Brief Announcement: Delta-State JSON CRDT: Putting Collaboration on Solid Ground

Amos Brocco

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

Abstract. In this paper we present a framework to support the implementation of offline-first asynchronous collaboration using a variety of data storage and communication backends. In particular, our approach can make use of Solid pods to exchange data between users.

Keywords: Collaborative Applications · JSON · CRDT · Solid

1 Introduction

In this paper we present a framework for the development of collaboration features into applications, by combining the advantages of CRDTs [7, 10] with the flexibility and safety of decentralized storage. As such, our intent is to exploit Solid pods [1] as a communication channel for exchanging data between users. However, thanks to the simplicity and modularity of our design, storage solutions such as shared folders or cloud file-sharing platforms, as well as physical devices like thumb drives, can also serve the same function as a pod. To ease the integration of our solution, we allow transparent replication of complex JSON documents without explicit editing. It is therefore possible to implement collaborative features into existing programs without altering their data model. The remaining of this paper presents a brief review of the relevant related work in the field of CRDTs, a formal overview of our data structure and some details of its implementation. Finally, an evaluation of the proposed solution and the corresponding results will be discussed.

2 Related work

CRDTs can be grouped into three different categories, namely operation-based, state-based and delta-state based. Operation-based solutions [3] 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 [8] or the *Automerge* [6] library, but require explicit editing of the document in order to keep track of each modification. State-based CRDTs [10] 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. The main drawback of state-based CRDTs is that the size of the state can become very large [2], consequently delta-state solutions (referred to as δ-CRDTs) have been proposed [2, 9]. 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. The CRDT discussed in this paper uses delta states and presents a practical architecture for collaborative applications with a modular design supporting different types of communication and storage backends.

3 Overview of the data structure

We consider a generic collection C of JSON objects which is generated from an arbitrary JSON document using a reversible data transformation algorithm [5]. In contrast to other solutions, which require explicit editing operations, this algorithm processes an input document (as produced by an application) to automatically extract nested objects and determine their changes. This collection can be replicated on multiple sites and concurrently updated by each participant. Each object o in this collection is identified by a UUID id_o and its value (or content) can be modified independently on each replica. We assume that objects are atomic and immutable. The collection is a grow-only data structure, where deletions are recorded using a *tombstone*. To efficiently compare different versions of an object, the content x is hashed to produce a string digest $H(x)$.

Object map On each replica of the data structure, 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 allows for retrieving the value associated with a specific digest.

Revision string Let x_N represent the content of the N-th version of object 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), $Tail_N = T(H(r_{N-1}))$, 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 exact content through the embedded digest string $H(x_k)$.

Revision trees Modifications made to an object can be recorded as sequences of *revision strings*. The history of modifications made to each object o across all replicas is represented by a *revision tree* rt_o, which can be conveniently stored as collection of tuples $\langle r_N, r_{N-1}\rangle$ (r_{N-1} being referred to as the *parent* of r_N). The revision tree for a newly created object is $\langle r_1, \emptyset\rangle$. The revision with the highest numerical index is considered the *winning revision* r_W, and determines the contents that shall be returned when querying for the latest version of an object. If multiple revisions share the same index, revision strings are compared in lexicographic sort order.

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.

3.1 Delta-state decomposition

According to [2], 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 $X' = m_\delta(X) = X \sqcup m_\delta(X) = X \sqcup U$, where $X, X' \in S$ and $m_\delta(X) \in S$ is the delta mutation. Delta-mutators m_δ are defined by the relation $m_\delta(X) = X \sqcup U$. 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 [2]) are fulfilled.

3.2 Delta-state serialization and adapters

The serialization format is derived from the one presented in [4]. Update sets are serialized into two different JSON structures, namely *delta blocks* and *data packs*. The former stores revision tree updates, whereas the latter maintains the actual content of each new object. Both structures are immutable, hence they can be cached locally to reduce network overhead. 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 *whose content is available*. To store and replicate data on different platforms, the low-level task of reading or writing the delta blocks and data packs is fulfilled by means of *adapters*. Adapters can be used to seamlessly support different types of storage, such as main memory, filesystems (where delta blocks and packs are files), databases, decentralized data pods, or cloud sharing platforms (such as Dropbox and Nextcloud).

3.3 Example architecture of a collaborative application

The functionalities of the underlying CRDT are exposed through a high-level API which implements methods to update, read and synchronize replicas. An application can make use of these methods to support asynchronous collaborative editing without reinventing its data model.

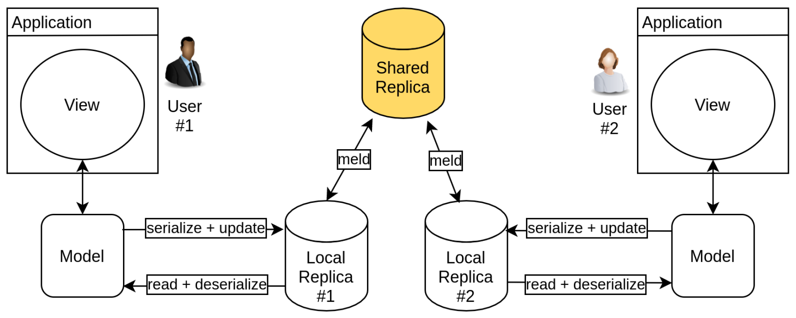

Fig. 1. Example architecture of a collaborative application

Figure 1 shows an example architecture with two local replicas and one shared replica (a simple data store). Each user can work on their data while offline. Replication is achieved by serializing the contents of the data model into a JSON document and subsequently *update* the local replica. By means of the *melding* procedure, changes made to the local replica are propagated to the shared replica through a backend adapter. Afterwards, changes from the shared replica are *melded* into the local one, integrating modifications made by other users. Finally, the local replica can be *read* and deserialized so as to obtain an updated model. This workflow can be used to mimic the co-authoring functionality called *save and refresh* offered by Microsoft Office.

Solid pod adapter The Solid specification [1] provides a standard for building an ecosystem of personal web-accessible data *pods*. Access to a pod is controlled by the owner, and linked data is exploited to promote interoperability between applications and pods. In our framework we store delta blocks and data packs as LDP Resources inside LDP Containers.

4 Evaluation

To evaluate the proposed solution, we consider a synthetic benchmark to determine the space overhead in comparison to Automerge [6]. We employ an editing trace[1] of a large text document with 182 315 single-character insertion operations, and 77 463 single-character deletion operations. 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. As shown in Table 1, as the number of edits in each batch increases, our delta-state CRDT incurs a smaller space overhead compared to Automerge.

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

Batch size	Batches	Cumulative size of the changesets (MBytes)		
		Automerge	Delta-State JSON CRDT	Full-states
10 ops	25 978	56.5	86	1 699.8
100 ops	2 598	54	23.7	170
1 000 ops	260	52.8	16.5	17
10 000 ops	26	51.5	15	1.7

Table 1. Results of the synthetic benchmark after 259 778 edit operations

5 Conclusion

In this paper, we presented a JSON δ-CRDT solution to support the development of multi-user applications based on asynchronous collaboration. Our design is both simple and modular, and by means of adapters, it allows for seamless interoperation between different storage and communication backends, such as Solid pods. Future work include performance optimizations, the implementation of additional adapters, and porting the library to other languages and platforms (such as WebAssembly).

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. Solid technical reports. accessed aug. 5, 2021 (2021), https://solidproject.org/TR/
2. Almeida, P.S., Shoker, A., Baquero, C.: Delta state replicated data types. Journal of Parallel and Distributed Computing **111**, 162–173 (2018)
3. Baquero, C., Almeida, P.S., Shoker, A.: Making operation-based crdts operation-based. In: Proceedings of the First Workshop on Principles and Practice of Eventual Consistency. PaPEC '14, Association for Computing Machinery, New York, NY, USA (2014)
4. Brocco, A.: The document chain: a delta-crdt framework for arbitrary json data. In: SEBD: 29th Italian Symposium on Advanced Database Systems (2021)
5. Brocco, A., Ceppi, P., Sinigaglia, L.: libjots: Json that syncs! In: SEBD: 28th Italian Symposium on Advanced Database Systems (2020)
6. Kleppmann, M., Beresford, A.R.: A conflict-free replicated json datatype. IEEE Transactions on Parallel and Distributed Systems **28**(10), 2733–2746 (2017)
7. Letia, M., Preguiça, N., Shapiro, M.: Consistency without concurrency control in large, dynamic systems. SIGOPS Oper. Syst. Rev. **44**(2), 29–34 (Apr 2010)
8. Nicolaescu, P., Jahns, K., Derntl, M., Klamma, R.: Yjs: A framework for near real-time p2p shared editing on arbitrary data types (06 2015)
9. Rinberg, A., Solomon, T., Khazma, G., Lushi, G., Shlomo, R., Ta-Shma, P.: Array CRDTs using delta-mutations. In: 8th Workshop on Principles and Practice of Consistency for Distributed Data. PaPoC 2021, ACM (Apr 2021)
10. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA (Jan 2011)