Melda: A General Purpose Delta State JSON CRDT

Amos Brocco
University of Applied Sciences and Arts of Southern Switzerland
Lugano, Switzerland
amos.brocco@supsi.ch

Abstract

In this paper we present a delta state conflict-free replicated data type for arbitrary JSON documents called Melda, which aims at enabling the implementation of offline-first asynchronous collaboration into applications. The proposed framework does not rely on a coordination service, and supports different types of decentralized storage solutions to tackle the problem of ensuring security, privacy and data portability in the context of collaborative document editing applications. We present our solution both through a formal description of the replicated data type and through some implementation details; moreover we provide an evaluation of the algorithmic complexity, and by means of a synthetic benchmark we analyze the metadata overhead, the actual performance, and the scalability of our approach.

ACM Reference Format:
Amos Brocco. 2022. Melda: A General Purpose Delta State JSON CRDT. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Collaborative document editing applications enable their users to work jointly on a common task by sharing and coordinating concurrent modifications. Collaboration can happen either in real-time (simultaneous editing) or asynchronously (modifications can be independently made offline and each resulting version is later merged into each other). We consider the latter approach of particular interest, because it is suitable for creating offline-first (or local-first) applications [8] that can operate even without network connectivity. The development and deployement of such software requires nonetheless both a communication technology and a data synchronization mechanism for exchanging and integrating modifications. The majority of the existing solutions rely on centralized services in the cloud, which implement not only the necessary coordination logic but also process and store user's data, thus creating single points of failure and rising safety and privacy concerns. In particular, collaborative applications typically force the user to surrender private data to a third party, trust its actions, and understand and accept the risks involved. To overcome these issues, in recent years there has been a constant push toward the development of decentralized solutions which aim at ensuring security, privacy and data portability. In the context of collaborative document editing, we ought to develop a solution based on serverless coordination and decentralized storage. To get rid of coordination services, we can turn our attention to conflict-free replicated data types. Conflict-free replicated data types (CRDTs) [9, 14] are data structures tailored for distributed computing which are meant to achieve strong eventual consistency. CRDTs allow each replica to be independently and concurrently updated, without the need for explicit coordination. By exchanging update information in the form of operation sequences [2, 7], serialized states [14] or delta updates [1], the logic behind CRDTs ensures that each replica eventually converges to a common state [14]. The development of collaborative applications is a primary use-case for such data structures. CRDTs also support the vision of a fully decentralized web, allowing people and applications to work together on their data by exploiting different types of communication channels as they see fit, and by leveraging private data storage facilities.

In this paper we present Melda [1], an open source JSON CRDT framework for the implementation of collaboration features into applications. Melda supports different data storage and exchange backends, which allows for combinining the advantages of CRDTs with the flexibility and safety of decentralized storage and peer-to-peer communication (both asynchronous and synchronous). This work extends [4] and [3] with improvements to the data structure, a formal description of the CRDT itself, a complexity analysis, and additional evaluation and comparison.

2 Related work

There exist different types of conflict-free replicated data types (CRDTs), ranging from simple types such as counters, registers, and sets [14], to more complex data structures [7, 11]. As explained in [12], CRDTs imply two fundamental properties: first, any replica should be modifiable with neither centralized nor distributed coordination, second, when

[1] https://github.com/slashdotted/libmelda

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

any two replicas receive the same set of updates they must reach (converge to) the same state (which leads to strong eventual consistency). CRDTs can be grouped into three different categories, namely operation-based, state-based and delta state based. Operation-based solutions [2] rely on update operations which are propagated to all replicas, and are best suited for high-frequency updates, such as in the context of real-time collaborative text editors. In the literature it is possible to find examples of operation-based CRDTs which support JSON-like data, for instance Yjs [11] or the *Automerge* [7] library. In particular, Automerge is aimed at building collaborative applications in JavaScript, and beside JSON it provides support for other CRDT types, such as counters or text buffers. As pointed out in [1], operation-based CRDTs have several advantages in the realm of real-time collaborative applications, as they can allow for simpler implementations and smaller replica update messages; however, they rely on reliable exactly-once causal broadcast of these updates [2]. On the other hand, state-based CRDTs [14] always store the full state of the data, and are therefore better suited for situations where updates are less frequent, communication is unreliable, or operations are not commutative. With state-based replication it is also easier to verify the correctness of the data in a particular point in time. The main drawback of state-based CRDTs is that the size of the state can become very large [1], consequently delta state solutions (referred to as δ-CRDTs) have been proposed [1, 13]. δ-CRDTs rely on disseminating small updates (changesets) called delta mutations: these updates are idempotent, which means that they can be applied possibly several times to an existing state without compromising its consistency, and can be disseminated over unreliable communication channels.

Recognizing the merits of these approaches with respect to asynchronous collaboration tools, the CRDT presented in this paper relies on delta states. In contrast to other approaches, Melda is not limited to linear sequences of elements (such as [5]) or simple data structures (as in [14]), but supports arbitrary JSON documents with complex hierarchies.

3 Overview of the data structure

The purpose of Melda is to achieve eventual consistency by letting each user work on its own replica while offline, and exchange the necessary information to update other replicas whenever possible. In the end, we guarantee that the state seen by all users will eventually converge, provided that all modifications generated in the system reach every replica. To prove our point, we will present the key elements of our data structure, and demonstrate convergence by framing them into a formal definition of both a state-based and a delta state CRDT.

3.1 Definitions

We consider a grow-only collection C of JSON objects, which is replicated on multiple sites and concurrently updated by each participant. Objects are atomic and immutable: even partial modifications replace the full content with a new version of the object, which is appended to the existing collection. Deletions are recorded using *tombstones* (a special value which represent the final version of an object).

Object. An object o is a data structure comprising zero or more name-value pairs as defined in RFC 8259 [2]. Names must be character strings, and values can be any valid JSON data type. In the considered framework, each object must be uniquely identified by a string value id_o. The value id_o is independent from the content of the object, therefore it is possible to update an object to a new version o' without changing its identifier, i.e $id_o = id_{o'}$.

Content digest. To efficiently compare different versions of an object, the content x is hashed to produce a string digest $H(x)$. The hashing algorithm is implementation-specific, but we assume that the digest can be represented as a string using a binary-to-text encoding such as hexadecimal or $Base64$ encodings. The identifier of the object, referred to as id_x, is omitted from the hash computation, therefore two versions x_m and y_n of two objects x and y with the same content can have the same digest, i.e. $H(x_m) \equiv H(y_n)$.

Object map. On each replica, the set O of tuples $\langle H(x), x\rangle$ stores the content of each version of each object. The set O is referred to as the *object map*, and is used to retrieve the value associated with a specific digest.

Revision string. Let x_N represent the content of the N-th version of an object identified by id_x. The *revision string* r_N associated with x_N is defined as N-$H(x_N)_Tail_N$, where N is a monotonically increasing numerical index (starting at 1), $Tail_0 = \emptyset$, $Tail_N = T(H(r_{N-1}))$, and T is a determistic function (such as the identity function, a hashing function or a simple string transformation). A revision string r_k univocally refers to a specific version x_k in the history of an object, and allows for retrieving the corresponding content from the *object map* using the embedded digest string $H(x_k)$.

Revision trees. Modifications made to an object can be recorded as sequences of *revision strings*. Due to concurrent modifications (which are expected to happen on different replicas), multiple sequences might exist for the same object. For example, while a replica might produce modifications which translate into a sequence $r_1, r_2 \ldots r_{N-1}, r_N$, changes made on another replica might result in $r_1, r_2 \ldots r_{N-1}, r'_N$ (the last revision string differs). To account for this nonlinearity, the history of modifications made to each object o across all replicas is recorded into a *revision tree* rt_o, and is stored as

[2] The JavaScript Object Notation (JSON) Data Interchange Format https://datatracker.ietf.org/doc/html/rfc8259

a collection of tuples $\langle r_N, r_{N-1}\rangle$ (r_{N-1} is referred to as the *parent* of r_N, whereas r_N is the *successor* of r_{N-1}). A revision tree for a newly created object is stored as $\langle r_1, \emptyset\rangle$.

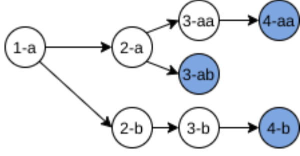

Figure 1. Revision tree and conflicting leaf revisions (denoted with a blue background).

Leaf revisions. Revisions that have no successors in the tree are called *leafs*. If a revision tree exhibits only one leaf revision, the corresponding object has no conflicts. On the contrary, concurrent modifications can lead to the existence of multiple leafs inside a revision tree. Generally speaking, if two leaf revisions share a common ancestor they are considered as *in conflict*. As illustrated in Figure 1, revisions *4-aa*, *3-ab*, and *4-b* are conflicting leaf revisions of the considered revision tree. Conflicting leaf revisions can be marked as *resolved* (using a special type of *tombstone* value), so as to ignore them in subsequent operations.

Winning revision. The leaf revision with the highest numerical index is considered the *winning revision* r_W, and determines the content that shall be returned when querying for the latest version of an object (i.e. the longest edit branch wins, as in CouchDB [6]). If multiple revisions share the same index, revision strings are compared in lexicographic sort order, and the highest one is deterministically chosen as the winner. In the example shown in Figure 1, both *4-aa* and *4-b* share the same numerical index 4, but *4-b* can still be elected as the lone winner using lexicographic comparison. From the application's point of view, the choice of the winning revision is arbitrary; however, because all revisions are kept in the CRDT, it is always possible to elect a different revision as a new winner. In the previous example, the revision tree could be extended with a new tuple $\langle 4-b, 5-ab\rangle$ to obtain a new winner referencing the same content as revision *3-ab*.

State set. Given a replica of a collection of JSON objects C, we define its *state set* (or simply *state*) $X = D \cup O$, where $D = \bigcup_{o\in C} rt_o$, and O is the *object map* as defined above.

3.2 Mapping onto a state-based CRDT

As defined in [1], 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), M is a set of mutators that update a state $X \in S$ to produce a new state $X' = m(X)$ such that $\forall m \in M, X \in S : X \sqsubseteq m(X) \wedge m(x) \in S$, and Q is a set of query functions (for reading the data).

In Melda, partial ordering can be obtained by means of the subset $\sqsubseteq$ relation. The *join* operation $\sqcup$ corresponds to the union between two sets. By its very nature, the union fulfills the requirement of being commutative, associative, and idempotent. Since $\forall A, B \in S$, $A \sqcup B \in S$, and $A \sqcup B := sup\{A, B\}$, we have a least upper bound (or *supremum*), making S a join-semilattice. Concerning mutators, we observe that any modification translates into adding one or more tuples to the state set. A single update of state $X \in S$ can thus be formalized as $X' = X \cup \{\langle r_i, r_{i-1}\rangle\}$, where $X' \in S$ is the resulting state. Mutators are therefore *inflations* [1], and the requirement $X \sqsubseteq m(X)$ holds $\forall m \in M, X \in S$.

Given the associativity, commutativity, and idempotence of the join operation the requirements for convergence (as stated in [1]) are fulfilled. Also, our CRDT is isomorphic to a G-Set (Grow-only Set [14]).

3.3 Mapping onto a delta state CRDT

Propagating the full state in order to update all replicas is an expensive operation. In this regard, δ-CRDTs exchange *deltas* (fine-grained states), which are comparatively smaller than full-states, but still ensure convergence as with state-based CRDTs. According to [1], 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. In the considered scenario, each transition from state X to state X' can be represented by an *update set* $U = X' \setminus X$, with $U \in S$, since S is closed under set difference. Update sets are equivalent to delta-mutations, since $X' = m(X) = X \sqcup m_\delta(X) = X \sqcup U$, where $X, X' \in S$ and $m_\delta(X) \in S$ is the delta mutation. Delta-mutators m_δ are defined by the relation $m_\delta(X) = U$. Furthermore, update sets also translate into delta-groups D, as the relation $X' = X \sqcup D$ holds when $D = m_\delta(X)$, and by associativity this relation is verified even when considering a join of several delta-groups. Since the join operation is associative, commutative and idempotent, the requirements for convergence (as stated in [1]) are fulfilled. It is therefore possible to introduce the following definition:

Delta state. : $\forall X, X' \in S$, $\Delta_{X',X} = X' \setminus X$ is called a *delta state*. $\Delta_{X',X} \in S$ and $X' = X \cup \Delta_{X',X}$. Any state $X \in S$ can be decomposed into an arbitrary number $N \in \mathbb{N}_{>0}$ of delta states $\Delta_i \in S, i \in \mathbb{N}_{>0}$, such that $X = \cup_N^{i=1} \Delta_i$.

4 Implementation

Melda is not intended as a replacement for the application's data model, but as a complement to the existing one. The latter might comprise hierarchies of objects whose changes are difficult to track individually. To update the CRDT, Melda processes a JSON serialization of the application's data model, decomposes it into a collection of objects, and determines changes by comparing those objects against the state set. In the following, some implementation details will be discussed.

Updating the CRDT. Using a reversible transformation algorithm derived from [5], the JSON document representing the data model is recursively *flattened* to move nested objects into an associative array C (Algorithm 1). Each moved object is replaced by a unique string reference generated by the MakeIdentifier function: this reference depends either on the value of an *_id* field, or the path of the object inside the document; all other strings are *escaped* (using an Escape function). The resulting object is marked as the *root*. Afterwards, all the objects in C are compared against the current state: changes produce new revision tuples, whereas new values are stored into the *object map*. To minimize the space taken by large arrays, a difference algorithm can be used to create *patches* against previous revisions. To reconstruct the document from the state set, references in the *root* object are recursively replaced by the value of the winning revision.

Non-destructive array merging. Several applications that deal with structured data, such as rich text editors, collection management systems or financial accounting software, are built around arrays of objects. Since Melda only updates objects as a whole, concurrent updates made to such arrays (in particular additions) might not persist, unless special attention is devoted to conflicting revisions. As an example, we consider concurrent modifications that add a new element to an array $[A, B, C]$, which we assume being part of a JSON object, where the letters A, B, and C each represent another object. On a replica R_1, a new element **D** is appended at the end of the array (leading to $[A, B, C, \mathbf{D}]$), whereas on replica R_2 a new element **E** is inserted between **A** and **B** (resulting in $[A, \mathbf{E}, B, C]$). These updates produce two conflicting revisions: upon merging, only one of those revisions will be deterministically chosen as the *winner*. Therefore, the value of the array might either be $[A, B, C, \mathbf{D}]$ or $[A, \mathbf{E}, B, C]$. To retain both additions, during the reconstruction process conflicting arrays are combined using Algorithm 2, in order to produce a *merged view*. The algorithm merges an array S into an array T. The $\text{IndexOf}(A, e)$ function returns the position of an element e inside an array A, whereas the $\text{Insert}(A, i, e)$ procedure inserts an element e into an array A at a specific position i. The first element e in the source array S which is also found in the target array T is called *pivot*: all other elements from S which are not yet in T are inserted relative to this element. In the previous example, the merging procedure with $S := [A, B, C, D]$ and $T := [A, E, B, C]$ results in $T = [A, \mathbf{E}, B, C, \mathbf{D}]$, which retains both E and D. As the array merging procedure only concerns the data returned to the client application, no persistent changes are made to the state. If conflicting arrays contain the same elements in a different order, the *merged view* retains the order from the T array. Deletions are handled by excluding objects whose winning revision is a *tombstone*.

Algorithm 1 Flattening Procedure

```
C := {}                              ▷ Associative array of extracted objects
procedure FLATTEN(value, path := [])
    switch type of value do
        case String
            return ESCAPE(value)
        case Array
            ab := []                                   ▷ Flattened array
            for each v ∈ value do
                ab ← ab ∪ FLATTEN(v, path)
            end for
            return ab
        case Object
            ob := {}                                   ▷ Flattened object
            id_ob := MAKEIDENTIFIER(value, path)
            for each [k, v] ∈ value do
                ob[k] ←FLATTEN(v,path ∪ [id_ob, k])
            end for
            C[id_ob] ← ob
            return id_ob
        default return value
end procedure
```

4.1 Delta state serialization

Changes recorded during an update are stored into a temporary in-memory delta state. Upon commit, this delta state is serialized into two JSON structures, namely a *delta block* and a *data pack*. The former stores revision tree updates, whereas the latter stores values from the object map. Both structures are immutable and uniquely identified by the hash value of their content, hence they can be cached locally to reduce transmission costs. This separation removes the space overhead for objects that do not change or objects that share the same content. Moreover, objects which are found in existing packs are excluded from the newly created packs. To cope with the possibility that a *data pack* hasn't yet been delivered to a replica, we redefine *winning revision* as the one with the highest numerical index and the highest lexicographical value *whose content is available on that replica.* Delta blocks and data packs are stored or exchanged between peers using adapters, which currently support a variety of storage and communication backends, namely main memory (for temporary data structures), local or shared filesystems (using

Algorithm 2 Array Merging Procedure

procedure MergeArrays(S, T)
  $\iota := 0$
  $\pi := 0$
  for each $e \in S$ **do**
    if $e \in T$ **then**
      $\iota \leftarrow$ IndexOf(T, e)
      break
    else
      $\pi \leftarrow pi + 1$
    end if
  end for
  $\tau := 0$
  for each $e \in S$ **do**
    if $e \in T$ **then**
      $\iota \leftarrow$ IndexOf(T, e)
    else
      if $\tau < \pi$ **then**
        Insert(T, ι, e)
        $\pi \leftarrow \tau$
      else
        $\iota \leftarrow \iota + 1$
        Insert(T, ι, e)
      end if
    end if
    $\tau \leftarrow t + 1$
  end for
end procedure

either uncompressed JSON files or DEFLATE[3] compressed JSON files), and Solid pods [15].

Delta blocks. When data is modified, revision trees are extended and new tuples are added to the state set. These changes are encoded inside records which are grouped into *delta blocks*. Two types of records are considered: *creation* records and *modification* records. A *creation* record contains two fields, $\{id_x, H(x_1)\}$, where id_x is the UUID of a newly created object x, and x_1 is the content of the first version of x: these values can be used to compute a revision string r_1^{id} and produce the first revision tuple for that object, i.e. $\langle r_1, \emptyset \rangle$. A *modification* record contains three fields $\{id_x, r_{N-1}, H(x_N)\}$, where id_x is the UUID of the modified object x, r_{N-1} is the revision upon which the modification is made, and $H(x_N)$ is the hash of the updated version of x. These values are used to compute a new revision string of the object r_N^{id} and the corresponding tuple $\langle r_N, r_{N-1} \rangle$. The application might perform multiple updates before *committing* to a block.

Data packs and indices. Object maps are stored in the form of *data packs*, which are identified by the hash value of their content. While not strictly necessary for the operation of the CRDT, to efficiently enumerate and read the objects contained in a large packs, a corresponding *index* can also be generated: indices map the digest of an object to a position inside the pack. If not present, an *index* can be rebuilt from the corresponding *data pack* at any time.

[3] https://datatracker.ietf.org/doc/html/rfc1951

4.2 API

The functionalities of the underlying CRDT are exposed through a high-level API. The most important methods are:

Reload *Delta blocks* are read (in any order) from the backend adapter and the current state (namely, revision trees) is reconstructed.

Update An input JSON document is transformed into a collection of objects (using Algorithm 1) which are then compared against the current state. Differences result in new revisions (added to the revision trees of the concerned objects) and new values (added to the object map).

Commit A new *delta state*, comprising a *delta block* and possibly a *data pack*, is pushed to the backend adapter.

Read The actual content of each object (as determined by its winning revision) is read from the relevant *data packs* and, starting from the *root* object, the original JSON document is reconstructed and returned to the application.

Meld Melding is our trivial replication logic which involves copying missing elements (*delta blocks*, *data packs* and *indices*) from a source replica to a target, resulting in the union between these two states.

Additional methods are exposed to allow access to previous revisions of an object or to override the winning revision.

5 Evaluation

We first present a complexity analysis of the algorithms involved in the reload (i.e. state reconstruction), update, read, and meld procedures. Next, a synthetic benchmark to determine the space overhead in comparison to Automerge [7] will be presented. Automerge was chosen because it is a popular and well documented CRDT framework. Finally, the scalability of Melda with respect to the number of delta states will be discussed.

5.1 Algorithmic complexity

Prior to any other operation, the state of the CRDT must be reconstructed. This operation involves reading all delta states (more specifically, delta blocks), in order to replay all the creation and modification records. As a whole, this step has an average complexity of $O(n \cdot m)$, where n is the number of delta blocks and m is the average size of such blocks. Since revision data is kept into memory, the minimum amount of the space required while working with the CRDT is proportional to the number m of objects and the average number r of revisions for each object, therefore the space complexity is $O(m \cdot r)$. It should be noted that the actual complexity also depends on the backend adapter: for simplicity, we assume

that both the space and the time complexity of the operations performed by an adapter are $O(1)$. Updates involve multiple steps: parsing the input JSON document into a collection of JSON objects and determining the corresponding edit (mainly by comparing the hash value of their content). Parsing the JSON file has a time and space complexity proportional to the size of the input file, that is $O(n)$, where n is the size of the input. Converting the document into a collection of JSON objects performs a depth-first traversal of the input structure, and has therefore a complexity $O(n)$, where n is the number of nested objects contained in the input document. Finally, processing all the n objects and computing their hash value has a time-complexity $O(n \cdot m)$, where m is the size of the object. Inserting a value in the object map, as well as retrieving the revision tree of an object are both assumed to be $O(logn)$ operations (where n is the number of objects in the data structure), whereas querying or updating revision trees of a particular object is expected to perform with a time-complexity of $O(logr)$, where r is the number of revisions for that object. If deltas are used to optimize array modifications, a difference algorithm would need to be executed. If an algorithm such as [10] is employed, for each array an additional time-complexity of $O(n \cdot d)$ must be accounted for, where n is the sum of the lengths of the considered arrays and d is the size of the minimum edit script. Conversely, a *patching* algorithm to apply an edit script would be needed when reading the data, adding a complexity proportional to the size of the edit script. Committing has a complexity proportional to the number of revision tuples and the objects that are sent to the adapter. Reading the data structure involves determining the *winning* revision for each object and subsequently reconstructing the document. We assume that access to the revision tree of an object is $O(logn)$ (where n is the total number of objects), and that determining the winning revision is $O(logr)$, where r is the number of revisions for that object. Reconstructing the original structure of the document has a time and space complexity $O(n)$, where n is the number of string references that need to be replaced by objects. Melding merges the contents (delta blocks, data packs, and indices) of two replicas R_1 and R_2. Let n be the number of items in R_1, and m the number of items in R_2. If accessing the items within each replica has a time-complexity $O(logn)$ (where n is the number of objects in the replica), determining the items which are in R_1 but not in R_2 has a time-complexity $O(n \cdot logm)$.

5.2 Synthetic benchmark

Although Melda is aimed at collaborative applications which deal with arbitrary JSON documents, simple text editing is perhaps one of the most common and well understood use cases for CRDTs. We therefore consider a benchmark based on an editing trace[4] of a large text document with 182 315 single-character insertion operations, and 77 463 single-character deletion operations. These edits result in a final text document of 104 852 ASCII characters. Melda is evaluated through a Rust program which maintains a JSON document containing an array of objects (one for each character): each character is serialized as `{"#":"ord","_id":"uuid"}`, where *ord* is the hexadecimal Unicode code of the character and *uuid* is a unique identifier (UUID v4, 32 hexadecimal digits). Delta states and packs are stored in the compressed format on the local filesystem. The benchmarking program for Automerge is based on version *1.0.1-preview.6* (which uses binary changesets): to represent our document we consider both *Automerge.Text()*, as well as an array of objects (one for each character, as with Melda). In the latter case, our goal is to determine the overhead of the CRDT with more complex JSON documents. With Automerge we record edits using the *insertAt* and *deleteAt* methods. To simulate asynchronous collaboration, the editing process is divided into batches of 10, 100, 1 000, and 10 000 single-character operations: when a batch is completed, a *changeset* is generated. With Automerge we consider the value produced by the *getChanges* function, whereas with Melda we consider *delta states* (*blocks*, *packs*, and *indices*). In both cases it is possible to reconstruct the final version of the text document by replaying all these changesets (whereas Melda takes care of reloading all the delta states when the data structure is initialized, with Automerge it is necessary to call *applyChanges* for each changeset). The cumulative size of all changesets is closely related to the overall communication overhead, as modifications are typically propagated over a network. All tests were performed on a Linux machine (Ubuntu 18.04.5 LTS) with an Intel® Xeon® Gold 6142 CPU clocked at 2.60GHz, with a hard limit of 4 cores and 8GB of memory. For Automerge we employed Node *v16.13.2*.

Batch size	Changesets (MBytes) / Creation time / Read time		
	Am.Text()	Am	Melda
10	4.1/73s/45s	11/13.5h/2h	22/2.3h/225s
100	0.72/21s/15s	5.3/12.5h/14h50m	11/12m/24s
1 000	0.24/15s/11s	4.7/12h/14h25m	9/110s/6s
10 000	0.17/19s/11s	4.3/12h/14h17m	7.5/35s/12s

Table 1. Results after 259 778 edit operations (*Am* for Automerge).*Creation time* is the time required to produce all the changesets, whereas *Read time* considers reloading and replaying the changesets and producing a text document.

Metadata overhead. As shown in Table 1, the smallest changesets were produced by *Automerge.Text()*, as it is optimized for collaborative text editing. In comparison, when storing arbitrary objects the space overhead incurred by Automerge increases significantly. Melda produces a larger overhead than Automerge, although the size is only two

[4] https://github.com/automerge/automerge-perf

times bigger than Automerge when storing an array of objects. For comparison, the cumulative size of all versions of the document with batches of 10,100,1 000 and 10 000, operations is 1.7GB, 170MB, 17MB and 1.7MB respectively: all the considered CRDTs are more efficient in almost all scenarios.

Build time and read time. The best build and read times were achieved using *Automerge.Text()*. When dealing with an array of objects, Automerge is the slowest one, requiring several hours both to create changesets, as well as to reload them in order to read the document. Melda takes more time than *Automerge.Text*() at creating changesets and reconstructing the text document, but is considerably more performant than *Automerge* at dealing with an array of objects. Surprisingly, Melda took less time to read data in the 1 000 operations scenario than in the 10 000 one.

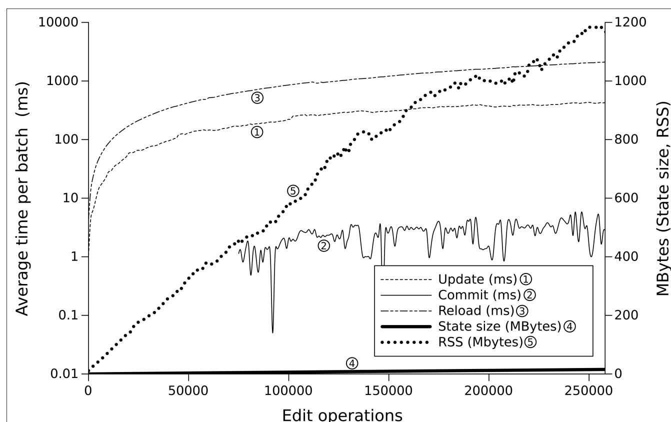

Figure 2. Scalability results (batch of 10 operations).

Scalability. To evaluate the scalability of our approach we consider the time required to commit a batch of changes, reload all the changesets after each commit, and to update the state with the next batch of edits. The total memory consumed after each commit was measured as well. The results, shown in Figure 2, correspond to the scenario with a batch size of 10 operations (with a total of 25 978 delta states). The resident set size (RSS) takes into account not only the memory used by the CRDT but also the application's data model. As expected, the commit time remains almost constant throughout the scenario (since the batch size is fixed). All other considered values grow at most linearly with respect to the number of recorded operations and the size of the state, matching our complexity analysis.

6 Conclusions

In this paper, we presented Melda, a JSON δ-CRDT solution to support the development of multi-user applications based on asynchronous collaboration. The underlying CRDT and its logic were formally presented and some implementation details were discussed. An evaluation was conducted through a synthetic benchmark. Future work includes performance and overhead optimizations, investigation into pruning mechanisms, better documentation, the implementation of additional adapters, and porting the library to other programming languages and frameworks.

Acknowledgments. This work has been financially supported by the Swiss Innovation Agency, Project nr. 42832.1 IP-ICT and by Banana.ch SA.

References

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

[2] Baquero, C., Almeida, P. S., and Shoker, A. Making Operation-Based CRDTs Operation-Based. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency* (New York, NY, USA, 2014), PaPEC '14, Association for Computing Machinery.

[3] Brocco, A. Delta-State JSON CRDT: Putting Collaboration on Solid Ground. In *Stabilization, Safety, and Security of Distributed Systems - 23rd International Symposium* (2021), C. Johnen, E. M. Schiller, and S. Schmid, Eds., vol. 13046 of *Lecture Notes in Computer Science*, Springer, pp. 474–478.

[4] Brocco, A. The Document Chain: a Delta CRDT framework for arbitrary JSON data. In *Proceedings of the 29th Italian Symposium on Advanced Database Systems, SEBD 2021, Pizzo Calabro (VV), Italy, September 5-9, 2021* (2021), S. Greco, M. Lenzerini, E. Masciari, and A. Tagarelli, Eds., vol. 2994 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 59–70.

[5] Brocco, A., Ceppi, P., and Sinigaglia, L. libJoTS: JSON That Syncs! In *Proceedings of the 28th Italian Symposium on Advanced Database Systems* (2020), M. Agosti, M. Atzori, P. Ciaccia, and L. Tanca, Eds., vol. 2646 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 116–127.

[6] CouchDB Team. CouchDB 2.0 reference manual, 2015.

[7] Kleppmann, M., and Beresford, A. R. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems 28*, 10 (2017), 2733–2746.

[8] Kleppmann, M., Wiggins, A., van Hardenberg, P., and McGranaghan, M. Local-First Software: You Own Your Data, in Spite of the Cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (New York, NY, USA, 2019), Onward! 2019, Association for Computing Machinery, p. 154–178.

[9] Letia, M., Preguiça, N., and Shapiro, M. Consistency without Concurrency Control in Large, Dynamic Systems. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 29–34.

[10] Myers, E. W. An O(ND) Difference Algorithm and Its Variations. *Algorithmica 1* (1986), 251–266.

[11] Nicolaescu, P., Jahns, K., Derntl, M., and Klamma, R. Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types, 06 2015.

[12] Preguiça, N., Baquero, C., and Shapiro, M. *Conflict-Free Replicated Data Types CRDTs*. Springer International Publishing, Cham, 2018, pp. 1–10.

[13] Rinberg, A., Solomon, T., Khazma, G., Lushi, G., Shlomo, R., and Ta-Shma, P. Array CRDTs Using Delta-Mutations. In *8th Workshop on Principles and Practice of Consistency for Distributed Data* (Apr. 2021), PaPoC 2021, ACM.

[14] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011.

[15] Solid Project. Technical Reports. Accessed Mar. 1, 2022, 2022.