Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI **ISIN TECHNICAL REPORT**

The Document Chain

Delta State CRDT Framework for Collaborative Applications

Author(s)

Amos Brocco[1]

Technical Report Number

2021-1

Date

May 14, 2023

[1] amos.brocco@supsi.ch

Abstract

The Document Chain is an append-only data structure which implements a delta-state conflict-free replicated data type (CRDT) for arbitrary JSON documents. Document Chains can be synchronized together to integrate changes made by different clients/users. Since each chain represents different editing states of the same document, we can consider them similar to branches in a version control repository. The chain can store its contents on a variety of backends, either local (for example, the filesystem) or remote (such as Dropbox or a Solid pod). In this paper we formally describe the Document Chain and prove that replicas can be made to converge without central coordination. Furthermore we describe an architecture for collaborative applications built upon the Document Chain and we provide an evaluation of its performance.

1 Introduction

Collaborative editing applications are expected to seamlessly merge modifications made by different users into a single coherent document. The replication process can be performed either in real-time (in so called, synchronous replication) or in an asynchronous way. In the latter case, users can work on their copy (or replica) of the document while remaining offline, and defer synchronization with other users to a later time when a network connection is available. Asynchronous replication embraces the vision of offline-first or local-first applications, which are built to work offline but can also exploit online features to let users exchange data with a server or between each other.
The development of collaborative applications requires a suitable communication infrastructure as well as algorithms, data structures and protocols for dealing with concurrent modifications, conflicts and merging strategies. The Collaborative Document project tackles on these issues by introducing a novel conflict-free replicated data type (CRDT) which supports arbitrary JSON documents and can exploit different types of data exchange and communication channels. The goal of the project is to provide a framework for the development of collaboration features into existing programs, by offering an API which is simple to use and does not require significant changes to the existing software.
For collaboration we expect to use different shared storage solutions, such as network filesystems, cloud file-sharing platforms and physical devices. It is therefore necessary to seamlessly support different backends. This variety provides data portability and ensures that data cannot be locked into a particular platform or service.
The remaining of this paper is organized as follows: in Section 2 the relevant related work in the context of conflict-free replicated data types will be explored. In Section 3 provides a formal description of the Document Chain, proving its convergence without the need for neither central nor distributed coordination. In Section 4 we discuss some implementation details, whereas in section 6 we present several evaluation scenarios to assess the quality of our solution in comparison to other relevant solutions.

2 Related Work

Data replication typically requires synchronization and coordination protocols to ensure consistency while data is modified. These protocols can be implemented either using a centralized approach (for example, a central server), or through a distributed algorithm. Each solution can be nonetheless subject to some limitations, for example it might required reliable communication between replicas as means to ensure that all update messages reach their destination. By using conflict-free replicated data types (CRDTs) it is possible to modify any of the replicas without explicit coordination, neither using a centralized service, nor by means of a distributed synchronization protocol. According to

the definition found in [1], conflict-free replicated data types must also exhibit an additional property related to consistency: when any two replica receive the same set of updates they must reach the same state.
We recognize three different types of CRDTs: operation-based, state-based and delta. With operation-based CRDTs, we deal with single update operations that specify a change or modification to the data structure and need to be propagated to all replicas. These solutions implement specific algorithms to deterministically merge modifications made on different replicas [3]. Operation based CRDTs are typically used to deal with real-time editing or high-frequency updates, such as in the context of collaborative text editing, because update messages waste little bandwidth. As pointed out in [3] operation-based CRDTs depend on reliable exactly-once causal broadcast of these updates [3]. To support the implementation of collaboration features into applications, specific libraries which implement conflict-free replicated data types have been developed. With regard to operation-based CRDTs, a relevant example is the *Automerge* [5] library, which supports not only simple data types such as counters or character arrays, but complex JSON data too. In the latter case, Automerge offers an API to modify the contents of a JSON document and to later retrieve a list of changes (operations) which can be replayed on another replica.
When dealing with low-frequency updates or single edit operations are not commutative, state-based CRDTs [12] can provide some advantages over operation-based solutions. In particular it is easier to verify the correctness of the data in a particular point in time. However, state-based solutions require more storage space and consume more bandwidth, since the size of all states can become very large [4].
To overcome these issues, delta-state CRDT have been proposed. These data types (referred to as δ-CRDT) work by disseminating updates (changesets) called delta mutations [4] instead of full-states. An important property of such delta updates is that they are idempotent, which means that they can be applied possibly several times to an existing state without affecting its consistency. This property is of particular interest when the communication channel is unreliable or it is difficult to keep track of updates which might have already been applied in the past.
In this report we detail a novel approach based on delta-states called Document Chain. In contrast to existing solutions, which typically consider only simple types [4, **?**] we aim at supporting arbitrary JSON documents. Compared to operation-based solutions we do not require explicit editing of the input data through built-in functions: instead, delta updates are automatically determined by comparing an existing state with an input JSON document. This approach has the advantage of being simpler to integrate into existing applications, since it is not necessary to explicitly record changes made to the data model (as in operation-based CRDTs). Typically, it is only necessary to implement suitable serialization and deserialinzation methods to generate a JSON representation of the internal data model of an application.
Furthermore, the proposed CRDT can store its data on a variety of backends, such as on the filesystem or in a shared folder. Multiple users can update the Document Chain independently without explicit locks while accessing the data on the network. Furthermore, each update is sequenced into a chain to maintain causality and provide a way to easily recover previous versions of the data. Conflicts are dealt with in a non destructive manner by keeping all versions and by electing a winning version using a deterministic algorithm.

3 The Document Chain

A Document Chain is a data structure which stores a collection of JSON objects that can be replicated on multiple sites and concurrently updated by each participant. Each object in the chain is

identified by a UUID and its value (or content) can be modified independently on each replica of the chain, resulting in a new local state. Modifications made to an object are dealt with by creating a new version of the object itself, which is stored alongside previous versions. Accordingly, each version can be considered as an immutable. Deletions are recorded using *tombstones*, namely values which represent a final version of an object. Therefore, a Document Chain is a grow-only data structure, where all present and past versions of the data are available. By keeping track of the history of each modification it is possible to navigate through the different states of the chain.
The Document Chain is an instance of a delta-state CRDT: in the following we will present a formal description of the data structure, and prove that under the right assumptions all replicas can be made to converge to the same state with neither explicit coordination nor synchronization from a central authority or using a distributed consensus protocol. For the sake of simplicity, the forthcoming description does not include all the implementation details and optimizations, but focuses on the essential algorithmic aspects which can be used prove convergence.

3.0.1 Key concepts

The purpose of a Document Chain is to store a collection of JSON objects. To keep track of the objects that belong to the collection, and record the history of modifications, we compute unique revision identifiers for each version. In the following, we describe how those identifiers are generated and how it is possible to keep track of all modifications.

Content hash To efficiently compare different versions of an object, their content is hashed to produce a string digest $H(x)$. The hashing algorithm is implementation-specific, however the digest is expected to be represented as a string using a binary-to-text encoding such as $Base64$. The identifier of the object, referred to as id_x, is omitted from the hash computation, so to avoid storing the same data multiple times when two versions x_m and y_n of two objects x and y have equivalent content (i.e. $H(x_m) \equiv H(y_n)$).

Partial revision string Modifications made to an object can be recorded as an ordered list of digest values $[H(x_1), \ldots, H(x_N)]$, where x_k represents the contents of the k-th version of object x. The order of the list can be maintained by prefixing the absolute position of the element in the list (a numerical *index* starting at 1) to the digest. The sequence thus becomes $[1\text{-}H(x_1), \ldots, N\text{-}H(x_N)]$, and we refer to each value $N\text{-}H(x_N)$ as a *(partial) revision string*.

Revision tree Each modification produces a new revision string. It is possible to keep track of the history of each update by maintaining a directed *revision tree* for each object in the collection. A revision tree is composed of nodes, which represent revisions, and edges, which represent a causality relation between revisions. A modification of an object at revision r_N which produces a new revision r_{N+1} is recorded by adding a new node corresponding to r_N to the tree, which is connected to another node for r_{N+1}. Revision r_N is referred to as the *parent* of r_{N+1}, whereas the latter is referred to as the *successor*. By navigating the revision tree it is possible to determine previous values or conflicting versions.

Full revision string Due to concurrent modifications (which are expected to happen on different replica of the Document Chain that are subsequently merged), the sequence of revision strings might become non-linear (we can refer to it as a *revision tree*). In this situation a revision might have more than one successor. As an example, if two users concurrently modify revision $k\text{-}H(x_k)$,

the sequence would have two different successors of that revision, namely $(k+1)$-$H(x_{k+1})$ and $(k+1)$-$H(x'_{k+1})$). To uniquely identify revisions which share the same numerical index and hash value but belong to different branches of the tree, we add a suffix (referred to as $Tail$) to each revision string (apart from the first one, which has not predecessor). The $Tail$ is computed by a determistic function T on the hash value of the preceding revision string, and must ensure that all revision strings are different (i.e the labels of the nodes in each revision tree are unique). Therefore, the *(full) revision string* r_N associated with the contents x_N becomes N-$H(x_N)$_$Tail_N$, where $Tail_N = T(H(r_{N-1}))$.

Leaf revisions and conflicts Revisions that have no successors in the tree are called *leafs*. Conflicting revisions result from concurrent modifications made on different replicas. If a revision tree exhibits only one leaf revision, the corresponding object has no conflicts. On the contrary, concurrent (conflicting) modifications can lead to the existence of multiple leafs inside a revision tree. In the simplest case, if two or more revisions share the same parent they are considered as *in conflict*. As illustrated in Figure 1, revisions *r4aa*, *r3ab*, and *r4b* are conflicting leafs in the considered revision tree.

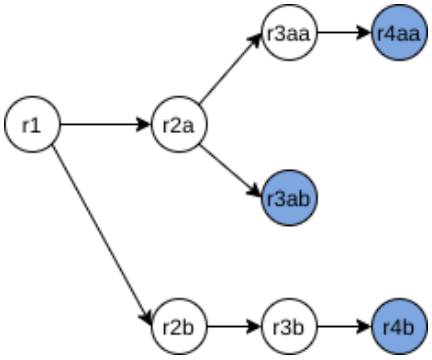

Figure 1: Leaf revisions (blue background): if there exists multiple leaf revisions, conflict arise.

Revision history Since a revision tree is connected, it can be conveniently stored as set of edges, where each edge is a tuple of revision strings. Given a revision tree, it is possible to determine the *history* of all the modifications made to an object that led to a particular revision by simply tracing a path in the tree (namely a sequence of revision strings) starting from the origin (i.e. the first revision). In the example shown in Figure 2, the revision history for revision *r4b* is *[r1, r2b, r3b, r4b]*.

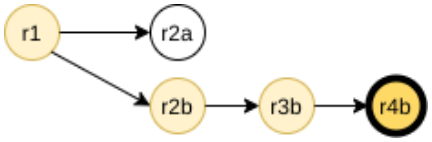

Figure 2: Revision tree and revision history for *r4b* (highlighted with a yellow background). In this example, Revision *r4b* is also the winning revision.

Winning revision The revision with the longest revision history (or the highest index) is considered the *winning revision*. In the example depicted in Figure 2, the winning revision is *r4b*. If multiple

revisions have the same index value, revision strings are compared in lexicographic sort order, and the highest is deterministically chosen as the winning one. For example, in Figure 2, the history of both *r4aa* and *r4b* has a length of 4, but *r4b* can still be elected as the lone winner. The winning revision is used to determine the contents that shall be returned when querying for the latest version of an object and to identify the revision that will be updated when the content is modified. By definition, a winning revision is also a leaf revision.

3.1 The Document Chain as a labeled rooted tree

The (full) state of a Document Chain (which comprises the history of all modifications) can be described by a tuple $\langle D, O \rangle$, where D is a map which associates the UUID of an object with its revision tree (the set of edges), and O is a map which associates $H(x)$ with a value x representing the contents of a specific version of an object.
The D map can also be represented as a labeled rooted tree, where first-level vertices (connected to the root) represent the objects (which are labeled according to their UUID id_x), and descendants of those objects are the corresponding revision trees.
Similarly, the O map can also be represented as a labeled rooted tree, with first-level vertices labeled according to the hash-value $H(x)$ of some content x: each first level vertex is connected to exactly one second level vertex which contains the corresponding content.
With this representation, the full state $\langle D, O \rangle$ of a Document Chain can be translated into a labeled rooted tree (with a root node R). The initial (empty) state of such a graph is $\langle V, E \rangle$ where $V = \{R, D, O\}$ and $E = \{(R, D), (R, O)\}$.

3.2 The Document Chain as set of tuples

Edges in the aforementioned tree can be represented by means of tuples $\langle X, Y \rangle$. To ensure that edges belonging to different revision trees are recognizable as such, we replace *revision strings* with *universal revision strings* $r_N^{id} = id_x : N\text{-}H(x_N)\_Tail_N$, where id_x is the *universal unique identifier* (UUID) of the object x. Accordingly, a Document Chain can be uniquely represented by a set of such tuples.

3.3 The Document Chain as a state-based CRDT

According to [4], a state-based CRDT consists of a triple (S, M, Q), where S is a join-semilattice (a set with partial order $\sqsubseteq$ and a binary *join* operation $\sqcup$ which returns the least upper bound of two elements in S while being commutative, associative, and idempotent), Q is a set of query functions (for reading the data), and M is a set of mutators that update a state $X \in S$ to produce a new state $X' = m(X)$ such that $\forall m \in M, X \in S : X \sqsubseteq m(X)$. We will now try to map the structure and operation of the Document Chain into this definition to prove that it represents a state-based CRDT.
The comparison operation used to obtain a partial ordering can be simply mapped to a subset $\sqsubseteq$ relation. The *join* operation $\sqcup$ can be assigned to a union between two sets of edges E and E', which results in $E'' = E \cup E'$. By its very nature, the union fulfills the requirement of being commutative, associative, and idempotent. Moreover, given two states E, E' (both in S) we can always find a state $E'' \in S'$ such that $A \sqsubseteq E'', \forall A \in \{E, E'\}$, hence E'' is the least upper bound of $\{E, E'\}$, making S a join-semilattice.
Concerning mutators, by representing the Document Chain as a tree, any modification translates into adding one edge to the state. An update of state $X \in S$ can thus be formalized as $E'_X = E_X \cup \{e\}$, where e is the edge to be added, and E_X,E'_X are the sets of edges of the current state X and the

resulting state X' respectively. Mutators are therefore *inflations* [4], and the requirement $X \sqsubseteq m(X)$ holds $\forall m \in M, X \in S$.
Because the set union operation $\cup$ is commutative, associative, and idempotent, the Document Chain is a state-based CRDT. This should not come as a surprise to anyone, given that the former is isomorphic to a G-Set (Grow-only Set [12]).

3.4 The Document Chain as a δ-CRDT

Propagating the full state of the Document Chain (and other state-based CRDTs) in order to update all replicas is an expensive operation. In this regard, Delta State-based CRDT (δ-CRDT, [4]) employ *deltas* (fine-grained states), which are comparatively smaller than full-states, while ensuring convergence as with state-based CRDTs.
According to [4], a δ-CRDT consists of a triple (S, M^δ, Q), where S is a join-semilattice of states, M^δ is a set of delta-mutators, and Q is a set of query functions. The state transition at each replica is given by either joining the current state $X \in S$ with a delta-mutation ($X' = X \sqcup m_\delta(X)$), or by joining the current state with some received delta-group D ($X' = X \sqcup D$). Delta-mutators are defined as functions, corresponding to an update operation, which take a state X in a join-semilattice S as parameter and return a delta-mutation $m_\delta(X) \in S$. Finally, a delta-group is inductively defined as either a delta-mutation or a join of several delta-groups.
Each revision update represents one or more edges to be added to the state graph, hence in our case we have $X' = m_\delta(X) = X \sqcup m_\delta(X)$, where $X, X' \in S$ and $m_\delta(X) \in S$ is the delta mutation. By grouping delta-mutations into delta-groups D, the relation $X' = X \sqcup D$ holds when $D = m_\delta(X)$ and by associativity can be extended to a join of several delta-groups. Delta-mutators simply *joins* a set of updates to an existing state.
Since the Document Chain is equivalent to a δ-CRDT grow-only set, as proposed in [4], it is possible to achieve state convergence by ensuring that all delta-mutations generated in the system reach every replica.

3.5 State serialization

The serialization of the Document Chain considers both the formal model as described in the previous section, and additional functional requirements. More specifically, we want to ensure that each modification can be traced back to its author, and that it is possible to navigate through the history of a Document Chain in order to retrieve specific versions of the data. Unfortunately, the delta-state decomposition discussed in Section 3.4 fails to capture the causality between updates. Hence, we propose to decompose the state into a chain of *revision update record blocks* (each representing a sequence of updates) alongside with *data packs*.

3.5.1 Revision update record blocks

When an object is modified, a corresponding *update record* is produced. When the full state is updated, multiple records might be generated: we thus consider sequences of records that we call *revision update record blocks* (or simply *blocks*). Blocks are propagated to other replicas to update their state, either by direct connection or by means of a common storage.
We assume that the contents of a new version of an object x can be retrieved by means of the corresponding hash value ($H(x)$), hence update records only describe the modification in terms of an edge to be added to the revision tree of that particular object (and, in general, to the full state of the Document Chain). We recognize two ways of representing an update record: a *compact* form,

and a *complete* form. The compact form requires that updates are causally applied, whereas the full form has no such requirement.
In the complete form, an update record represents a finite path (starting from the root R) inside the state graph, such as, for example $(R, D, id, r_1^{id}, r_2^{id}, \dots r_{N-1}^{id}, r_N^{id})$, where id is the UUID of the modified object. To reduce the actual space required by such a record, each revision $r_N^{id} = id : N\text{-}H(x_N)\_Tail_N$ could be replaced with $H(x_N)$ by omitting several information which can be easily recovered: the object's identifier id (which is already part of the path), the index prefix $N-$ (which can be reconstructed by considering the current position inside the path), and the $\_Tail_N$ suffix (which can be computed from the preceding revision string). Furthermore, R and D could be omitted as we are implicitly considering updates to a revision tree. Without loss of information, the update record can be therefore stored as $(R, D, id, H(x_1), H(x_1), \dots H(x_{N-1}), H(x_N))$. When processing such an update record all vertices (corresponding to objects or revision strings) and edges along the path that do not yet exist must be created.
Using the compact form, we recognize two types of records: creation records and modification records. A creation record is a tuple in the form $\langle id_x, H(x_1)\rangle$, where id_x is the UUID of an object x, and x_1 is the first version of x: these values can be used to compute a revision string r_1^{id}). A modification record is a triple in the form $\langle id, H(x_N), r_{N-1}\rangle$, where id_x is the UUID of the modified object x, $H(x_N)$ is the hash of the updated version of x and r_{N-1} is the revision upon which the modification is made. These values are used to compute a new revision of the object r_N^{id}. Modification records represent both updates to a new version, and deletions (marked with a *tombstone* revision).

3.5.2 Data packs

The content of each object referenced by an update record is stored within *data packs*. Each object is uniquely identified by the hash value of its contents, thus avoiding duplicates. To efficiently enumerate the objects contained in a *data pack* and ease content retrieval, an *index* is generated alongside each *data pack*. The index maps the hash value of an object to a position inside the pack file: it is therefore possible to enumerate all the objects inside a pack or to extract a specific object from a pack by performing partial reads. Pack files and the corresponding indices share the same stem (the name of the file without the extension), which is generated by hashing the contents of the pack file. For simplicity, *data packs* can also be parsed as arrays of JSON documents, therefore the corresponding indices can be recovered if necessary. Indices also provide an advantage in a distributed environment, since their size is typically smaller than the corresponding pack. Furthermore, both packs and indices are immutable, and can be cached locally to reduce the communication overhead. To keep track of newly created data, each *revision update record blocks* also stores a reference to the corresponding *data pack*: this allows for determining all new objects that have been created during that update.

3.5.3 Block ordering

In order to ensure causal consistency between modifications, we introduce an ordering relation that applies to any pair of block. This allows for determining the order of each update and thus maintain an history of modifications. It should be noted that this ordering is not strictly necessary to ensure the convergence of the Document Chain.
Each block is linked to the previous ones in the chain (according to the edit history) by means of a hash reference (which is computed by hashing the contents of the block), as shown in Figure 3. We employ the plural form "ones" because the sequence of blocks can become non-linear when multiple

replicas are concurrently updated (i.e. new blocks are independently added on different branches and on different replicas) and those updates are disseminated asynchronously to other replicas. The referenced blocks are referred to as *anchors*. Anchors are determined by looking at the current local chain and correspond to all blocks that haven't yet been referenced by any other block.

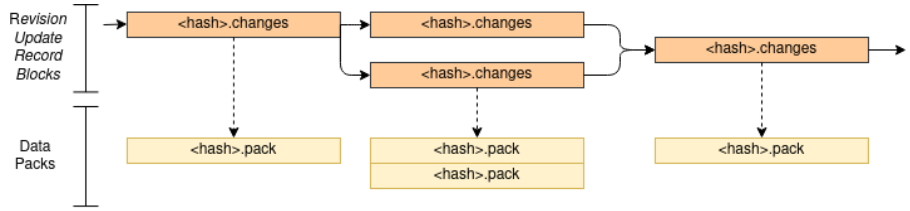

Figure 3: The Document Chain - Logical overview: Revision Update Record Blocks are denoted using the *.changes* suffix, whereas Data Packs use the *.pack* suffix; arrows represent *hash references* between elements in the chain.

With the addition of a binary relation $\leq$ between a block w and its ancestors (represented as a set $Ancestors(w)$) such that $a \leq b, \forall a \in Ancestors(b)$, the set of blocks becomes partially ordered. The relation between blocks also simplifies the process of determining missing data which has not been delivered to a replica, improving the consistency of the structure.
In practice, we construct a rooted tree of all blocks, where each block is uniquely identified by the hash of its contents and edges are determined by means of hash-references. Each edge also represents a state of the Document Chain, which can be reconstructed by joining all the update blocks starting from the root of the graph (which is called the origin block). To ensure convergence, this partial ordering is however insufficient: the set of states S also needs to form a join-semilattice, because block the aforementioned $\leq$ relation does not apply to blocks on different branches of the tree.

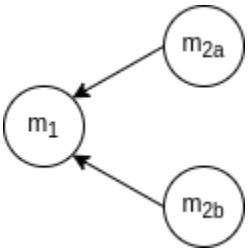

Figure 4: The reference to an ancestor block (represented by an arrow) is not enough to define a join-semilattice: in this example, given $S = \{m_{2a}, m_{2b}\}$, $\nexists\, x \in S : sup\ S = x$, i.e. there is no least upper bound.

Consider the example graph shown in Figure 4, which assumes a Document Chain results from the joining of concurrently updated replicas: block m_{2a} and m_{2b} have no least upper bound, therefore the graph does not represent a join-semilattice. To overcome this issue we introduce an additional ordering between blocks based on a lexicographical comparison $<$ on their identifiers: with this additional ordering (which is deterministic and agreed upon by all replicas), given a set S of revision update record blocks, for all x and y in S, a least upper bound of the set $\{x, y\}$ always exists. More specifically, the example in Figure 4 can be transformed into a complete-semilattice (i.e. a join and meet semilattice), as shown in Figure 5 (where m_{2b} becomes the least upper-bound of $\{m_{2a}, m_{2b}\}$, and m_1 is the greatest lower-bound), and the blocks form a totally ordered set (which is also called

a chain).

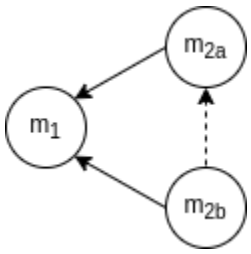

Figure 5: By introducing a lexicographical relation $<$ between block identifiers (represented by a dotted arrow, where $x \rightarrow y \Leftrightarrow y < x$), given $S = \{m_{2a}, m_{2b}\}$, we have a least upper bound $sup\ S = m_{2b}$.

3.5.4 Conflict-free modification and replication

The Document Chain supports lock free concurrent modifications as long as the backend storage (which will be discussed in the forthcoming section) allows for concurrent appending of new data. The replication process can also be performed without any locking mechanism: chains can be replicated by simply merging two or more collections of blocks, packs and indices together. For example, if blocks and packs are stored as files inside a directory, replication to a target chain is achieved by simply copying files between the two locations. Because files are named after the hash value of their content, files that already exist in the target directory can be ignored. At any time, the integrity of the chain can be verified: thanks to hash references inside blocks it is possible to determine wheter the chain is complete or not, and identify the missing or corrupted blocks or packs. If digital signatures are employed, the authenticity of the data can also be verified.

3.6 Supporting hierarchical JSON documents

As discussed in the previous sections, a Document Chain is a data structure which stores a collection of JSON objects. Each object is individually versioned and represent the smallest unit of data. Forcing client application to map their internal data model to such a collection is often impractical and might hinder the adoption of the proposed concept. Moreover, contrary to operation-based CRDTs such as *Automerge* [5] our approach strives to avoid requiring explicit editing of the input data through a specific API. We therefore propose a reversible data transformation algorithm which processes an arbitrary input JSON document and decomposes it into a collection of individual objects. At any time, the state of the Document Chain can be processed to recreate the original structure of the document. This process requires that the root of the input document is an object: in situations where this requirement is not originally fulfilled (for example, if the root of the document is an array), the input can should be refactored to satisfy this condition. In the following the procedures for updating the state from a structured input document and for transforming (reading) the state back into the original structure are presented. Maps (*associative arrays*) are denoted with curly braces { }; square brackets [ ] are used to indicate both arrays and access to a specific field of a map. Methods (whose goal should be self-explanatory) are invoked using the dot notation.

Algorithm 1 Flattening Procedure

```
1:  Objects := {}                                              ▷ Map of extracted objects
2:  procedure FLATTEN(value, path := [])
3:      switch value.type() do
4:          case String
5:              string := value.as_string()
6:              return STRING_PREFIX + string
7:          case Array of Object
8:              array := value.as_array()
9:              order := []
10:             for each object ∈ array do
11:                 id := MAKEIDENTIFIER(object, path)
12:                 FLATTEN(object, path)
13:                 order.append(id)
14:             end for
15:             ordering := {}
16:             ordering[ORDER_FIELD] ← order
17:             Objects[path] ← ordering
18:             return ARRAY_PREFIX +path
19:         case Array
20:             array := value.as_array()
21:             newarray := []
22:             for each value ∈ array do
23:                 newarray.append(FLATTEN(value, path))
24:             end for
25:             return newarray
26:         case Object
27:             object :=value.as_object()
28:             for each [key, value] ∈ object do
29:                 p := path.append(key)
30:                 t := FLATTEN(value, p)
31:                 object[key] ← t
32:                 Objects.append(t)
33:             end for
34:             return OBJECT_PREFIX + path
35:         default return value
36: end procedure
```

Algorithm 2 Creation of Revision Update Record Blocks

```
1: procedure MakeRecordBlock(Objects, ⟨D, O⟩)
2:     Processed := []
3:     block := []
4:     for each [id, content] ∈ Objects do
5:         if id ∈ state then
6:             h := Hash(content)
7:             w := D[identifier].winning_revision()
8:             wh := w.hash()
9:             if h ≠ wh then
10:                O[h] ← content
11:                r := MakeRevision(w,h)
12:                ur := MakeRecord(UPDATE, id, w, r)
13:                D ← D ∪ {ur}
14:                block.append(ur)
15:            end if
16:            Processed.append(id)
17:        else
18:            h := Hash(content)
19:            O[h] ← content
20:            r := MakeRevision(w,h)
21:            ur := MakeRecord(CREATE, id, w, r)
22:            D ← D ∪ {ur}
23:            block.append(ur)
24:        end if
25:    end for
26:    for each id ∈ State do
27:        if id ∉ Processed then
28:            w := state[identifier].winning_revision()
29:            t := MakeTombstone(w)
30:            ur := MakeRecord(DELETE, id, w, t)
31:            D ← D ∪ {ur}
32:            block.append(ur)
33:        end if
34:    end forreturn block
35: end procedure
```

3.6.1 Destructuring process (Update State)

Our solution employs a two-step difference detection algorithm (derived from [10]) to determine the changes between the input data (an arbirary JSON document which represents the new state) and the latest available state of the CRDT. In the first step (summarized in Algorithm in 1), the hierarchical structure of the input document is recursively *flattened* to extract nested objects contained within arrays (of objects) and other objects. Each object is assigned a unique identifier (according to some user-specified rules or using a deterministic algorithm which uses the path of the object inside the hierarchy). To ensure that this procedure is reversible, extracted objects are replaced with string references (all other strings are escaped to avoid conflicting with those references). To be able to reconstruct arrays (during the *unflattening* process), the sequence of identifiers of each object inside destructured arrays is stored in an *ordering* object. The *STRING_PREFIX*, *ARRAY_PREFIX*, and *OBJECT_PREFIX* values are implementation dependent character strings used to distinguish between different types of references, whereas *ORDER_FIELD* is the key to be associated with the ordering sequence array. The MAKEIDENTIFIER function is implementation specific, and is supposed to deterministically generate or obtain a unique identifier for the given object (typically by combining one or more user-defined fields).
In the second step (Algorithm 2), all extracted objects are compared against the current state $\langle D, O \rangle$ of the Document Chain. When changes are detected, this comparison produces a corresponding revision update record: if the object is not found in the current state a *create* record is appended to the block, whereas if the object is found, the contents are compared to determine if a modification took place and an *update* record needs to be created. Objects in the chain that have disappeared in the input documents are considered as deleted: in this case a *tombstone* revision is created and a corresponding *delete* record is appended to the block. The contents of a record can be either *complete* or *compact*, depending on the implementation-specific details of the MAKERECORD function. The MAKEREVISION and MAKETOMBSTONE functions generate a new revision string r_{N+1} given a revision string r_N. Finally, the HASH function is used to compute the digest of the content of an object using a deterministic hash algorithm, such as SHA-256 or xxHash. The complete update procedure is listed as Algorithm 3 (we assume that the *Objects* array is globally visible inside both the aforementioned procedures).

Algorithm 3 Update State Procedure

1: $Objects$:= {} ▷ Map of extracted objects
2: **procedure** UPDATESTATE($\langle D, O \rangle$, $Input$)
3: $Root$:= FLATTEN($Input$)
4: $Objects \leftarrow Objects \cup \{Root\}$
5: **return** MAKERECORDBLOCK($Objects$, $\langle D, O \rangle$)
6: **end procedure**

3.6.2 Reconstruction process (Read State)

The goal of the reconstruction process is to read the state of the Document Chain and rebuild the original structure of the document. The core procedure of this process is presented in Algorithm 4: starting from an initial value (typically the root of the resulting hierarchy, as shown in Algorithm 5), the unflattening procedure is recursively called to replace all references (to arrays and objects) with the corresponding value (as obtained from the current state). Ordering objects are used to restore the proper sequence of objects inside arrays, whereas non-reference strings are simply unescaped

by removing the corresponding prefix.

3.6.3 Non-destructive array merging

Structured documents that contain arrays of objects can be used to serialize collections, lists or tables. Several applications that deal with structured data are built around a list data model. Whereas items in the list can be easily serialized into JSON objects and considered as atomic data, the list itself is expected to be modified in a non-atomic way. If an object contains an array of other objects, the previously described synchronization process would easily lead to unexpected data losses when concurrent edits take place. In particular, let's consider concurrent modifications that add elements to an array. We begin with an array of three elements **[A,B,C]**, where the letters **A**, **B**, and **C** each represent an object. The *flattening* procedure creates a corresponding ordering document that stores this order. On the local replica, an user might add a new element **D** and transform the array to **[A,B,C,D]**, whereas another user might insert a new element **E** between **A** and **B** (by editing its own local replica). Therefore, there would be two conflicting ordering documents with different contents. What would be the correct or expected value of the array if the two replicas are synchronized? If no special care is taken, the resulting value would either be **[A,B,C,D]** or **[A,E,B,C]**. At the lowest level, these two versions correspond to conflicting leaf revisions. While semantically correct, the result means that one of the two users would see their newly added item disappear.

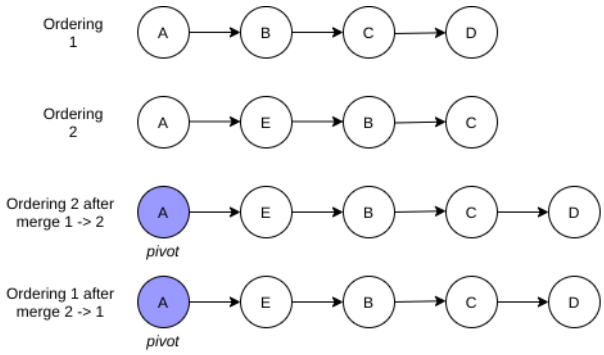

Figure 6: The Document Chain - Document Ordering Example.

The logic of the Document Chain takes care of this problem and produces a temporary new version of the array that retains both modifications, namely **[A,E,B,C,D]**, by merging arrays that belong to conflicting versions of an ordering document. This example is illustrated in Figure 6. The algorithm for merging arrays is described in Algorithm 6.
The algorithm first determines a common element between the two arrays. The first element in the source array which is also in the target array is considered as a *pivot element*: all other elements will be placed in relation to this element to maintain, as close as possible, the existing order between newly added elements.
Consider the example shown in Figure 7. When merging the ordering *2* into ordering *1*, the common element *A* is identified as the initial pivot. The algorithm iterates over the elements of the second ordering: *C* is the next common element, which becomes the new pivot. In the next step, element *D* is processed: this element is not present in the first ordering, therefore it is inserted after the current pivot. Element *D* becomes the new pivot. The same procedure is applied to element *E*, which in turn becomes a pivot.
Another example is shown in Figure 6. In this case, the pivot elements are *A*, then *C*, *D*, *E*, *F*.

Algorithm 4 Unflattening Procedure

```
1: procedure UNFLATTEN(⟨D, O⟩, value)
2:     switch value.type() do
3:         case String
4:             string := value.as_string()
5:             if string.startsWith(ARRAY_PREFIX) then
6:                 id := string.remove(ARRAY_PREFIX)
7:                 w := D[id].winning_revision()
8:                 wh := w.hash()
9:                 o := O[w.hash()]
10:                a := []
11:                for each id ∈ o[ORDER_FIELD] do
12:                    w_id := D[id].winning_revision()
13:                    wh_id := w_id.hash()
14:                    so_id := O[w_id.hash()]
15:                    a.append(UNFLATTEN(⟨D, O⟩, so_id))
16:                end for
17:                return a
18:            end if
19:            if string.startsWith(OBJECT_PREFIX) then
20:                id := string.remove(OBJECT_PREFIX)
21:                w := D[id].winning_revision()
22:                wh := w.hash()
23:                o := O[w.hash()]
24:                o :=
25:                for each [key, value] ∈ o do
26:                    o[key] ← UNFLATTEN(⟨D, O⟩, value))
27:                end for
28:                return o
29:            end if
30:            return string.remove(STRING_PREFIX)
31:        case Array
32:            array := value.as_array()
33:            na := []
34:            for each value ∈ array do
35:                na.append(UNFLATTEN(⟨D, O⟩, value))
36:            end for
37:            return na
38:        case Object
39:            o := value.as_object()
40:            for each [key, value] ∈ object do
41:                o[key] ← UNFLATTEN(⟨D, O⟩, value)
42:            end for
43:            return o
44:        default
45:            return value
46: end procedure
```

Algorithm 5 Read State Procedure

1: **procedure** READSTATE($\langle D, O \rangle$, $root_{id}$)
2: w := D[$root_{id}$].winning_revision()
3: wh := $w.hash()$
4: o := O[$w.hash()$]
5: **return** UNFLATTEN($\langle D, O \rangle, o$)
6: **end procedure**

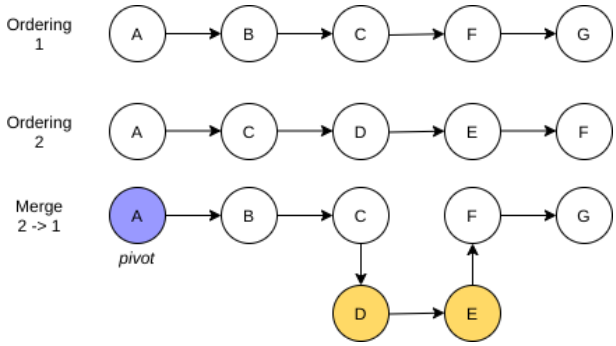

Figure 7: The Document Chain - Document Ordering Example.

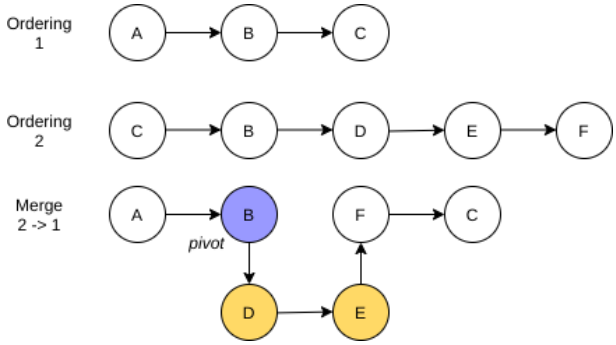

Figure 8: The Document Chain - Document Ordering Example.

4 The Document Chain as a framework

The deployment of a Document Chain entails three key elements: the implementation of the presented δ-CRDT data model, a high-level API for accessing the data, and a generic layer for managing storage and propagation of delta states. In this section a generic framework which provides all these elements will be presented and discussed.

4.1 Document chain API

The functionalities of the underlying CRDT are exposed through a high-level API which implements methods to read and update data, navigate the underlying Document Chain (to retrieve previous versions, or states), fork a new chain or merge the current state into another chain.
The minimal API is comprised of four functions, namely **update**, **read**, **replicate**, and **meld**. The update function receives an input JSON document and applies the destructuring algorithm presented in the previous sections in order to convert it into a collection of objects which represent the new

Algorithm 6 Array merging algorithm

```
1: procedure MERGEARRAYS(Source, Target)
2:     InsertPosition := 0
3:     PivotIndex := 0
4:     for each item ∈ Source do
5:         if item is in Target then
6:             InsertPosition ← Target.IndexOf(item)
7:             break
8:         else
9:             PivotIndex ← PivotIndex + 1
10:        end if
11:    end for
12:    CurrentIndexInSource := 0
13:    for each item ∈ Source do
14:        if item is in Target then
15:            InsertPosition ← Target.indexOf(item)
16:        else
17:            if CurrentIndexInSource < PivotIndex then
18:                Target.insert(InsertPosition, item)
19:                PivotIndex ← CurrentIndexInSource
20:            else
21:                InsertPosition ← InsertPosition + 1
22:                Target.insert(InsertPosition, item)
23:            end if
24:        end if
25:        CurrentIndexInSource ← CurrentIndexInSource + 1
26:    end for
27: end procedure
```

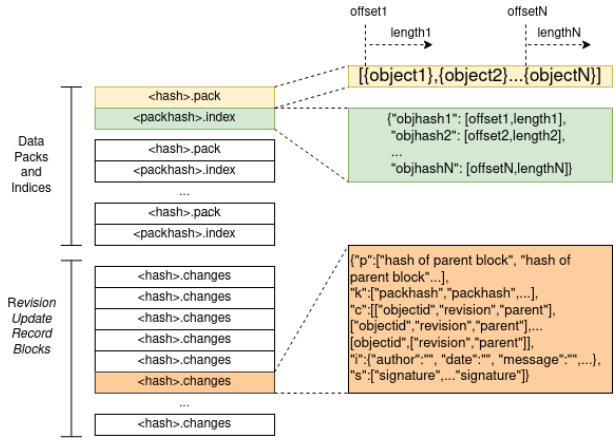

Figure 9: The Document Chain - Data Structures: : Revision Update Record Blocks are denoted using the *.changes* suffix, Data Packs use the *.pack* suffix and Data Pack Indices use the *.index* suffix; the *[offset,length]* tuples inside indices refer to a byte range within the corresponding pack.

state. The read function produces a JSON document starting from a document chain. Replicate and meld are two replication strategies implemented by the framework: the details of these two functions are provided in Section 4.3.
The Document Chain framework provides both a simple API, based which uses the standard C++ library, as well as an extended Qt API. In the following the Qt API will be presented.

4.2 Adapters

In order to provide a flexible way of storing data on different backends, the low-level task of reading or writing the elements of the chain is fulfilled by pluggable adapters, which can interface with several backends. Several adapters have been implemented to save data on the filesystem (which stores blocks and packs as files), on cloud sharing platforms (such as Dropbox and Nextcloud), on a relational database (for example MariaDB or SQLite), and in-memory (for temporary data).
Adapters implement a state machine in order to support asynchronous operation. The states of an adapter are illustrated in Figure 10. The initial state, **UNINITIALIZED** signals that the adapter is not ready to be used, whereas the **INIT** state is used to indicate that the adapter has been initialized but still need to be synchronized with the backend storage. When the adapter is in the **READY** state the Document Chain can be read or updated. On the contrary, if the state is **NEED_PULL** or **NEED_PUSH**, data must be transferred from or to a remote location. Transfers are signaled by transitioning to the **BUSY_PULL** or **BUSY_PUSH** states.
Adapters that access data on remote locations might need to maintain a local cache. Accordingly, the adapter transitions to different states in order to signal the need for pulling or pushing data from and to a remote site. Each adapter need to implement a specific API comprised of the following methods:

- **init**: used to initialize the adapter;
- **update**: used to update the state of the adapter and synchoronize it with the storage backend;
- **push**: used to initiate or resume a push operation;
- **pull**: should start or resume a pull operation;

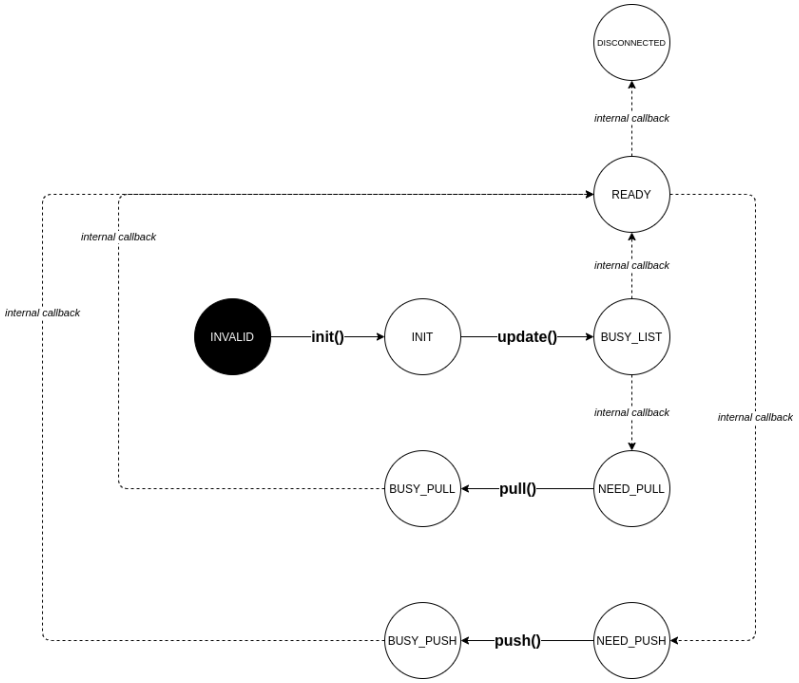

Figure 10: The Document Chain - Adapter States.

- **list_objects**: lists all objects of a specific type;
- **read_object**: performs a partial or complete read of an object;
- **write_object**: writes an object;
- **free_char_ptr**: releases the memory allocated during a read operation;
- **prefetch**: prefetches an object (for example, from a remote storage);
- **begin_chain_update**: signals the beginning of an update operation;
- **end_chain_update**: signals the end of an update operation (used for flushing the backend storage when deferred writes occour).

Different types of adapters have been implemented to fulfill the requirements of the project, namely a filesystem adapter, a SQLite adapter, a Dropbox adapter and a WebSocket adapter.

4.2.1 The QAdapter class

Each adapter must implement the aforementioned methods by subclassing the abstract class *QAdapter*. This class derives from *docchain::Adapter* and provides Qt specific features, such as a signal to notify for state changes.

- `void set_state(docchain::AdapterState state, const QString &message = {})` (*protected*): this method is used by derived classes to change the state of the adapter: when the state is changed a corresponding *stateChanged* signal is emitted.

- `void stateChanged(docchain::AdapterState state, docchain::AdapterState previous, const QString& message)` (*signal*): this signal is emitted when the state of the adapter changes. Both the new state and the previous one are provided, as well as a string message.

4.2.2 Filesystem Adapter

The Filesystem Adapter (Figure 11) stores all data items as files inside a directory. The contents of the directory can be subsequently synchronized using a suitable protocol (such as *rsync*). A strict writing order is enforced by the adapter, as to store data packs and indices before delta blocks: this prevents situations where a block cannot be fully processed because the relevant, dependent data is not yet available.

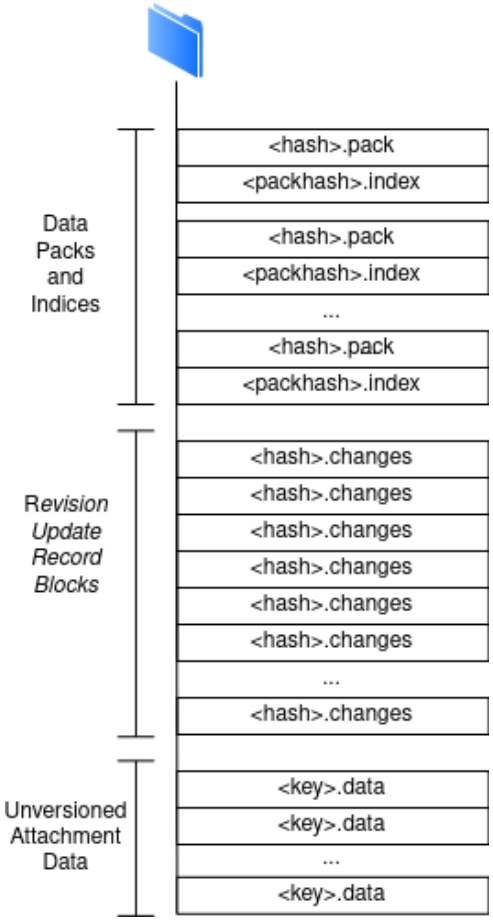

Figure 11: The Document Chain - Filesystem Adapter

4.2.3 SQLite Adapter

The SQLite adapter is built upon the *SQL* module from the Qt Framework. It supports writing on a single database file where data items are divided into different tables (Figure 12). Each table contains pairs of keys (hash values) and string data. Data is written to the database using transactions,

which enclose each commit operation. This adapter could be easily modified to support other types of SQL databases, as supported by Qt.

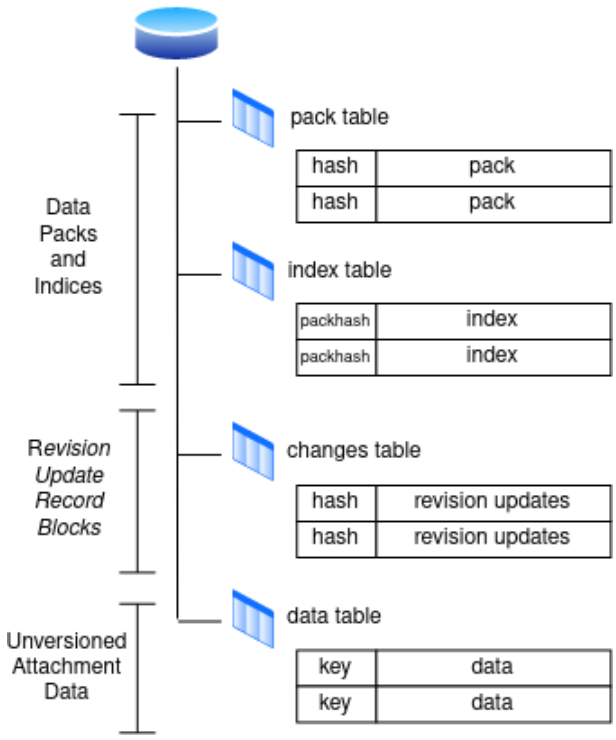

Figure 12: The Document Chain - SQLite Adapter

4.2.4 Memory Adapter

The memory adapter (Figure 13) stores its data by means of an associative array. This adapter is useful for temporary data and can be employed to store local replicas.

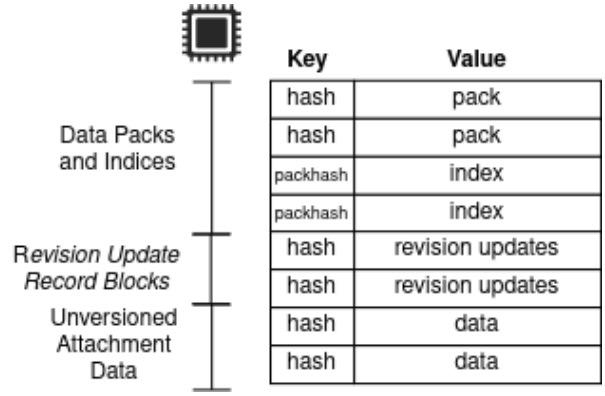

Figure 13: The Document Chain - Memory Adapter

4.2.5 WebSocket Adapter

The WebSocket adapter (Figure 14) is comprised of two parts: the adapter itself (client) and a server component. The server makes use of a persistent adapter to provide a storage backend which is accessible from a client. The adapter connects to a server in order to read and write data.

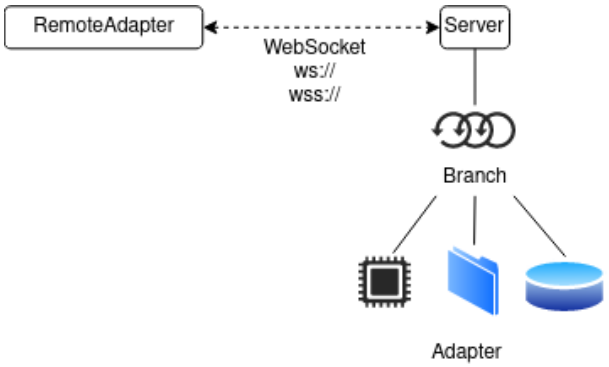

Figure 14: The Document Chain - WebSocket Adapter

4.2.6 Dropbox Adapter

The Dropbox adapter enables users to store data in a shared folder. The adapter requires authentication, therefore an additional state, called *NEED_AUTH*, was added. As soon as the Dropbox adapter is initialized, the authentication state is checked, with two possible outcomes: If the application is not authenticated, the adapter goes in the *NEED_AUTH* state, until the user authenticates itself; otherwise, if the application is already authenticated, the update method is called.
The update method reads the files list stored on Dropbox: if no files are present, the adapter goes in the *READY* state, whereas if files need to be fetched from the server, the adapter goes in the *NEED_PULL* state. A client application can react to this change and start to execute a pull. When the local state is in sync with the remote one the adapter transitions to the *READY* state. This workflow is illustrated in Figure 15.

Authentication In order to work, the Dropbox adapter needs an authentication token from Dropbox. The token can be passed directly to the adapter when creating it, or can be obtained by an Oauth authenticator, which handles the authentication procedure. Once the authenticator gets the token, it passes it to the adapter. The Oauth authentication is performed using the Qt Network Authorization library.

4.2.7 Detailed description

The API provides several types to deal with the Document Chain itself and with adapters. The *QDocumentChain* class implements the methods to interact with the data structures, whereas classes deriving from *QAdapter* implement storage backends. In this section all the methods implemented by the Qt version of the Document Chain will be presented.

- `explicit QDocumentChain(std::shared_ptr adapter, const QJsonObject& config = {}, const QString& block = {})`: a DocumentChain object can be constructed by providing a suitable storage adapter. An optional configuration JSON can be provided: this file can

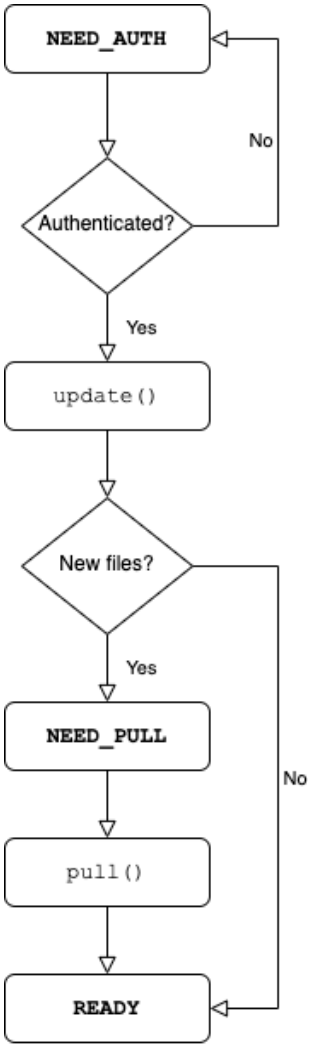

Figure 15: The Document Chain - Dropbox Adapter States and Workflow (Pull)

be used to pass additional parameters to the underlying *libstooldb* library. Moreover, a *block* identifier can be used to process the chain up until a specific commit.

- `bool touch() const`: the touch method forces a read for every referenced object on the backend adapter. This can be used for prefetching data from a remote storage (if local caching is supported by the adapter).
- `void update(const char *json)`: this method is used to update the Document Chain with the given JSON data. The input will be processed by the *flattening* procedure, resulting in a new state. Changes will not be written to the storage backend until an explicit *commit* is performed.
- `void update(const QJsonDocument &json)`: same as the previous method, but this version accepts a QJsonDocument instead of a string.
- `QJsonObject meld_to(QDocumentChain &target, bool include_caches = false, bool full_blend = false) const`: this method is used to *meld* the chain into another (a detailed

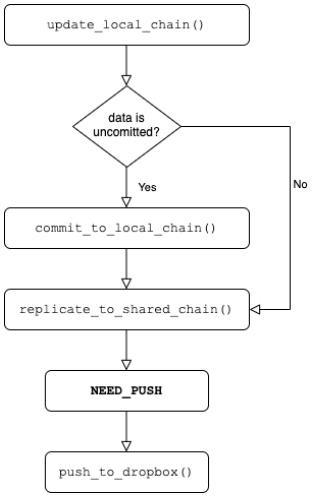

Figure 16: The Document Chain - Dropbox Adapter States and Workflow (Push)

description of the melding process will be provided in Section 4.3). Melding can include cache data if indicated by the corresponding parameter. Furthermore, it can either meld only loaded blocks or all blocks (by reloading the state of the backend adapter), regardless of the currently loaded chain (if *full_blend* is set to *true* all blocks will be considered).

- `QJsonObject read() const`: transforms (*unflattens*) the data in the chain and returns the corresponding JSON document.
- `QStringList anchors() const`: returns a list of all anchor blocks.
- `QString viewpoint() const`: returns the current target anchor block, or an empty string if multiple anchors are present. If the document chain was instanced with a specific commit point, this method returns the corresponding block identifier.
- `QString cache() const`: requests the creation of a cache of the current version of the chain. This cache speeds up the access to the document chain because previous blocks are not needed anymore to access the data at that commit point.
- `QStringList caches() const`: returns a list of all cached commit points (blocks).
- `QJsonObject changes(const QString &block = {}) const`: returns a description of all the changes that were made in the specific block. If no block identifier is specified, the changes made in the current viewpoint/last commit is provided.
- `QJsonObject details(const QString &block) const`: returns a detailed description of all the changes that were made in the specific block.
- `QJsonObject revision(const QString &revision) const`: returns the data associated with the given revision.
- `QString create_object(const QString &id, const QJsonObject &o)`: creates an object with the specified contents. This method returns the first revision string.
- `QString update_object(const QString &id, const QJsonObject &o)`: updates an existing object with the specified contents. This method returns the resulting revision string.

- `QString delete_object(const QString &id)`: deletes an existing object and returns the resulting revision string.
- `QJsonObject value(const QString &id) const`: returns the current value of an existing object, which corresponds to its winning revision.
- `QJsonObject conflicting() const`: returns an overview of objects which have conflicting revisions.
- `void resolve_conflict(const QString &id = {}, const QString &revision = {})`: resolves a conflict by choosing the winning revision. If no identifier is provided, all conflicts in the chain will be resolved by choosing the currently winning revision. If a winning revision is specified (and the identifier is not empty), the winning revision will be overridden.
- `QStringList all_docs() const`: returns a list of all objects' identifiers in the chain.
- `QString root() const`: returns the identifier of the root object.
- `QString winning_revision(const QString &id) const`: returns the winning revision of the specified object.
- `QString first_revision(const QString &id) const`: returns the first revision of the specified object.
- `QString parent_revision(const QString &id, const QString &rev) const`: returns the parent revision of the specified revision of an object.
- `QStringList history_of(const QString &id, const QString &revision = {}) const`: returns the history of all revisions up to the specified one. If no revision is specified, the currently winning revision is considered.
- `QString commit(const QString &author, const QString &description, unsigned int pack_size_limit = 0)`: commits data and changes to the backend: this will create a new block and possibly a new pack (if new content is produced). The author of the commit as well as a description message can be specified. Furthermore, it is possible to limit the size of the produced data packs by specifying a soft limit using the corresponding parameter.
- `void reload(const QString &block = {}) const`: reloads data from the backend storage.
- `void store(const QString &key, const QBuffer &data)`: stores arbitrary data on the backend. This data is not tracked by the document chain, but can be used for attachments.
- `QStringList data() const`: retrieves all arbitrary data keys.
- `void load(const QString &key, QBuffer &data) const`: fetches arbitrary data into a buffer.
- `std::shared_ptr<QDocumentChain> branch(std::shared_ptr<docchain::Adapter> adapter)`: branches a new chain which will use a different adapter. Branching will invoke a *meld* to replicate the current state into the new chain.

- `std::shared_ptr<const DocumentChain> branch_readonly(std::shared_ptr<Adapter> adapter, const std::string &block = {}) const`: branches a new read-only chain which will use a different adapter. Branching will invoke a *meld* to replicate the current state into the new chain. It is possible to retrieve the state of the chain at a specific commit point.
- `std::shared_ptr<const QDocumentChain> branch_view(const QString &block = {}) const`: branches a new read-only chain. It is possible to retrieve the state of the chain at a specific commit point.
- `std::shared_ptr<QDocumentChain> spin_off(std::shared_ptr<docchain::Adapter> adapter)`: branches from the current document chain state without keeping the chain history of the source.
- `QStringList created_since(const DocumentChain& other) const`: returns a list of the identifiers of the object that have been created (and are therefore not present in the *other* chain).
- `QStringList updated_since(const DocumentChain& other) const`: returns a list of the identifiers of the object that have been modified since the *other* chain.
- `QStringList deleted_since(const DocumentChain& other) const`: returns a list of the identifiers of the object that have been deleted since the *other* chain.
- `bool uncommitted() const`: returns true if the document chain contains uncommitted changes.
- `QJsonArray staged() const`: returns a description of all uncommitted changes.
- `QJsonObject stage() const`: returns a snapshot of all uncommitted changes.
- `void replay_stage(const QJsonObject &snapshot, bool overwrite)`: replays a snapshot of all uncommitted changes. If *overwrite* is set to true, the snapshot will overwrite the existing state.
- `QJsonArray state() const`: returns a list of loaded blocks.
- `QJsonArray safe_state() const`: returns a list of loaded blocks that have all referenced packs available.
- `void mold(const QJsonArray &chainstate)`: filters out all blocks but the one included in the *chainstate* array.
- `void reset(const QString &block = {}) const)`: discards all uncommitted changes and restores the last committed state of the chain. If a block is specified the specific commit point is retrieved.
- `void discard())`: discards all uncommitted changes.
- `bool contains(const QString &id) const`: check if an object exists in the chain.
- `QStringList chain(const QString &block = {}) const`: returns list of blocks (history) up to the specified one.
- `QJsonObject check_compatibility(const DocumentChain &target) const`: checks whether the chains are compatible (for a replication or meld).

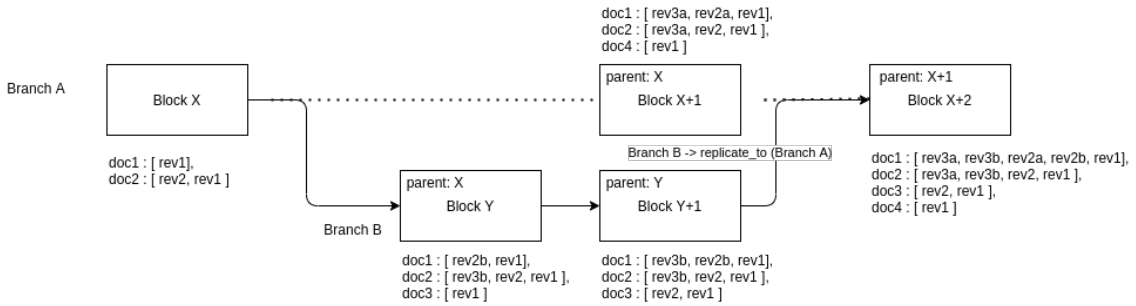

Figure 17: The Document Chain - Replication

- `void undiff()`: replaces (by updating) differential orderings with full orderings.
- `bool check_missing_data() const`: check whether all pack dependencies are met.
- `QJsonObject check() const`: check the consistency of the chain.
- `static QString sync_lib_version() noexcept`: returns the version string of the underlying library.
- `std::shared_ptr<docchain::Adapter> adapter() const`: returns the adapter used by the chain.
- `QJsonObject config() const`: returns the configuration used by the chain.

4.3 Replication vs Melding

The Document Chain API provides two different ways of synchronizing changes made on one replica, namely replication and melding. Replication is performed at the revision tree level: the changes made to revision trees observed on a source chain are copied to a target chain. Those changes appear as an update on the chain, thus they need to be committed. On the other hand, melding is performed at the block level: blocks, packs and indices found on the source chain that are missing on the remote chain are copied. The resulting data will be the same in both cases, however only melding preserves the commit history and information from the source chain.
As an example, consider the chains shown in Figure 17 and 18. A chain consisting of a block *X* initially contains two objects *doc1* and *doc2*. The chain is subsequently forked, so as to create two replicas which are independently updated. New objects are created on both branches of the chain and finally those chains are brought togheter either by replicating the contents (Figure 17) or by melding them (Figure 18). With replication, the newly committed block (*X+2*) only references *X+1* as a parent: all changes from the *B* branch are simply integrated into the other branch. On the contrary, with melding block *X+2* references both *X+1* and *Y+1*, meaning that the origin of all the changes (i.e. the corresponding commits) is preserved.

4.3.1 Collaborative application design

Using the Document Chain a single-user application can be seamlessly converted into a collaborative application by implementing some additional operations to serialize and deserialize the existing data model into a JSON document. As shown in Figure 19, it is possible to employ different Document Chains (in the example, a *shared* one and *local* ones) to exchange modifications made by

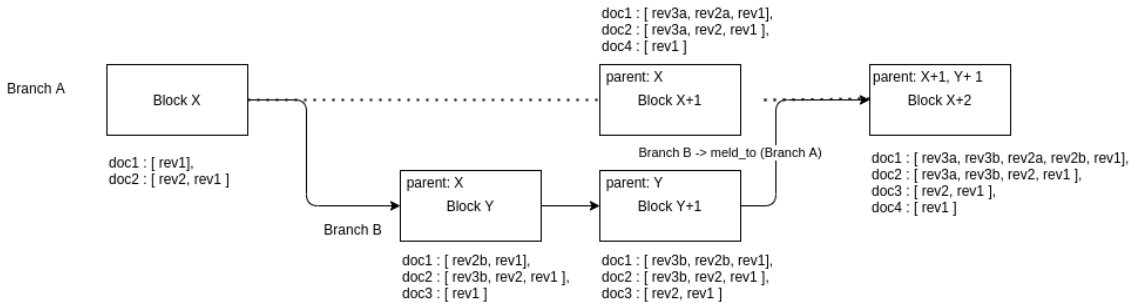

Figure 18: The Document Chain - Melding

different users. The serialized data from the model is compared against the local chain in order to determine the changeset which is subsequently committed to the local chain. The *push changes* operation merges changes from the local Document Chain into the shared one, whereas the *pull changes* operation merges changes from the shared chain into the local one. Finally, the local chain can be read and deserialized in order to obtain an updated model.
As replication does not enforce any specific communication channel, a particularly cost-effective way to exchange updates between multiple clients/participants is through cloud-based file sharing platforms. As discussed in the introduction, the use of such infrastructure can reduce the overall maintenance costs for the developer while ensuring that data is stored according to the end-user requirements. Since Document Chains are based on immutable elements (which are identified by the hash value of their contents) it is easy to implement a caching mechanism to reduce network traffic in remote replication.

5 Further improvements

The design of the document chain, as presented in the previous section, leaves room for further improvements. A significant reduction to the storage space can be achieved by storing only differential updates to ordering documents (which would normally store the full sequence of document identifiers). Update record blocks could also be digitally signed to add accountability and non-repudiability.

6 Evaluation

To evaluate the proposed approach we conducted several experiments targeted at measuring the space overhead and the overall performance. In this section we discuss a synthetic benchmark which simulates a collaborative editing session of a shared text document.
We first present an evaluation discussed in [9], with an experiment derived from [10], in order to determine the storage overhead, the changeset size, the maximum resident set size, and the time required to reconstruct the full state from all changesets. We compare the obtained results with *automerge* [5], which was chosen because it is a well-known CRDT which natively supports JSON data and exhibits some commonality with our solution. We created a setup which simulates subsequent edits to a JSON document which contains an array of financial transactions: the first version (denoted as V_1) is listed in Listing 1. The root object contains a data field, which corresponds to a sub-object containing an array of transactions. Each transaction is defined by an identifier (the "_id" field), a string representing a currency, a numerical value, and two strings which stand for the sender

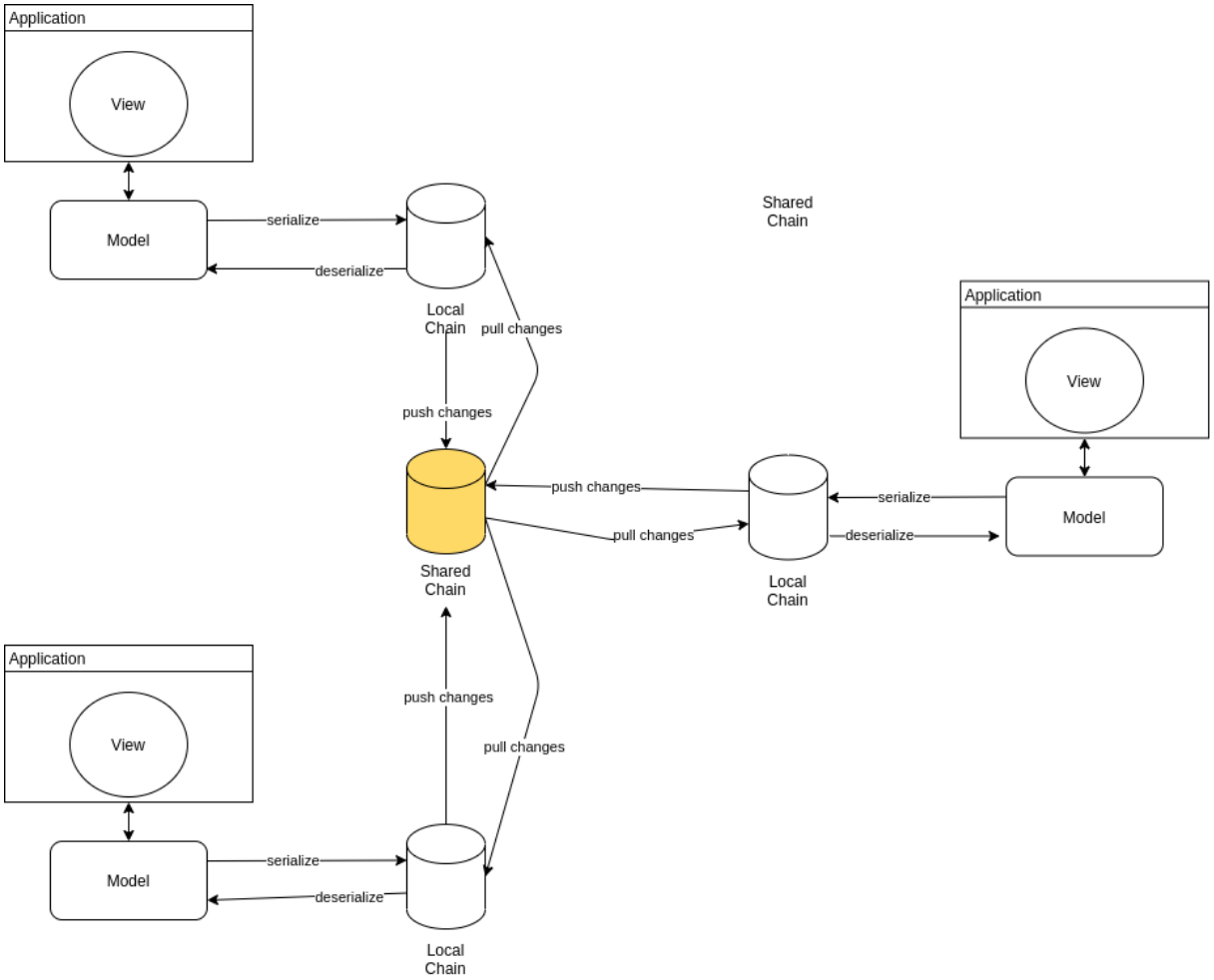

Figure 19: The Document Chain - Collaborative Application Architecture

and recipient accounts. Furthermore, the root object also contains a filed named "info" which maps to an object with a transaction counter (mapped to the "txcount" field).

```
{"data": { "transactions":[ {" _id ":"391...32",
         "currency": "EUR",
         "value": 22412,
         "from": "13465 -45566",
         "to": "34655 -67554"
        }]}, "info": {"txcount ": 1}}
```

Listing 1: Sample JSON document

The evaluation is comprised of 1000 steps, each comprising several edits to the data structure. More specifically, document version V_N is modified to produce a new version (denoted as V_{N+1}) by performing the following changes: first, a new object is added to the *transactions* array, and second, the contents value of the *currency* field an existing object are replaced with new data (specifically the "EUR" string). Moreover the value of the field *txcount* is updated to reflect the size (number of elements) of the *transactions* array. With *automerge*, new versions of the document are generated by passing the required update operations to the *change* method (since it is an operation-based CRDT): each changeset in then saved to a separate file; on the contrary, with the Document Chain the new version of the whole document is processed by the update logic, which will automatically devise the corresponding changeset. Moreover, each step entails a commit of the chain, which relies on the filesystem adapter to produce three files for each step (a revision update record block, a data pack and the corresponding index). To evaluate the impact of the hashing function used to generate revision strings inside the chain, we consider both SHA256 and xxHash. To keep the evaluation scenario as simple as possible, digital signatures are omitted. All tests are performed on an AMD Phenom^{TM} II X4 965 processor with 16 GiB of RAM running Ubuntu 20.10. For *automerge*, version 0.14.2 running on Node.js version 12.18.2 is used.

6.1 Storage overhead

The first measurement concerns the overhead due to the additional information required for maintaining a history of all updates made to the data structure. Concerning the Document Chain we consider the total size of the data, which comprises pack files, the corresponding indices and the revision update blocks; for *automerge* we compute the cumulative size of all changesets obtained using the *getChanges* method (which returns an array of operations to be applied to version V_N in order to obtain V_{N+1}).
As shown in Figure 20 the Document Chain approach results in a comparable overhead to *automerge*, nonetheless depending on the hashing algorithm used for generating revision strings, a slight difference is observable. It should be noted that *automerge* records changes made to single fields whereas the Document Chain stores full JSON objects for each revision: the former might therefore produce better results for small changes in large objects, whereas the latter ensures the inner consistency of each object. The size of the input document is reported as *Full state*: since both CRDTs record and enable access to the whole history of the document, we report both the size of a single version of as single full state as well as the cumulative size of all previous states. In this regard, both CRDTs allow for maintaining the full history of a document with considerably less storage overhead.

6.2 Size of the changeset

Changesets group a series of updates that need to be applied to version V_N in order to obtain version V_{N+1}. In a distributed scenario the size of a changeset determines the amount of data that needs to

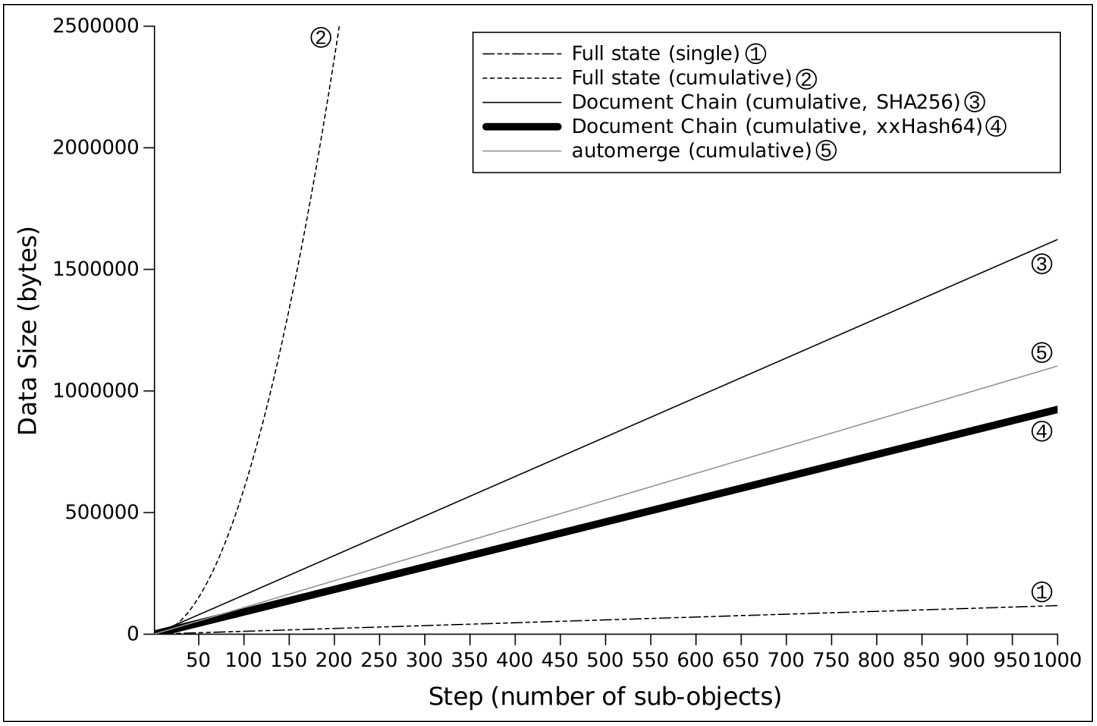

Figure 20: Storage Overhead

be exchanged between participants in order to update their current state.
As can be observed in Figure 21, both CRDTs provide an efficient way of replicating changes, with a resulting cost which is orders of magnitude less than transmitting the full state at each update.

6.3 Maximum resident set size (RSS) in full state reconstruction

Manipulating large JSON data structures can be expensive in terms of memory, in particular on resource constrained devices. Accordingly we measure the Maximum resident set size (RSS) during each experiment to evaluate the memory used by the CRDTs while reconstructing the full state based on the changesets available at each step. For the Document Chain, measurements have been obtained using the *time* command, whereas for *automerge* we employ *process.memoryUsage().rss*. As shown in Figure 22, *automerge* is penalized by the fact of being a Javascript library running on Node.js, whereas the Document Chain is a native library written in C/C++.

6.4 Full state reconstruction time

Reconstructing the full state from the available changeset is also a time consuming. For the Document Chain, we evaluate the time required by this operation using the *time* command, whereas for *automerge* we employ *Date.now()* to obtain the current time before and after the process (in order to avoid taking into consideration the initialization time of the Node.js runtime).
As shown in Figure 23, *automerge* takes considerably more time compared to both Document Chain scenarios, despite the fact that the former has to read one-third of the files compared to the latter.

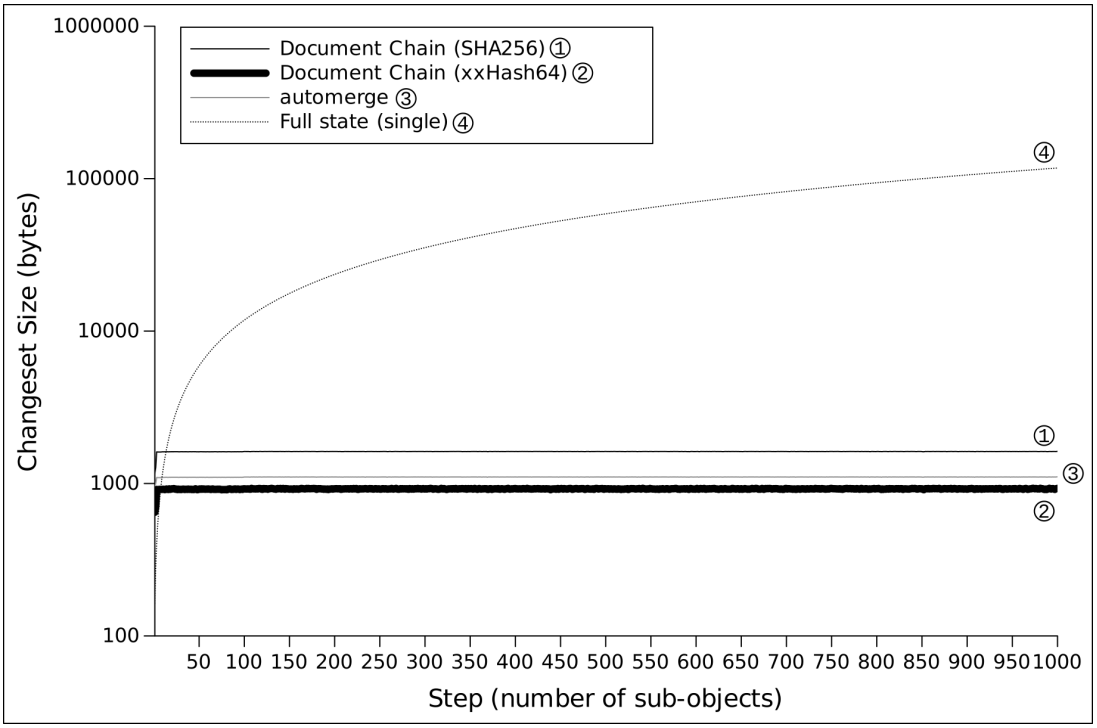

Figure 21: Size of the changeset

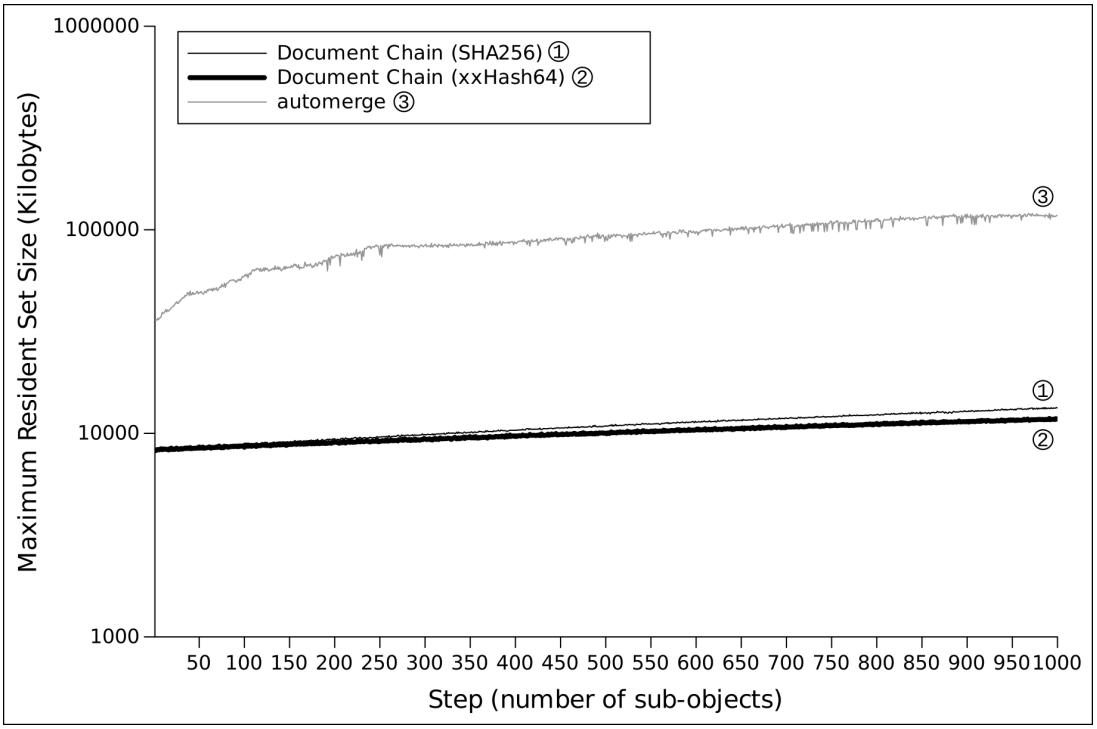

Figure 22: Maximum resident set size (RSS)

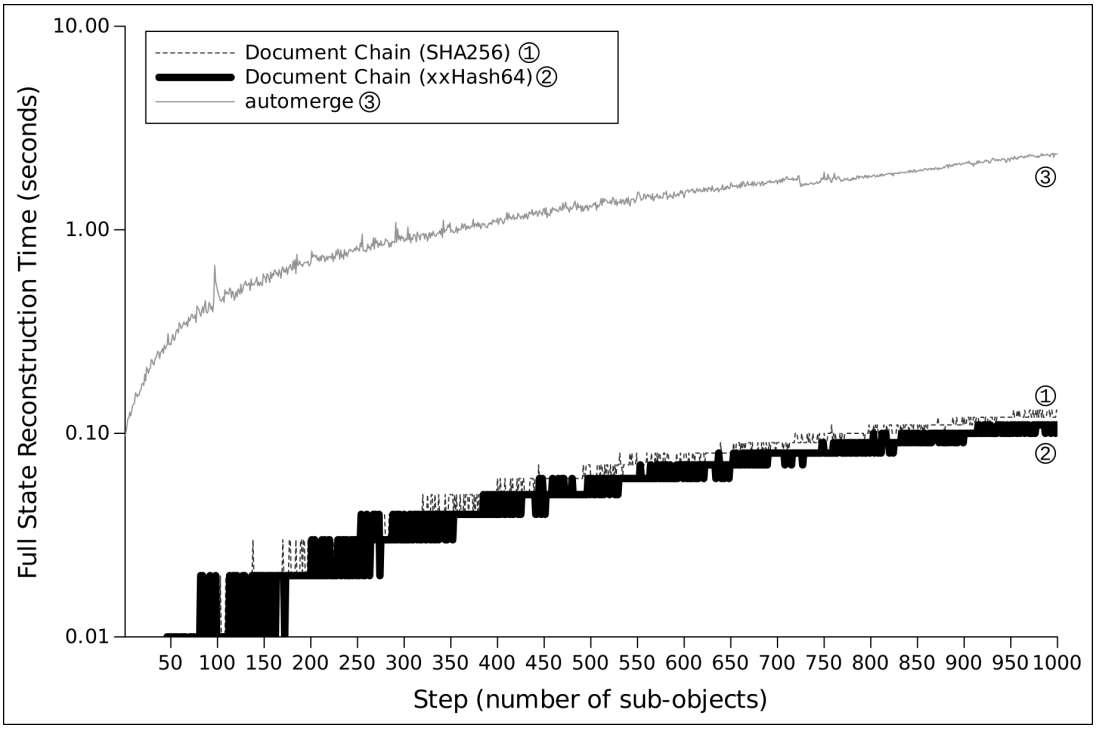

Figure 23: Full state reconstruction time

7 Conclusion

In this technical report, we presented the concept of Document Chain, which implements a JSON-based CRDT suitable for building collaborative applications or integrating collaboration features into existing applications. The proposed approach does not require synchronous communication between participants and can therefore exploit existing cloud-based file sharing platforms to exchange updates, such as Dropbox. Each partipant can modify its own replica of the document independently; since the replication process is based on multi-version concurrency control (MVCC) the Document Chain ensures non-destructive conflict management. In comparison with similar solutions such as Automerge, the Document Chain implements a simpler programming interface which does not require explicit updates to the documents in order to record modifications made to the document.

References

[1] Mark Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. Lecture Notes in Computer Science - LNCS. 6976. 386-400, 2011

[2] Mihai Letia, Nuno Preguiça, and Marc Shapiro. Consistency without concurrency control in large, dynamic systems. SIGOPS Oper. Syst. Rev. 44, 2 (April 2010), 29–34, 2010

[3] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making Operation-Based CRDTs Operation-Based. AIS 2014: Distributed Applications and Interoperable Systems pp 126-140, 2014

[4] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based CRDTs by delta-mutation. In International Conference on Networked Systems, NETYS 2015, pages 62–76. Springer LNCS volume 9466, May 2015.

[5] Martin Kleppmann and Alastair R Beresford. A conflict-free replicated JSON datatype. IEEE Transactions on Parallel and Distributed Systems, 28(10):2733–2746, April 2017.

[6] Pascal Grosch, Roman Krafft, Marcel Wölki, and Annette Bieniusa. AutoCouch: A JSON CRDT framework. In 7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2020. ACM, April 2020.

[7] CouchDB Team, CouchDB 2.0 Reference Manual, Samurai Media Limited. 2015.

[8] PouchDB, https://pouchdb.com/

[9] Amos Brocco, (2021). The Document Chain: a Delta CRDT framework for arbitrary JSON dataThe Document Chain: a Delta CRDT framework for arbitrary JSON data SEBD.

[10] Amos Brocco, Patrick Ceppi, and Lorenzo Sinigaglia, (2020). libJoTS: JSON That Syncs! SEBD.

[11] Solid Technical Reports, 2021-02-07, https://solidproject.org/TR/

[12] Shapiro, M., Preguica, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Rapp. Rech. 7506, InstitutNational de la Recherche en Informatique et Automatique (INRIA), Rocquencourt,France. 2011

[13] Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. Convergent and Commutative Replicated Data Types. Bulletin- European Association for Theoretical Computer Science, European Association for Theoretical Computer Science; 2011, pp.67-88.

[14] Almeida, Paulo Sérgio, Ali Shoker, and Carlos Baquero. "Delta state replicated data types." Journal of Parallel and Distributed Computing 111 (2018): 162-173.