The Document Chain: a Delta CRDT framework for arbitrary JSON data

Amos Brocco[1]

[1]*Department of Innovative Technologies, SUPSI, Lugano, Switzerland*

Abstract

In this paper we present a generic δ-CRDT framework for arbitrary JSON data called the Document Chain. In contrast to similar approaches based on explicit update operations, the proposed solution is able to automatically determine changesets for nested data structures. Data can be stored on different backends, and replication can be performed without locking and with non-destructive conflict resolution. The Document Chain is meant to support the development of collaborative applications by exploiting existing communication and storage solutions, such as shared folders or file sharing platforms in the cloud. An evaluation of the performance in terms of storage overhead, replication cost and computation time indicates that the proposed framework compares favorably with other solutions, and is therefore suitable for implementing collaboration features into existing applications.

Keywords

JSON, CRDT, Data Synchronization

1. Introduction

Many collaborative applications enable users to concurrently create and modify content on the web. Some applications provide real-time collaboration and continuous synchronization of the users' activity within a live environment, where each user sees changes made by other users. An alternative approach employs asynchronous replication, which allows for implementing offline-first (or local-first) solutions that can operate even without network connectivity. In order to replicate updates made while offline, when the system is online the application will connect to a server or to another instance, submit updates made to its local replica and possibly retrieve remote changes made on other replicas. The synchronization process itself would normally be performed in a synchronous manner: both the local system and the remote one need to be online at the same time. Furthermore, the synchronization logic must deal with conflicts that arise when the same data is concurrently modified by different users while offline: in this regard, applications not only need to store all the updates and changes made by the user, but must also implement algorithms to safely integrate third-party changes into the data model.

An interesting approach to this problem is provided by conflict-free replicated data types (CRDTs) [1, 2]: these data structures can be replicated between multiple participants in a network, and each participant can update the data independently and concurrently without coordination between the replicas. The logic behind CRDTs ensures that it is always possible to

SEBD 2021: The 29th Italian Symposium on Advanced Database Systems, September 5-9, 2021, Pizzo Calabro (VV), Italy
amos.brocco@supsi.ch (A. Brocco)

resolve conflicts that might arise from those concurrent updates and prevent inconsistencies. Although these technologies solve the problem of data replication, the problem of having an adequate infrastructure for exchanging data between the participants remains. Alongside with the client application, both in the synchronous and the asynchronous approaches, a suitable data exchange infrastructure must also be offered and maintained: such an infrastructure is typically created for the specific needs of the application and might pose some risks concerning security and privacy. To ease the burden of implementing such a complex architecture, we propose a generic framework called Document Chain which enables the use of existing file sharing and communication solutions to provide asynchronous conflict-free replication of JSON data and support the development of non-realtime collaboration features. By exploiting existing platforms (for example, a shared folder or a cloud-based file sharing service), which might already be in use by the end-user, we can reduce the overall maintenance costs for the developer while ensuring that data is stored according to the end-user requirements. The following sections will discuss some of the related work in the field of CRDTs, the architecture and technical details of the proposed solution and an evaluation of its performance.

2. Related work

In [1] the authors provide a definition for conflict-free replicated data types which entails two fundamental properties: first, any replica should be modifiable without any coordination with another replica, second, when any two replica receive the same set of updates they reach the same state. There exist different types of CRDTs: operation-based, state-based and delta. Operation based solutions only record update operations and use specific strategies to merge contents generated on different systems (such as [3]). Operation based CRDTs are suited for high-frequency updates, such as within real-time collaborative text editors. To support those applications, and in particular the ones which rely on JSON data, specific libraries have been developed. An example which is worth mentioning is called *json-crdt* [4], which is based on the *automerge* [5] library. The authors provide several example use cases and applications that exploit the realtime replication capabilities of the library to synchronize several instances of a collaborative online editor. The system performs as intended, but as pointed out by the authors themselves, the very same nature of operation-based CRDTs results in a very large history of change records, and it is almost impossible to remove unneeded historical data (pruning), because if a client comes back online later in time it might require that information to successfully merge its changes. Moreover, some concurrent operations, such as modification and deletion, might produce unexpected results and objects with partially missing data fields. State-based CRDTs [6] always store the full state of the data, and are therefore more suitable in situations with low-frequency updates or where single operations are not commutative. Another advantage of state-based replication is the fact that it is easier to verify the correctness of the data in a particular point in time. State-based CRDTs can be implemented by means of a distributed database such as *CouchDB* [7] or one of its variants (such as [8]). The main drawback of state-based CRDTs is that the storage required to save all states can become very large [9]. To overcome this issue delta-based solutions which rely on disseminating updates (changesets) called delta mutations (referred to as delta CRDT or δ-CRDT) have been proposed

[9]. An important property of such updates is that they are idempotent, which means that they can be applied possibly several times to an existing state without compromising its consistency.

In this paper we present an approach based on deltas called Document Chain, which can be used on JSON documents: delta updates are determined by comparing full states, and are subsequently grouped into blocks which are linked together as a chain. The chain structure keeps changes ordered and ensures consistency. Document Chains are useful as an alternative to state-based CRDTs for arbitrary JSON documents because they reduce the storage requirements for each update while allowing arbitrarily complex documents to be synchronized. As with any CRDT, users can update a Document Chain independently, without the need for locks or any explicit conflict management mechanism. Document Chains follow an eventual consistency model: each sequence of blocks can be used to reconstruct a specific version of the document. Conflicts are dealt with in a non destructive manner by keeping all versions inside the Document Chain and by electing a winning version using a deterministic algorithm. Document Chains can be stored on a filesystem, on a cloud storage, inside a database or just kept in memory: changes can be merged irrespective of the underlying storage technology.

This works improves and extends the one presented in [10] as follows: first, it allows for creating JSON documents which can be concurrently modified by several users without the need for locks and other synchronization mechanisms; furthermore, it introduces commit points which can be used to validate changes and to navigate through the history of the document. Last, it is designed to work not only in-memory but with multiple storage backends, such as shared folders in a local network or file-sharing platforms in the cloud. In contrast to the aforementioned operation-based approaches, the solution discussed in this paper does not require that the client application keeps track of the changes made to its data model in order to update the CRDT accordingly, and thanks to non-destructive conflict resolution it avoids issues such as partially missing data fields when processing concurrent updates and deletions. Compared to existing state-based solutions such as CouchDB, our approach can be easily embedded into an application and does not require an explicit mapping between the the data model of the client application and the internal organization of the database (typically a collection of JSON objects).

3. Technical overview

The implementation of the Document Chain comprises two key elements: a suitable CRDT data model and a high-level API for accessing the document chain and enabling the use of different types of storage backends. In this section both elements will be presented and discussed.

3.1. CRDT for arbitrary JSON documents

At the lowest level the CRDT employed in our solution improves on the work presented in [10], and allows for an arbitrary JSON document (referred to as the client application's own document) to be processed as input data. In contrast to operation-based CRDTs, the client application is not required to provide an explicit list of editing operations.

The framework implements a two-step difference detection algorithm to determine the changes between the input data (an arbirary JSON document) and the latest available state of the CRDT. In the first step (called flattening [10]), the input document is processed to destructure nested objects: each one of those objects is assigned a unique identifier (according to some user-specified rules) and is stored in an output collection. Since the flattening procedure needs to be reversible, during flattening nested objects are replaced with string references. In contrast to [10], we've been able to significantly reduce the storage space generated by changes made to nested arrays by storing only differential updates to ordering documents. In the second step, each object that has been placed in the output collection is compared with the latest version found in the chain, in order to determine the correct update operation (creation, deletion or update). Each operation is subsequently recorded as a transition between two revisions of an object, alongside with the contents of new revisions. Revision strings are comprised of a numerical index and the hash value of the object. To support CRDT operations, the contents (user data) and revision information are separated into two different structures, namely *data packs* and *revision update records blocks.*

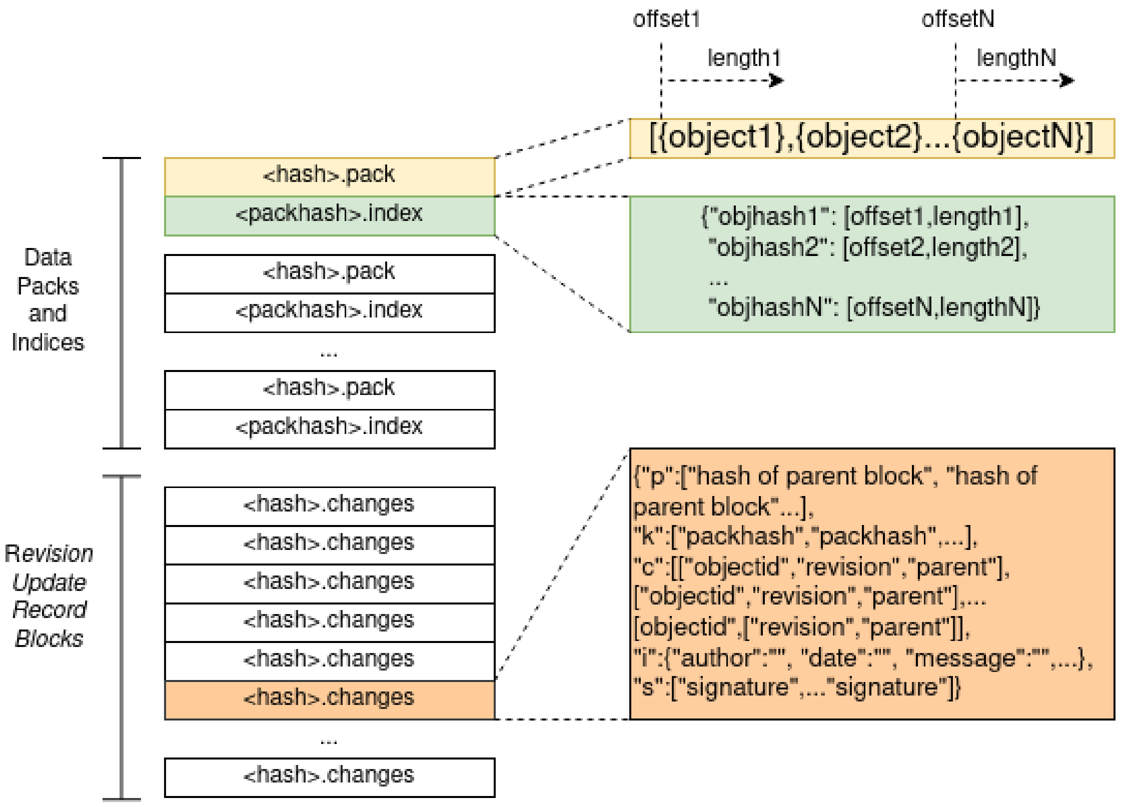
offset1
length1
offsetN
lengthN
[{object1},{object2}...{objectN}]
Data Packs and Indices
<hash>.pack
<packhash>.index
<hash>.pack
<packhash>.index
...
<hash>.pack
<packhash>.index
{"objhash1": [offset1,length1],
"objhash2": [offset2,length2],
...
"objhashN": [offsetN,lengthN]}
Revision Update Record Blocks
<hash>.changes
<hash>.changes
<hash>.changes
<hash>.changes
<hash>.changes
<hash>.changes
...
<hash>.changes
{"p":["hash of parent block", "hash of parent block"...],
"k":["packhash","packhash",...],
"c":[["objectid","revision","parent"],
["objectid","revision","parent"],...
[objectid",["revision","parent"]],
"i":{"author":"", "date":"", "message":"",...},
"s":["signature",..."signature"]}

Figure 1: 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.

3.1.1. Data packs

Objects extracted during the flattening procedure are stored within pack files: each object is uniquely identified by the hash value of its contents to avoid duplicates. To efficiently enumerate the objects contained in a pack file and retrieve their content, a corresponding index file is also generated. The index maps the hash value of an object to a position inside the pack file. 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 redundancy purposes, pack files can also be parsed as collections of JSON documents (since they represents a JSON array), therefore the corresponding indices can be recovered if necessary. Indices also provide an advantage in a distributed environment, since they are smaller in size than the corresponding packs, can be cached locally (since they are immutable) and allow for determining the available content without the need for downloading the data packs.

3.1.2. Revision update record blocks

Revision update records store a log of all changes made to a particular object: more specifically, they record a transition between the previous and the next revision of one or more objects. Since revision strings contain the hash value of the contents of an object, it is trivial to recover the actual data by fetching the corresponding pack file. Records are grouped inside blocks, which correspond to a specific commit point made to the application data. Blocks also store the identifier of all related pack files created for the same commit point, and optional metadata such as the author and date of the update or digital signatures to prove the authenticity of the block. Blocks are identified by a key obtained by computing the hash value of their content. To keep track of dependencies between subsequent commit points and determine the order in which blocks need to be processed, each block also stores references to one or more previous blocks (referred to as *anchors*). If concurrent edits happen, multiple blocks might reference the same *anchors*: this situation won't affect the consistency of the chain, because conflicts are dealt with by a deterministic algorithm.

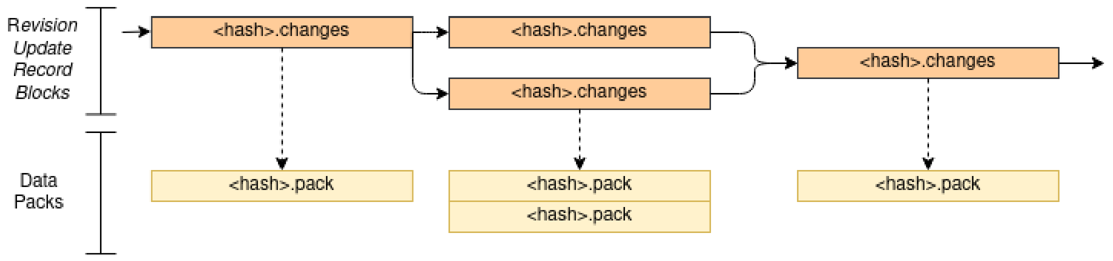

Figure 2: The Document Chain - Logical overview: Revision Update Record Blocks are denoted using the *.changes* suffix, whereas Data Packs use the *.pack* suffix; arrows represent *hash references* between elements in the chain.

3.1.3. Document chain

The history of all changes made to the application's own JSON document, as recognized by the library, is stored as a collection of linked revision update blocks and data packs called Document Chain (as shown in Figure 2). To obtain the latest version of the client's data, the changes recorded in all update record blocks are replayed and data packs are read to source the required revisions for each object. The unflattening procedure [10] is subsequently executed on the resulting collection of JSON objects to reconstruct the full state of the JSON document. Since each block in the Document Chain represents a separate commit point, it is possible to navigate through the history of all committed changes and retrieve a specific version of the JSON document.

Figure 1 illustrates the data structures used inside a Document Chain (data packs, indices and revision update record blocks). It should be noted that those structures can reside on different kinds of storage, such as the local filesystems, shared folders in a company's intranet, databases or cloud platforms, and can be updated independently by each participant without locking.

3.1.4. 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.

3.2. 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 new chain or merge the current state into another chain. 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. We are currently experimenting with a filesystem adapter (which stores blocks and packs as files), adapters for cloud sharing services (such as Dropbox and Nextcloud), a database adapter (for storing data inside relational databases such as MariaDB or SQLite), and an in-memory adapter.

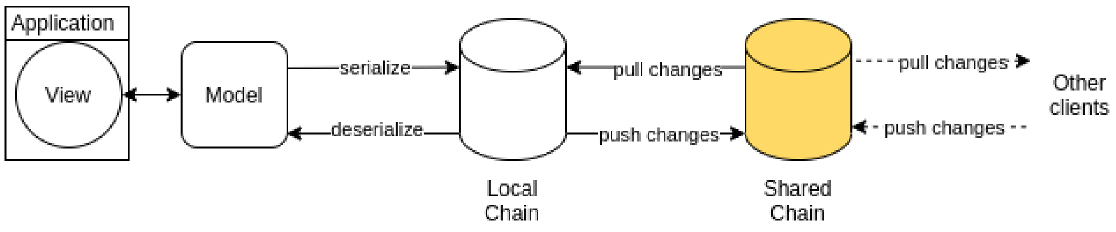

Figure 3: The Document Chain - Collaborative Application Architecture

3.2.1. 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 3, it is possible to employ different Document Chains (in the example, a *shared* one and *local* ones) to exchange modifications made by 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.

4. Evaluation

To evaluate the Document Chain solution we present 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 simulate 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 and recipient accounts. Furthermore, the root object also contains a filed named *info* which maps to an object with a transaction counter (associated with the *txcount* key).

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

Listing 1: Sample JSON document V_1

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 value of the *currency* field of two existing objects is replaced with new data (specifically with the "EUR" or "USD" 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: 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 and block/pack names inside the chain, we consider both SHA256 (with a 256 bit digest) and xxHash (with a 64 bit digest). 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.

4.1. 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 4 the Document Chain approach results in an overhead comparable 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 the full state as well as the cumulative size including all previous states. In this regard, both CRDTs allow for maintaining the full history of a document with considerably less storage overhead.

4.2. 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

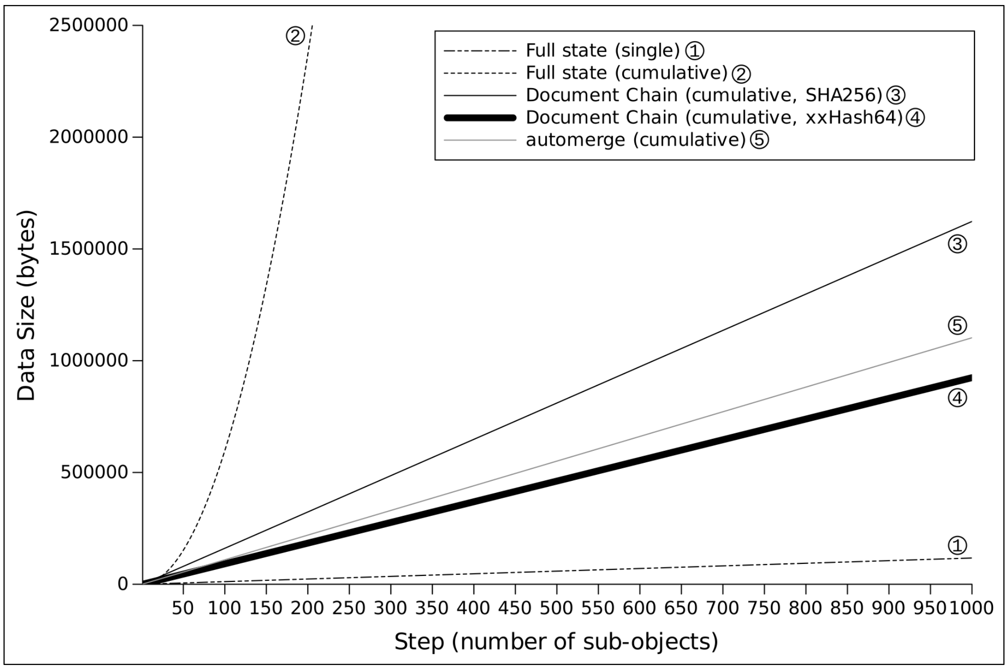

Figure 4: Storage Overhead

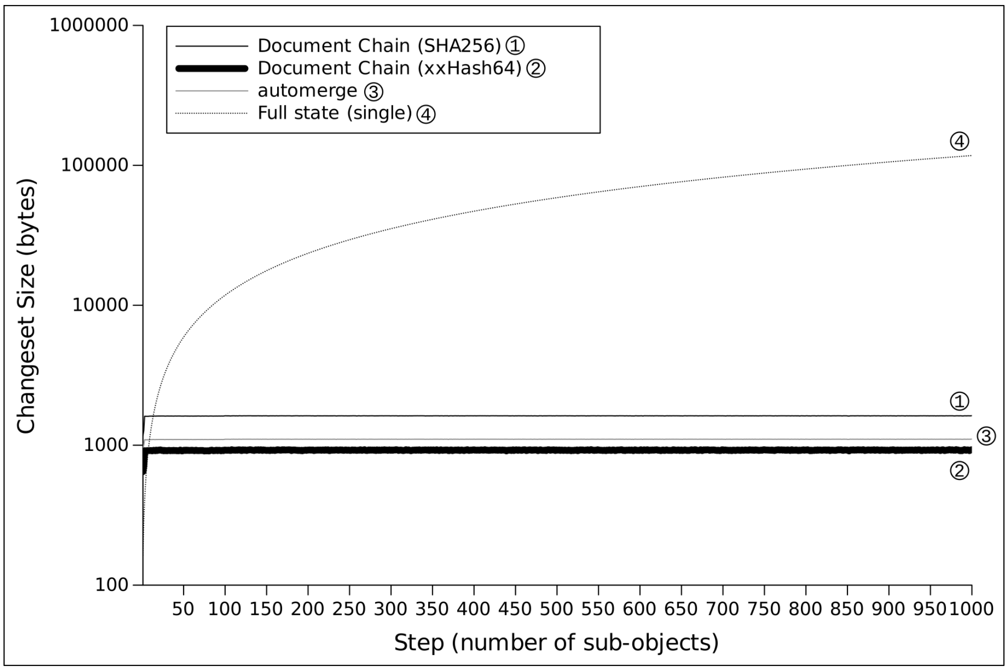

Figure 5: Size of the changeset

that needs to be exchanged between participants in order to update their current state. As can be observed in Figure 5, 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.

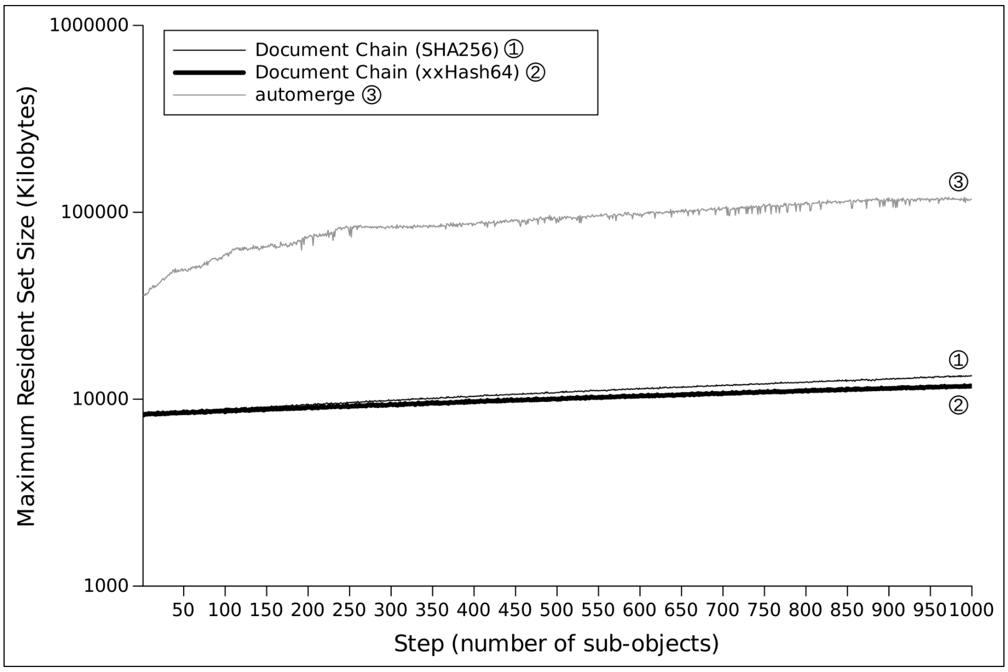

Figure 6: Maximum resident set size (RSS)

4.3. Maximum resident set size (RSS) for 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 the built-in *process.memoryUsage().rss*. As shown in Figure 6, *automerge* requires more memory than the Document Chain. This difference can be explained by the fact that all changesets produced and replayed by *automerge* need to be loaded in memory (those also contain the edited values), whereas the *Document Chain* keeps data inside packs which remain solely on disk until they are needed during the reconstruction process. Furthermore, *automerge* is a Javascript library which depends on the garbage collector of *Node.js*, whereas the Document Chain is a native library written in C/C++ which performs manual memory management.

4.4. Full state reconstruction time

Reconstructing the full state requires replaying all changesets. 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 time before and after the process (therefore the initialization time of the Node.js runtime is ignored). As shown in Figure 7, *automerge* takes considerably more time compared to Document Chain (either using the SHA256 algorithm or xxHash). In this regard, an important contributing factor is that with the former all changesets need to be loaded and replayed, whereas with the latter only revision update record blocks and indices need to be fully processed, as only the relevant packs (those that contain referenced objects) need to be subsequently read.

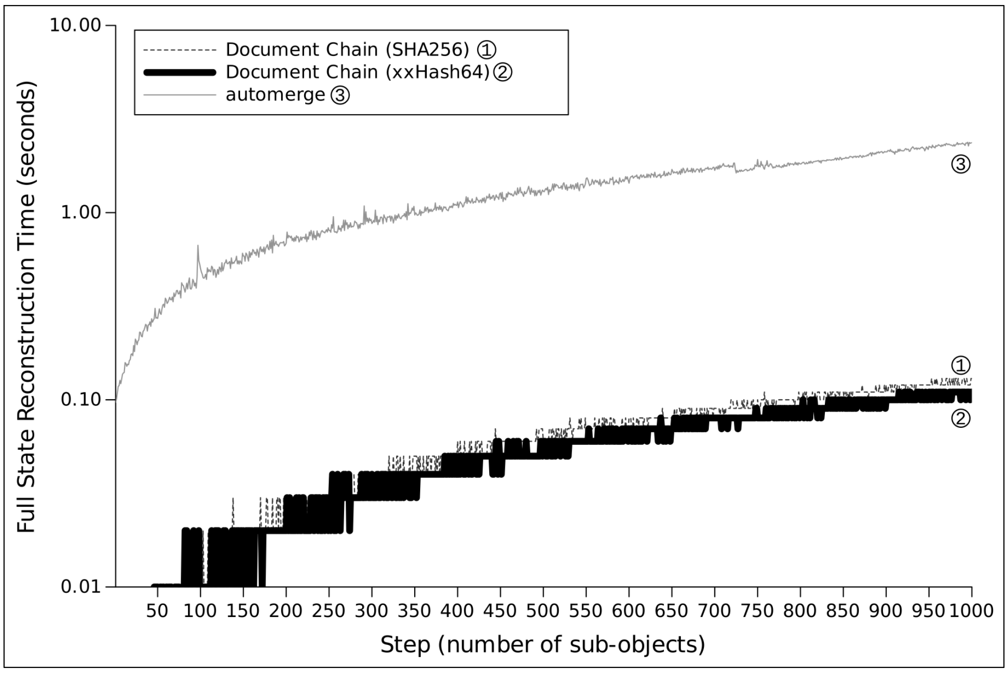

Figure 7: Full state reconstruction time

5. Conclusion

In this paper, we presented the concept of Document Chain, which implements a δ-CRDT framework for arbitrary JSON data. The proposed solution supports the development of collaborative applications by exploiting existing communication and storage solutions, such as shared folders or cloud services. The chain can be replicated and independently modified by multiple users: the replication process is efficient and ensures non-destructive conflict management. Moreover, since the full history of all the changes made to the document is recorded into the chain, it is possible to retrieve previous versions with ease. The Document Chain compares favorably with similar CRDTs, such as *automerge*, while offering a simpler interface which does not require explicit updates to the document to determine state changes. Future work includes the integration of an adapter for the SOLID infrastructure [11], in order to store data inside a private pod.

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] N. M. Preguiça, C. Baquero, M. Shapiro, Conflict-free replicated data types (crdts), CoRR abs/1805.06358 (2018).

[2] M. Letia, N. Preguiça, M. Shapiro, Consistency without concurrency control in large, dynamic systems, SIGOPS Oper. Syst. Rev. 44 (2010) 29–34.

[3] C. Baquero, P. S. Almeida, A. Shoker, 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] P. Grosch, R. Krafft, M. Wölki, A. Bieniusa, Autocouch: A json crdt framework, in: Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC '20, Association for Computing Machinery, New York, NY, USA, 2020.

[5] M. Kleppmann, A. R. Beresford, A conflict-free replicated json datatype, IEEE Transactions on Parallel and Distributed Systems 28 (2017) 2733–2746.

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

[7] Couchdb 2.0 reference manual, 2015.

[8] pouchdb, the javascript database that syncs!. accessed apr. 23, 2021, 2020. URL: https://pouchdb.com.

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

[10] A. Brocco, P. Ceppi, L. Sinigaglia, libjots: JSON that syncs!, in: M. Agosti, M. Atzori, P. Ciaccia, L. Tanca (Eds.), Proceedings of the 28th Italian Symposium on Advanced Database Systems, Villasimius, Sud Sardegna, Italy (virtual due to Covid-19 pandemic), June 21-24, 2020, volume 2646 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020, pp. 116–127.

[11] Solid technical reports. accessed apr. 23, 2021, 2020. URL: https://solidproject.org/TR/.