libJoTS: JSON That Syncs!*

Amos Brocco[1], Patrick Ceppi[1], and Lorenzo Sinigaglia[2]

[1] Department of Innovative Technologies, SUPSI, Manno, Switzerland
{amos.brocco,patrick.ceppi}@supsi.ch
[2] Banana.ch SA, Lugano, Switzerland
lorenzo@banana.ch

Abstract. In this paper, we present *libJoTS*, a C/C++ library that provides asynchronous offline replication of arbitrary JSON data with a minimal footprint. The primary use case for this library is to easily replicate changes made to a JSON file by different users. The replication process is based on multi-version concurrency control (MVCC) to ensure non-destructive conflict management, but in contrast to other popular databases, *libJoTS* has no runtime dependencies, supports hierarchical documents with nested objects, and stores all synchronizable data into a file. The library is self-contained, and provides a simple API for C/C++ programs as well as bindings for other languages, such as Java and Python. As such it can be easily linked into any application, allowing end-users to replicate changes to data asynchronously and offline. By using files, data sharing between users does not depend on any specific technology, and can be achieved either with an online file-sharing service or offline, thus ensuring full control on data privacy.

Keywords: JSON · MVCC · Data Synchronization

1 Introduction

Nowadays, many popular applications that once ran natively (offline) on personal computers are being pushed to the cloud. At user level, one of the most significant improvements brought by cloud applications is the ability to easily share data with other people and to work online in a collaborative fashion. In this regard, replication protocols [1–3] are the cornerstone of those multi-user applications, allowing updates made to each replica to be consistently merged together. Replication can be either synchronous or asynchronous. With synchronous replication, often simply referred to as synchronization, there is a continuous flow of information that propagates changes to all replicas: as soon as new data is

* This work has been financially supported by the Swiss Innovation Agency, Project nr. 26367.1 PFES-ES.

being created or updated, data is (atomically) replicated to all remote copies. With asynchronous replication, changes are not propagated in realtime, and the system might even tolerate temporary divergence between replicas. While the synchronous approach ensures that each replica always contains the same data, the overall performance and availability is affected by network availability, reliability and speed. Concerning cloud applications, developers are currently advised to develop software that is able to work even when there is no connectivity: in this context asynchronous replication mechanisms play an important role in allowing users to work on local data, while enabling synchronization with a remote server when online. Beside technical aspects, sharing data on the cloud and storing potentially sensitive data on someone else's computer, introduce privacy, security and legal concerns [4, 5]. In this regard, the user should be able to retain as much control as possible on his/her information, either by relying on offline applications and local file storage, or by using connected applications that offer an open or generic synchronization layer that does not depend on any vendor-specific cloud API. The first approach allows for selective disclosure of information (the user decides whom to send his/her information to), but typically makes synchronization between different versions of the same data difficult. The library presented in this paper, called *libJoTS*, tackles this issue by implementing an offline asynchronous replication mechanism for arbitrary JSON data. *libJoTS* allows any native application to store data in a synchronizable format; moreover because data is always stored as a file on disk, remote sharing can also be performed using any means, like e-mail or file-hosting platforms. Finally, synchronizable files can be easily encrypted (using either symmetric or asymmetric algorithms) and/or digitally signed to prove that data was not altered.

2 Related Work

Data replication and synchronization technologies are the cornerstone of distributed storage systems and cloud-based content editing applications (for example Google Docs™). Although there exist different algorithms and protocols to keep data synchronized [8], we restrict our focus to asynchronous replication, which represents the closest concept to the proposed solution. With asynchronous replication, updates on the master (or source) storage are not copied immediately to a secondary storage (or target), but can be delayed to a later time. Such an approach gives up on strict consistency and update timeliness across the network, while supporting the development of *offline first* solutions, which can operate even without a permanent connection between clients and servers on the network. In this regard, a well-known technology is CouchDB [3], a document-oriented NoSQL database which supports multi-master asynchronous replication: each instance can be independently updated, and changes can be replicated between any instance. Unfortunately, CouchDB is not suited for deployment on mobile devices or the web, therefore similar databases that share

https://developer.chrome.com/apps/offline_apps

the same replication protocol have been developed (for example, Couchbase Lite or PouchDB). With those solutions it is possible to build applications that can operate on a local database while offline and asynchronously replicate changes to another peer when a network connection becomes available. With *libJoTS* we aim to provide transparent offline data replication, by generating a synchronizable file which can be stored and exchanged through any communication channel. Such a synchronizable file might also be signed and/or encrypted, allowing for accountability and/or confidentiality of the data being exchanged. In contrast to a diff/patch [10] solution (which also work on files), *libJoTS* is tailored for semi-structured data in JSON format and ensures that the structure is maintained in the merged document. Data is versioned and each revision is stored directly into the file, allowing for historical data retrieval. Like CouchDB, conflict management is based on MultiVersion Concurrency Control (MVCC): accordingly, replication can be performed among different synchronizable files in no specific order, without compromising the structure of the underlying data. Although there exist comparison tools expressly conceived for JSON data (see for example, RFC6902), those solutions do not maintain an historical record of all versions of the data and might still require user intervention to ensure consistency in the event of concurrent conflicting modifications. In contrast to some of the previously cited solutions, *libJoTS* does not require the source data to be decomposed into a collection of documents, and the library can be easily integrated into desktop or mobile application, being self-contained and written in C++ (with a C API and bindings for other languages, currently Java and Python, and the possibility of targeting WebAssembly in the future).

3 Use Cases

With *libJoTS* we aim at supporting the development of distributed applications that rely on synchronized data without the need for a dedicated server or a specific cloud platform, by enabling information sharing regardless of the type of communication channel (for example, shared folder, e-mail or portable storage). Each user is in control of the data and can decide when to share its own version or merge changes from other users. In the following two use cases are presented.

3.1 *Offline first* application

We consider a desktop or mobile application that lets users edit documents offline on their device and later share their work with other people. User A produces the first version of document X; this version is sent to users B and C through a cloud-based file sharing service. Both user A, B and C make their changes to the document. Subsequently, user B sends its own updated version to User A by email, who can then merge the changes made by B in its copy. At the same time

https://www.couchbase.com
https://pouchdb.com
https://tools.ietf.org/html/rfc6902

user C sends its own version to B by email. User B integrates the changes made by C, merges incoming updates from A and then forwards its final version of the document to C (Figure 1). Users can perform arbitrary editing and replication, and the process can be scaled to an even larger group of users. Moreover, each user can store copies of synchronizable files for archival or backup purposes: those file can be later processed by the library in order to restore data in the application's own format or to peform synchronization.

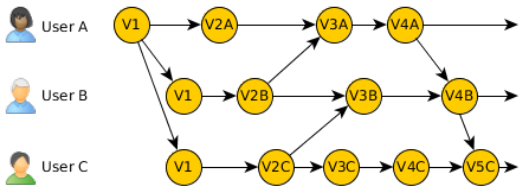

Fig. 1. *Offline first* sample workflow with multi-master replication.

3.2 Distributed configuration management

A common problem in distributed systems is configuration management. Configuration updates need to be replicated from a central server to a multitude of nodes. By means of synchronizable JSON files, *libJoTS* can be used to create mergeable configuration updates: because conflict handling is non-destructive, local changes are preserved while updates to global settings can be seamlessly integrated. Configuration updates can also be stored on disk to be merged later, and the authenticity of an update can be verified by means digital signatures.

4 libJoTS Data Model

The data model used by *libJoTS* is based on JSON documents. Similar to CouchDB the basic unit of replication is a JSON object: the replication algorithm compares two collections of JSON objects and determines the ones missing on each collection. Whereas, CouchDB and other document-oriented databases (like PouchDB) require the application to organize its data into separate JSON objects, *libJoTS* accepts either a collection of separate objects or an arbitrary JSON document (with a nested hierarchy of objects), and can seamlessly replicate their changes. Therefore, from an application's point of view, if JSON is already employed as a data interchange format, little to no change is required to make use of the synchronization capabilities of the library. The proposed approach for dealing with hierarchical JSON documents is to automatically transform an arbitrary input (for example, Listing 1.1) into a collection of objects which can be synchronized independently.

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

Listing 1.1. Sample application JSON (*unflattened*, version V_1)

The output of such a data transformation (Listing 1.2) is a *flattened* collection where array of objects from the input document have been replaced by a string reference, and their contents have been promoted to *first-level* objects. Optionally, nested objects can be promoted too.

```
[{"_id":"391...32c","currency":"!CHF","from":"!13465-45566","to
    ":"!34655-67554","value":22412},{"_id":"@123...8b1-u","s":["391473a1-bb89
    -4dbf-af93-7a8cc1ebc32c"]},{"_id":"root","data":{"transactions":"@123...8
    b1"},"info":{"txcount":1}}]
```

Listing 1.2. *Flattened* collection of JSON documents corresponding to Listing 1.1. Hash values (SHA-256) have been ellipsized for clarity.

We call this procedure *flattening*: each promoted object is associated with a unique identifier, which can be either created using rules specified by the user or automatically generated by the library using a hash algorithm such as SHA-256 [7], xxHash or BLAKE2. The *flattening* transformation is non-destructive and can be undone to restore the original *unflattened* hierarchical structure. To recover the order of the elements inside *flattened* arrays, we generate *ordering* documents with a list of identifiers which belong to the corresponding array in the *unflattened* JSON document. Listing 1.2 illustrates an ordering document (referenced by the *@123...8b1-u* identifier). To prevent the synchronization process from changing the relative order of the elements (as chosen by different users), ordering documents can be tied to specific user/instance identifiers: hence each user can synchronize data without losing their ordering. Moreover, strings are *escaped* using the '!' character to differentiate them from references to arrays or promoted objects.

```
{"d":{
  "391...32c":["d29...d0b"],
  "@123...8b1-u":["3fc...e82"],
  "root":["cc6...5d6"]},
 "o": {
  "3fc...e82":{"s":["391...32c"]},
  "cc6...5d6":{"data":{"transactions":"@123...8b1"},"info":{"txcount":1}},
  "d29...d0b":{"currency":"!CHF","from":"!13465-45566","to":"!34655-67554",
      "value":22412}},
 "r":"root"}
```

Listing 1.3. Synchronizable JSON document corresponding to Listing 1.1.

4.1 Synchronizable JSON

A *Synchronizable JSON* file contains a *flattened* collection of JSON objects along with versioning information. Similar to MultiVersion Concurrency Control (MVCC), concurrent updates and replication never overwrite existing data, but

https://cyan4973.github.io/xxHash
https://blake2.net

simply create a new revision for the concerned objects. The revision string is computed by *hashing* the contents of the object. All revisions but the first one have an *ancestor*: for each object, we maintain the history of revisions (also known as the *revision tree*), which is stored inside the *Synchronizable JSON*). By consulting the revision tree it is possible to determine the order of each update and thus the most recent revision (known as the *winning* revision), which coincides with the tip of the longest branch of the tree. The Synchronizable JSON data corresponding to Listing 1.1 is shown in Listing 1.3. The value associated with the key *d* (for *documents*) is a dictionary of all objects' identifiers and the corresponding revision trees, whereas the value corresponding to *o* (for *objects*) is a dictionary with the contents of each revisions. Finally, the *r* key points to the identifier of the root element (which is needed to reconstruct the original JSON document).

The replication algorithm implemented in *libJoTS* merges all changes found in a *source* document into a *target* document. Given two collections of versioned documents C_1 and C_2, the replication of C_1 to C_2 will copy all the documents and revisions of C_1 that are missing in C_2. At the same time, the revision tree for each document will be updated accordingly. If more than one revision shares the same ancestor a conflict will arise: in that case, a deterministic algorithm will choose the winning revision based on the longest revision branch and lexicographical comparison. This algorithm achieves replica convergence and eventual consistency using MVCC, as in [6].


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


Listing 1.4. Updated JSON document (version 2a)


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


Listing 1.5. Updated JSON document (version 2b)

As an example of the whole replication process, consider the two updated documents in Listings 1.4 and 1.5. Both have been produced starting from version 1: in version *2a*, the value associated with key *currency* has been changed from *CHF* to *EUR*, in version *2b* the value has been changed to *USD*.

The corresponding synchronizable documents can be merged together to produce a new version (Listing 1.6). Both the data contained in version 2a and in version 2b are present in the resulting document, and the revision tree for *391...32c* contains a reference to both revisions. The deterministic conflict resolution algorithm will choose either one of the revisions as *winner* (in this example, the result corresponds to Listing 1.5).


```
{"d":{
  "391...32c":["d29...d0b",["151...4cf_d295e90","8be...7cb_d295e90"]],
  "@123...8b1-u":["3fc...e82"],
  "root":["cc6...5d6"]},
 "o":{
  "151...4cf":{"currency":"!EUR","from":"!13465-45566","to":"!34655-67554",
       "value":22412},
```

```
  "3fc...e82":{"s":["391...32c"]},
  "8be...7cb":{"currency":"!USD","from":"!13465-45566","to":"!34655-67554","
      value":22412},
  "cc6...5d6":{"data":{"transactions":"@123...8b1"},
      "info":{"txcount":1}},
  "d295...d0b":{"currency":"!CHF","from":"!13465-45566","to":"!34655-67554",
      "value":22412}},
 "r":"root"}
```

Listing 1.6. Merged Synchronizable JSON document (2a,2b)

Because each revision is kept inside the *Synchronizable JSON* file, the more the data is updated, the bigger the size of the file becomes. This problem can be mitigated by enabling the built-in compression of the *Synchronizable JSON* file. To further limit this growth, we make use of pruning to delete the value associated with old revisions. More specifically, the *pruning* operation removes stale (i.e. non-winning) branches in each revision tree and deletes unreferenced revision objects. The client application can select the depth of the pruning operation in order to maintain an historical record of previous data values (for example, to provide an *undo* functionality).

5 API

To support the basic operations related to Synchronizable JSON files, the public API provides four core functions :

- `sync_json_update` updates or creates a *Synchronizable JSON* document from an arbitrary JSON object (in the application's own format). Since the library is stateless, the *Synchronizable JSON* object must be stored (i.e. saved to a file) by the application itself.
- `sync_json_read` reads a *Synchronizable JSON* document and converts it back into an application-compatible JSON document.
- `sync_json_replicate` replicates changes from a source *Synchronizable JSON* document to a target one.
- `sync_json_prune` removes historical information (i.e. old revisions) from a *Synchronizable JSON* document.

5.1 Offline Workflows

To better understand how the library is meant to be used by an application, in the following we present the expected workflow covering both the creation of *Synchronizable JSON* documents and offline replication of changes.

Updating a Synchronizable JSON file To generate the first version of a *Synchronizable JSON* document, the application needs to provide the input JSON file to the `sync_json_update`. The output document contains all synchronization information required for subsequent replications and can be shared with other

Additional functions for manipulating Synchronizable JSON are also available.

users. In order to update a *Synchronizable JSON* it is necessary to provide a *Synchronizable JSON* to the `sync_json_update` function, along with the updated JSON data. In a distributed scenario several *Synchronizable JSON* documents can be updated in parallel, independently from each other, without loosing the ability to later replicate their changes. In this regard, each document represents a different branch in the version tree.

Reading a Synchronizable JSON file Synchronizable JSON can be converted back to the application's own format using the `sync_json_read` function. By means of additional procedures provided by the library, specific revisions of the data can also be obtained.

Replicating a Synchronizable JSON file Using the `sync_json_replicate` function it is possible to replicate changes made to a *source* Syncronizable JSON file into a *target* file: we refer to this process as *offline replication*. Because each element of the document is versioned, the replication process does not lose any data (non-destructive conflict management). The user can also query for the differences between the new and the old version, as well as conflicting data.

5.2 Online Replication Workflows

Provided that both the source and the target peers are online at the same time (or are able to directly communicate), it is possible to optimize the replication process by sending only bits of modified data. Although not directly implemented by the library, two approaches are proposed: a *diff/patch* mechanism, and an *interactive replication protocol*.

***JSON Patch* approach** The *JSON Patch* solution requires a RFC6902 compliant *diff* and *patch* tool to produce a sequence of operations to apply to a target JSON file (at version V_M) in order to update its structure to be equal to a source version V_N. As mentioned previously, patching the application's JSON does not provide a robust solution to the replication problem, since data might be inadvertently overwritten or corrupted due to conflicting modifications; therefore we consider generating patches against Synchronizable JSON files in order to maintain the full revision history. The target peer needs to communicate its version V_M of the data to the source peer. The source peer then employs a JSON *diff* function to produce a *patch* needed to update V_M to V_N. Unfortunately, such an approach makes synchronization difficult (if not impossible) in a multi-master scenario, where multiple versions of a document with different replication histories arise from concurrent offline synchronizations made by different users: in this situation patches might not apply cleanly leading to data corruption.

Such as https://github.com/nlohmann/json

Interactive replication protocol If both the source and the target peers are online, a more efficient replication can be implemented through an interactive protocol.The source peer sends an *offer* message to the target, containing a list of document identifiers and their associated winning revision. The target replies with an array of identifiers of documents that are either unknown or whose latest revision is older than the one proposed by the source. Finally, the source provides a Synchronizable JSON document with just the missing information. Compared to the *JSON Patch* solution, an interactive replication protocol generates more traffic but does not require the source peer to keep all previous versions of a document. Moreover, it can replicate changes between different *branches* (i.e. Synchronizable JSON documents that have a different revision history). The library also provides methods to migrate data to and from CouchDB-compatible servers, hence replication through another database is also possible.

6 Benchmarks

We evaluate the performance of our library by considering a stepwise editing workflow of a JSON document. The evaluation process begins with version V_1 of a simple JSON document containing an array of objects (Listing 1.1). We subsequently create a new version by performing some changes to the data. More specifically, version V_N is modified to produce version V_{N+1} as follows: a new object is added to the `transaction` array, and the contents of an existing object (except the `_id` field) are replaced with new data. Moreover the value of the field `txcount` is updated to reflect the size (number of elements) of the `transaction` array. At each step, version V_N of the Synchronizable JSON and version V_{N+1} of the application JSON are processed by the `sync_json_update` function to produce version V_{N+1} of the Synchronizable JSON. The resulting document is *pruned* to remove all but the latest revision of the internal data. The last version of the Synchronizable JSON is subsequently replicated on the second-last version using `sync_json_replicate`, and the replication result is converted back to the application's own format using `sync_json_read`. The execution of all operations is profiled in order to measure the required time, the maximum amount of main memory used during the process (maximum RSS), and the resulting file size (for both the Synchronizable JSON and the application JSON). The library was configured to use the *SHA-256* hashing algorithm for generating revision hashes and compression was disabled (unless otherwise specified). All tests were repeated 10 times on a 64-bit Ubuntu Linux machine with an Intel© Core™ i7-6500U CPU running at 2.50GHz with 16GB or memory. The library was compiled with *gcc* version 9.2.1: the actual size of the library is less than 900 KB, with dependencies only against the standard library and the *pthreads* library. For comparison, a minimal install of CouchDB 2.3.1 takes about 40 MB, whereas version 7.1.1 of PouchDB (without JS runtime) is 123 KB.

Results also show the file size using the xxHash algorithm for comparison purposes.

6.1 Offline replication results

As shown in Figure 2, the time required to process JSON data using the library grows linearly with respect to the size of the file. In this regard, we note that the size of the final version of the file (containing a total of 1000 sub-objects) is about 133 KB, and that the total includes also the time spent loading the file from disk. As expected, reading Synchronizable JSON back into the application's own format is the least expensive operation, whereas updating or replicating data takes almost the same time. The growth in each graph is marked by a series of steps due to the allocation of memory for the internal structures of the library.

Additional testing allowed us to assess the scalability of the replication algorithm with respect to the input file size: on the same hardware, generating a Synchronizable JSON document from a 65.5 MB file with 335,389 sub-objects requires about 18 seconds (with the flattening process taking about 80% of the time), and produces a 110 MB output file. Merging two Synchronizable JSON files of such a size (where one file contains 479 updated objects with respect to the other) takes about 12 seconds, while converting the result back to the application's *unflattened* format requires about 10 seconds. With large files the total cost is clearly dominated by parsing the input JSON, which is a known problem when dealing with this format [9]. Trying to import the same amount of data into PouchDB (using the bulk docs interface) freezes the program.

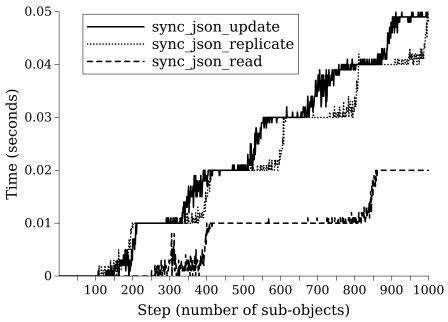

Fig. 2. Time required *vs* number of objects.

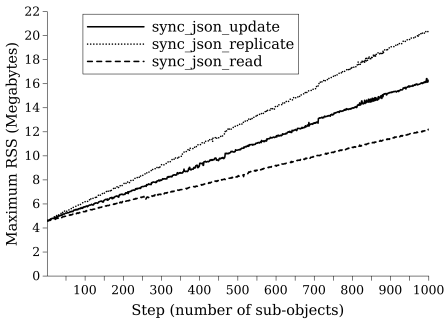

Fig. 3. Maximum Resident Set Size (RSS) *vs* number of objects.

Concerning memory allocation, Figure 3 illustrates the memory required for updating, reading and replicating data using the library. In each case the growth follows the increase in the size of the input document. Replication requires the largest amount of memory, since data from both input files need to be stored at the same time; as expected, reading takes the least memory amount. Unfortunately JSON parsing is memory intensive, and very high requirements are to be expected when dealing with large files: with the previously mentionend 110 MB Synchronizable JSON files, replication requires almost 3 GB of memory.

Figure 4 takes into consideration the cost for offline synchronization, by comparing the file size of both the original JSON file (application JSON) and Synchronizable JSON. The graph also shows the difference between two hashing algorithms (SHA-256 and xxHash) which can be used to generate object and revision identifiers. Even though pruning was used to limit the amount of historical information kept with the file, Synchronizable JSON clearly takes more space on disk compared to application JSON, because of the additional information required to support offline replication (namely, revision trees). It is interesting to note that xxHash allows for more compact synchronizable files compared to SHA-256, because it generates 64 bit hashes instead of 256 bit ones. If Synchronizable JSON files are not meant to be processed outside the library, built-in compression (based on the deflate [11] algorithm) can be enabled to further reduce the size of the file by about 40%. With respect to the workflow involving other databases with support for replication, the size of Synchronizable JSON is typically smaller: the on-disk space taken by PouchDB for the same information (stored as a structured log) is 1.3 Megabytes.

6.2 Online replication results

In order to asses the benefit of online replication (either using the *diff/patch* approach or the *interactive replication protocol*) we repeated the experiment discussed in the previous section assuming that new versions are generated on a source peer while measuring the amount of data that needs to be sent to a target peer in order to keep both replicas synchronized.

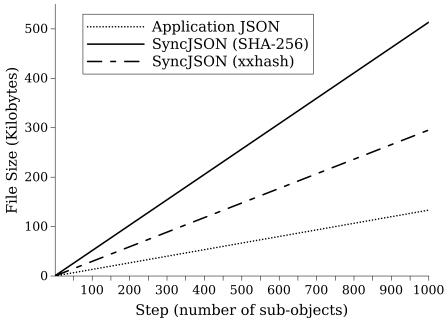

Fig. 4. File size *vs* number of objects.

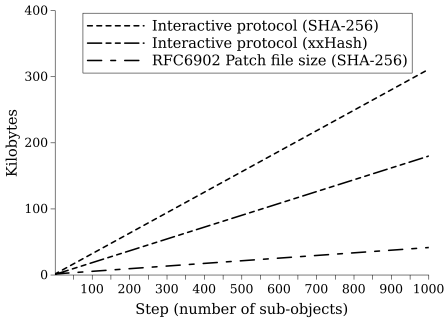

Fig. 5. Amount of exchanged data *vs* replication approach.

As shown in Figure 5, exchanging the full synchronizable JSON file consumes more bandwith than online protocols; the traffic generated using the RFC6902 approach is also significantly lower than the interactive protocol, because only a small amount of information needs to be sent to the target. However, such an approach would not work in a multi-master scenario, where multiple branches

need to be synchronized. Although not shown in the graph, the resulting traffic is not meaningfully affected by the hashing algorithm, with *xxHash* producing the same amount of data as *SHA-256*.

7 Conclusions

In this paper, we presented *libJoTS*, a C/C++ library that provides asynchronous offline replication of arbitrary JSON data. The primary use case is to easily replicate changes asynchronously and independently made to a JSON document by different users. The replication process is based on multi-version concurrency control (MVCC) to ensure non-destructive conflict management. The proposed approach draws inspiration from popular document databases with built-in support for replication, but provides a standalone lightweight solution targeting offline replication of small to moderately sized JSON documents. Synchronizable JSON files can be stored on disk, digitally signed and exchanged with any type of communication technology in order to be later merged on a remote system. The API focuses on an offline-first approach, allowing for exchanging synchronizable files using any technology, but also enables online replication using a simple protocol. The library, which we plan to release in the forthcoming months, is self-contained and can be easily integrated into any desktop or mobile application. Future work will focus on implementing an on-disk storage back-end to reduce memory usage (at the expense of processing time), and abstractions to ease integration with cloud based file sharing platforms.

References

1. Souri, Alireza and Pashazadeh, Saeid and Habibizad Navin, Ahmad. Consistency of Data Replication Protocols in Database Systems: A Review. International Journal on Information Theory (IJIT). 3. 2014.
2. Vidal Martins, Esther Pacitti, Patrick Valduriez. Survey of data replication in P2P systems. RR-6083, INRIA. 2006.
3. Apache Software Foundation, CouchDB Replication Protocol, version 3, retrieved February 27. 2020.
4. Sen, Jaydip. Security and Privacy Issues in Cloud Computing. 2013.
5. Paul Voigt and Axel von dem Bussche. The EU General Data Protection Regulation (Gdpr): A Practical Guide (1st ed.). Springer Publishing Company, Incorporated. 2017.
6. CouchDB Team, CouchDB 2.0 Reference Manual, Samurai Media Limited. 2015.
7. Handschuh, Helena. SHA-0, SHA-1, SHA-2 (Secure Hash Algorithm).. In Encyclopedia of Cryptography and Security (2nd Ed.) , edited by Henk C. A. van Tilborg and Sushil Jajodia , Springer, 2011.
8. Ezéchiel, Katembo and Kant, Shri and Agarwal, Ruchi.. Analysis of database replication protocols. 2018.
9. Langdale, Geoff and Lemire, Daniel. Parsing Gigabytes of JSON per Second. 2019.
10. MacKenzie, D., Eggert, P., and Stallman, R.. Comparing and Merging Files with GNU diff and patch. Network Theory Ltd, 2002.
11. P. Deutsch. RFC1951: DEFLATE Compressed Data Format Specification version 1.3. RFC Editor, USA. 1996.