
          )    { ... },
      ChangeGroup(userId: UserID, newGroupId: GroupID, fileId:
          NodeID)   { ... },
      ChangeOwnerPermission(userId: UserID, newPerm: AR, fileId:
          NodeID) { ... },
      ChangeGroupPermission(userId: UserID, newPerm: AR, fileId:
          NodeID) { ... },
      ChangeOtherPermission(userId: UserID, newPerm: AR, fileId:
          NodeID) { ... },
      ...
      CreateUser(with_admin_rights: boolean, id: string) { /* ... */
           },
      CreateGroup() { /* ... */ },
      AssignUserToGroup(authorId: UserID, groupId: GroupID, userId:
          UserID) { ... },
      CreateFile(userId: UserID, groupId: GroupID, fileId: NodeID) {
           ... see listing below ... },
      WriteFile(userId: UserID, fileId: NodeID) { ... },
      ...
      update(key: string) { }
    }
  }
}
```

Listing 6 The general structure of the DFS nested CRDT, highlighting the main nested children that contain the filesystem meta-data.

```
setHandler() {
  const me = this;

  this.handler = {
   ...

   CreateFile(userId: UserID, groupId: GroupID, fileId: NodeID) {
    const user = me.users.lookup(userId) as any;
    const group = me.groups.lookup(groupId) as any;

    if (group && user && group.group_users.contains(userId)) {
     console.log("adding file");

     me.files.update([{ key: fileId, op: "update" },
     { key: "file_owner", op: "write" }], userId);
     me.files.update([{ key: fileId, op: "update" },
     { key: "file_group", op: "write" }], groupId);

```

```
    const isAdmin = user.is_admin.is(FLAG_TRUE);
    const access_owner = new AccessRight(isAdmin, true, true);
    const access_group = new AccessRight(isAdmin, true, false);
    const access_other = new AccessRight(isAdmin, true, false);

    this.files.update([{ key: fileId, op: "update" },
    { key: "access_right_owner", op: "write" }], access_owner.
        toEnum());
    this.files.update([{ key: fileId, op: "update" },
    { key: "access_right_group", op: "write" }], access_group.
        toEnum());
    this.files.update([{ key: fileId, op: "update" },
    { key: "access_right_other", op: "write" }], access_other.
        toEnum());
   }
  },
  ...
 };
}
...
```

Listing 7 Structure of the operation handling code for the DFS. Included is the code for the CreateFile callback, which can either be invoked locally or as a result of a replicated operation.

```
CreateUser(with_admin_rights: boolean) {
  const id = this.getUID();
  this.performOp("CreateUser", [with_admin_rights, id]);
  return id;
};

CreateGroup() {
  const id = this.getUID();
  this.performNestedOp("update", [{ key: "groups", op: "update" },
  { key: id, op: "update" },
  { key: "created", op: "write" }], [FLAG_TRUE]);
  return id;
};

CreateFile(userId: UserID, groupId: GroupID) {
  const id = this.getUID();
  this.performOp("CreateFile", [userId, groupId, id]);
  return id;
}
...
```

Listing 8 User API for local mutations to DFS CRDT, allowing simple modification of the DFS meta-data.

```
test() {
  const userId = this.CreateUser(true);
  const groupId = this.CreateGroup();

  this.performOp("AssignUserToGroup", [userId, groupId, userId]);

  const fileId = this.CreateFile(userId, groupId);
  this.performOp("WriteFile", [userId, fileId]);

}
```

Listing 9 Example test code for the DFS CRDT, which creates a new admin user, a new group, adds the user to a group, and then creates and writes a file with this new user.