# OWebSync: A web middleware with state-based replicated data types and Merkle-trees for fluent synchronization of distributed web clients

#552

## Abstract

A lot of enterprise software services are adopting a fully web-based architecture for both internal line-of-business applications and for online customer-facing applications. Although wireless connections are becoming more ubiquitous and faster, mobile employees and customers are however not always connected. Nevertheless, continuous operation of the software services is expected.

This paper presents OWebSync: a web-based application middleware for the continuous synchronization of online web clients and web clients that have been offline for a longer time period. OWebSync implements a fine-grained data synchronization model and leverages upon Merkle-trees and convergent replicated data types to achieve the required performance both for online interactive clients, and for resynchronizing clients that have been offline.

In comparison with operation-based, generic middleware solutions, that are based on operational transformation or operation-based replicated data types, OWebSync scales better to tens of concurrent editors on a single document, and is also especially better in operating in and recovering from offline situations. As a state-based approach, OWebSync can achieve acceptable interactive performance with limited network overhead. This has been validated and evaluated in two industrial case studies.

***Keywords*** Data synchronization, Offline web applications.

**ACM Reference format:**
#552. 2018. OWebSync: A web middleware with state-based replicated data types and Merkle-trees for fluent synchronization of distributed web clients. In *Proceedings of EuroSys'19, Dresden, Germany, March 25–28, 2019,* 13 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Web applications have been the default architecture for many online software services, both for internal line-of-business applications such as CRM, HR, and billing, as well as for customer-facing software service delivery. Native fat clients are being abandoned in favor of browser-based applications. Browser-based service delivery fully abstracts the heterogeneity of the clients, and solves the deployment and maintenance problems that come with native applications. Nevertheless, native applications are still being used when rich and highly interactive GUIs are needed, or when applications need to function offline for a longer time. The former reason is disappearing more and more as HTML5 and JavaScript are becoming more and more powerful and even benefit from hardware acceleration. The latter reason should be disappearing too with the venue of Wifi, 4G and 5G ubiquitous wireless networks, even in tunnels and airplanes. However, in reality connectivity is often missing for several minutes to several hours. Mobile employees can be working in cellars or tunnels, and customers sometimes want to use your services while in an airplane.

A lot of native application-specific solutions and browser-plugins exist to tackle the problem in an ad-hoc solution. For example, a lot of Google web apps can be used in offline modus. However, there is no generic, fully web-based middleware solution that can be used by web applications to:

1. support fine-grained and concurrent updates by distributed web clients on local copies of shared data,
2. operate conflict-free in online and offline situations,
3. achieve continuous synchronization for online clients and fluent resynchronization for offline clients,
4. scale to tens (20-30) of online clients that concurrently edit a single shared document with interactive performance timings.

A lot of distributed NoSQL data systems, e.g. Amazon Dynamo [4], adopt synchronization based on Vector Clocks. This often lead to conflicts that need application-level resolving. Text-based versioning such as Git does not always guarantee consistent data structures after synchronization. Code, XML or JSON documents can end up malformed and often require user-level resolution. Operational Transformation [16] approaches are often used for real-time synchronization in interactive web applications (e.g. in Google Docs [21]) but are not resilient against message loss in case of long-time offline situations [8]. Commutative Conflict-free Replicated Data Types [15] as used in Legion [17], SMAC [5] and the JSON datatype of Kleppman [7] are also operation-based, but don't apply transformations to the operations. As such, the operations are commutative and can arrive and be applied in a different order. However, this technique also suffers when operations are lost.

State-based Convergent Conflict-free Replicated Data Types [15] (CRDTs) are resilient against message loss, but have often been considered as problematic with regard to the

amount of data that has to be transferred between all distributed entities, and therefore are considered less suited for interactive, collaborative applications. State-based CRDTs have been used in Riak [25] for example, to achieve background, asynchronous synchronization between back-end data centers internally.

In this paper we present OWebSync[1], a generic web middleware for browser-based applications, which supports concurrent updates on local copies of shared data between distributed web clients, and which supports continuous, fluent and fine-grained synchronization between online clients. The middleware supports fluent resynchronization when clients were offline for a longer time, e.g. in case of network failures. OWebSync leverages state-based CRDTs to support synchronization between clients and server. Merkle-trees [10] are used to enable more fluent synchronization of state-based CRDTs and limit the amount of data that has to be transferred. More specifically, OWebSync provides generic, reusable JSON [2] based data types that web applications can leverage upon to model their application data. These data types support fine-grained and conflict free synchronization of all items in the JSON documents.

Our comparative evaluation shows that all clients receive updates from other clients within the timespan of seconds, even when tens of clients are editing hundreds of shared objects in a single document. This makes it suitable for online, interactive and collaborative applications. Compared to operation-based middleware [26, 29], OWebSync scales better to tens of concurrent clients on a single document and is especially better in operating in and recovering from offline situations, even with silent network failure.

This paper is structured as follows. Section 2 provides two motivating case studies and then provides the rationale and more background on synchronization mechanisms such as CRDTs. Section 3 describes the generic, reusable JSON-based data types of OWebSync. Section 4 presents the deployment and runtime architecture of OWebSync. Section 5 compares and evaluates performance in online and offline situations. We discuss related work in Section 6 and then we conclude.

## 2 Motivation, Background and Approach

This section further explains the motivation of both the goal and approach of the OWebSync middleware. First we present two industrial case studies of online software services for both mobile employees and customers that often encounter long term offline situations. We then motivate our approach of state-based CRDTs with Merkle-trees and provide background information on Operational Transformation, Conflict-free Replicated Data Types and Merkle-trees.

[1] A try-out demo application on the middleware is available on an anonymous website (http://owebsync.cloudapp.net). One can open multiple Chrome browsers as concurrent clients. No personal identifiable information is gathered. No cookies are used.

***Case studies.*** We started from two industrial case studies from our applied research projects for the motivation, requirements analysis, and evaluation of the OWebSync middleware. The first case study is an online software service from eWorkforce. eWorkforce is a company that provides technicians to install network devices for different telecom operators at their customers' premises. The second company is eDesigners, who offers a web-based design environment for graphical templates that are applied to mass customer communication. This section will explain both case studies.

*eWorkforce* has two kinds of employees that use the online software service: the helpdesk operators at the office and the technicians on the road. The helpdesk operators accept customer calls, plan technical intervention jobs and assign them to a technician. The technicians can check their work plan on a mobile device and go from customer to customer. They want to see the details of the next job wherever they are, and need to be able to indicate which materials they used for a particular job. Since they are always on the road, a stable internet connection is not always available. Moreover, they often work in offline modus when they work in basements to install hardware. Booking all used materials as they are used is crucial for correct billing afterwards.

*eDesigners* offers a customer-facing multi-tenant web service to create, edit and apply graphical templates for mass communication based on the customer's company style. Templates can be edited by multiple users at the same time, even when they are offline. When two users edit the same document, a conflict occurs when the versions need to be merged. Edits that are independent of each other should both be applied to the template. For example, one edit can change the color of an object, another edit the size. When two users edit the same property of the same object, only one value can be saved. This should be resolved automatically as to not interrupt the user.

***Background, principles and approach.*** Next to the motivating case studies for our overall goal of OWebSync, we now describe our motivation and rationale of the approach. Therefore we first discuss the advantages and problems of state-of-the-art techniques such as Operational Transformation, operation-based CRDTs and state-based CRDTs.

*Operational Transformation (OT).* Operational Transformation [6] is a technique that is often used to synchronize concurrent edits on a shared document. For example, two clients can edit the text 'ABC' concurrently, where one client inserts '*' at position 1, and another client deletes the character at position 1. The former results in 'A*BC', the latter in 'AC'. To achieve the correct state ('A*C'), the first client needs to transform the incoming operation of the other client to a deletion at position 2. This means the operation needs to be transformed to the current local state. The problem is that the transformation of the incoming operations of other

clients on the local current state can get very complex, and that messages can get lost, or can arrive in the wrong order.

*Conflict-free Replicated Data Types (CRDTs).* CRDTs [15] are data structures that guarantee eventual consistency without the need for *explicit* conflict handling during synchronization by the application or user. Conflict-free thus means that conflicts are resolved automatically in a systematic and deterministic way, such that the application or user doesn't have to deal with conflicts. There are two kinds of CRDTs: operation-based (Commutative Replicated Data Types) and state-based (Convergent Replicated Data Types).

*Commutative Replicated Data Types (CmRDTs).* CmRDTs make use of operations to reach consistency, just like Operational Transformation (OT). But the operations in CmRDTs are commutative and can be applied in any order. This way, there is no central server needed to apply a transformation on the operations. As with OT, CmRDTs need a reliable message broadcast channel so that every message reaches every replica exactly once in the correct causal order [14].

*Convergent Replicated Data Types (CvRDTs).* CvRDTs are based on the state of the data type. Updates are propagated to other replicas by sending the whole state and merging the two CvRDTs. For this merge operation, there is a monotonic join semi-lattice defined over the states of a CvRDT. This means that there is a partial order defined over the possible states, and that there is a least-upper-bound operation between two states. The least-upper-bound is the smallest state that is larger or equal to both states according to the partial order. To merge two states, the least-upper-bound is computed and the result is the new state. CvRDTs don't require anything from the message channel, messages can get lost without a problem, since the whole state is always communicated. The main disadvantage is that the state can get quite large, and needs to be communicated every time.

*Delta-state CvRDTs.* $\delta$-CvRDTs [1] are a variant on state-based CRDTs with the advantage that in some cases only part of the state needs to be send for a correct synchronization. But for this to work, one need to keep some sort of history to find out which deltas need to be send and keep the causal order between the deltas.

*Merkle-trees.* Merkle-trees [10] or hash-trees are used to quickly compare two large data structures. Each item in a data structure is hashed, and then the hashes are combined in a hash on top, often in a binary way by combining two hashes from a lower level into a single hash at the higher level. This continues until the root of the tree is created with the top-level hash. Two data structures can now be compared starting from the two top-level hashes. If the root hashes match, the data structures are equal. Otherwise, the tree can be descended using the mismatching hashes to find the mismatching items.

To limit the overhead of messages with state exchanges between clients and server, we adopt Merkle-trees in the data structure to find the items that need to be synchronized. This data structure is discussed in Section 3. Together with other architectural performance tactics and implementation-level optimizations we can achieve fluent interactive synchronization. This is discussed in Section 4.

## 3 The OWebSync Data Model: Convergent replicated data types with Merkle-trees

In this section we describe the conceptual data model of OWebSync that web applications will need to use to ensure synchronization by the middleware. The data model is a Convergent Replicated Data Type (CvRDT) for the efficient replication of JSON data structures, and applies Merkle-trees to quickly find data changes.

The CvRDT consist of two other types of CvRDTs: a Last-Write-Wins Register (LWWRegister) [15] and an Observed-Removed Map (ORMap) [15] extended with a Merkle-tree. The LWWRegister is used to store values, such as strings, numbers and booleans, in the leaves of the tree. The ORMap is a recursive data structure that represents a map that can contain other ORMaps or LWWRegisters.

*Last-Write-Wins register (LWWRegister).* This data structure contains exactly one value (string, number or boolean) together with a timestamp of the last change of the value. The data structure supports three operations: reading the value, updating the value and merging a LWWRegister with another one. Each update operation also updates the timestamp. The merging operation will always result in the value and timestamp of the latest update. The other value is lost. This conflict resolution strategy essentially boils down to a simple last-write-wins strategy.

*Observed-Removed Map (ORMap).* The Observed-Removed Map is typically implemented using an Observed-Removed Set (ORSet) [15] which contains tuples with a key, a value and an identifier. An ORSet is constructed as in [15] with two grow-only sets. A grow-only set is also a CvRDT representing a set to which one can only add items. Such a set can easily be merged with other grow-only sets by simply creating a union. The ORSet contains a grow-only set for the added items (observed set) and a grow-only set for the removed items (removed set). Figure 1 presents the class diagram of the CRDTs that are used to represent the JSON datatype. This diagram also includes the internal CRDTs on which an ORMap is based. More specifically, ORMap extends ORSet, which extends a Two-Phase Set (2P-Set) [15]. The 2P-Set contains two Grow-Only Sets (G-Set).

We add an extra hash to the tuples in the ORMap to construct the Merkle-tree. When the child is a LWWRegister, the hash is simply the MD5-sum [13] of the value of that register. When the child is another ORMap, the hash of it is the combined hash of the hashes of all the children of that ORMap. This way, when one value in a register changes, all the hashes of the parents will also change, so that a change can be detected by only comparing the root hash.

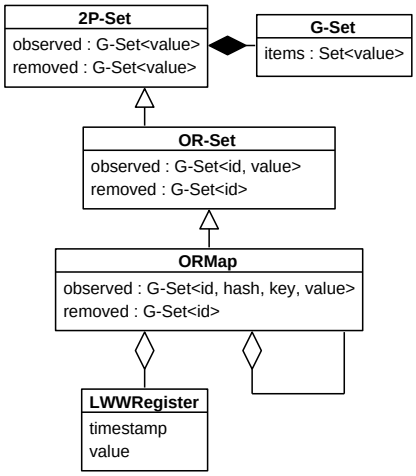



**Figure 1.** Class diagram of the CRDTs in OWebSync.

This data structure supports four operations: reading the value of a key, removing the value behind a key, updating the value of a key and merging the ORMap with another one. The read operation will be executed recursively to return a complete JSON object of the whole sub-tree behind the provided key when the child is also an ORMap, or will just return a primitive value if the child is a register. The remove operation will add the ID to the removed set. The update operation will update the hashes. To join two ORMaps, the union of the respective observed and removed set is taken. Then, the hashes are compared to check for changes in the children of the ORMap. When a mismatch is detected, the join is executed recursively to traverse the whole Merkle-tree below that key to detect all the changes. The conflict resolution of the ORMap boils down to an add-wins resolution. Concurrent edits to different keys can be made without a problem. Edits to the same key will be delegated to the child CRDT (either another ORMap or a register).

*Example.* As an example, we illustrate the conceptual representation of an application data object in the eDesigners case study, as well as the resulting CRDTs in the OWebSync data model. Figure 2 presents both the conceptual representation (Figure 2a) as well as two of the CRDTs (Figure 2b).

The latter represents the internal structure of two CRDTs that form the conceptual representation. First the key under which the CRDT is stored in a key-value store is listed, then the value of the CRDT. The first CRDT is an ORMap, the second a LWWRegister. For conciseness, only the "top" and the "left" properties are shown as children of "object36". In the real application all parameters as in Figure 2a are present.

*Considerations and discussion.* The current data model is of course best suited for semi-structured data that is produced and edited by concurrent users, like the data items in the case studies: graphical templates, a set of tasks or used materials for a task. This data model is less suited for applications like online banking or pure text editing.

In the current version of OWebSync, we only support a last-write wins strategy for updates in the leafs by authenticated and authorized users. Other tactics are also possible, for example a Multi-Value Register [15] where the application can build its own strategy. That would however require the implementation of application-specific resolution logic in the client. For example, in the eDesigners case study, concurrent edits of the color of the same object could then result in a merger of the two colors instead of overwriting the older color with the newer color.

In the current OWebSync data model, the removed-set of the ORMap keeps the IDs of all removed children eternally (so-called tombstones). As a result, the size of an ORMap can accumulate over time and performance will degrade. With a modest usage of deletion this will not be a large problem. Even when you delete a large sub-tree of several levels deep, only the ID of the root of the sub-tree is kept in the removed-set of the parent. All other data will be removed and is not needed anymore for correct synchronization. At the moment, OWebSync does not implement a solution for cleaning up tombstones, but one strategy could be to simply permanently remove all tombstones that are older than one month. We then expect that a client will not be offline for more than a month while performing concurrent edits. This can be enforced by letting the access control token of that user expire after a month of no usage.

Next to primitive values and maps, the JSON specification contains also the concept of ordered lists. This is currently not supported by OWebSync, and just like Swarm [28], we focused on the initial key data structures: last-write-wins registers and maps. Keeping a total numbered order, like lists do, is rarely needed and we did not need them for our two case studies. Unique IDs in a map are better suited in a distributed setting. In the case studies, the ordering of items in a set was also based on application-specific properties such as dates, times or other values, instead of an auto-incremented number of a list. Note that CvRDTs for ordered lists do exist, [15] and could be added in future work.

## 4 Web-based synchronization architecture

In this section we describe the deployment and execution architecture of the OWebSync middleware as well as the synchronization protocol. This middleware architecture is key to support the data model and synchronization model described in the previous section. We also elaborate on a set of key performance optimization tactics to achieve continuous, fluent synchronization for online interactive clients.

***Overall architecture.*** The middleware architecture is depicted in Figure 3 and consists of loosely-coupled client and server subsystems. First, the client-tier middleware API is fully implemented in JavaScript and completely runs in the browser without any need for add-ins or plugins. The server is a light-weight process listening for incoming web requests

```
{
    "drawings": {
        "drawing1": {
            "object36": {
                "fill": "#f00",
                "height": 50,
                "left": 50,
                "top": 100,
                "type": "rect",
                "width": 80
            }
        }
    }
}
```

**(a)** Conceptual representation of a single data object in the eDesigners case study.

```
* drawings.drawing1.object36:
      uuid: 14f545820826f04c634b408d06b8ba
      hash: 7319eae53558516daafac19183f2ee34
      observed:
          - uuid: 14f54581f8d5104c634b408d06b8ba
            hash: 65bdd1b610f629e54d05459c00523a2b
            key: "top"
          - uuid: 14f54581ffa3404c634b408d06b8ba
            hash: 675078769412850854849840 80f5951e
            key: "left"
          ...
      removed:
* drawings.drawing1.object36.top:
      uuid: 14f54581f8d5104c634b408d06b8ba
      hash: 65bdd1b610f629e54d05459c00523a2b
      value: "100"
      timestamp: 789778800000
```

**(b)** Structure of two CRDTs that represent "object36" and the property "top".

**Figure 2.** Datastructure of the eDesigners case study.

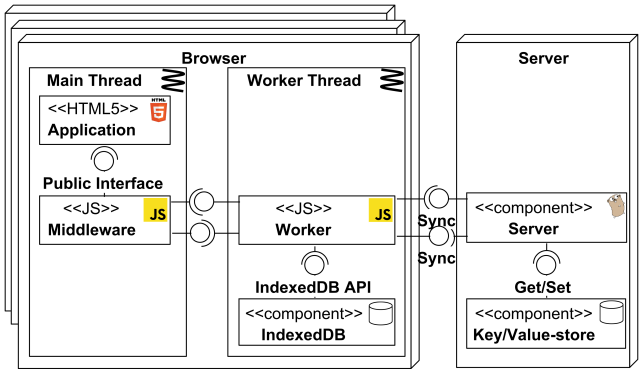



**Figure 3.** Overall architecture of the OWebSync middleware

and storing all shared data. The server is only responsible for data synchronization and does not run application logic. However, access control on the data is also supported and enforced at the server. Both the clients and server have a key-value store to make data persistent on disk. The many clients and server communicate using only web-based HTTP traffic and web sockets. All communication messages between client and server are sent and received using asynchronous workers inside the client and server subsystems. We first further elaborate on the client-tier subsystem with the public middleware API for applications, and then describe the client-server communication protocol for synchronization in detail.

***Client-tier middleware and API.*** The public programming API of the middleware is located completely at the client-tier. Web applications are developed as client-side JavaScript applications that use the following API:

- GET(path): Returns a JavaScript Object or primitive value for a given path.
- LISTEN(path, callback): Similar to a GET, but every time the value changes, the callback is executed.
- SET(path, value): Create or update a value at a given path.

The OWebSync middleware is loaded as a JavaScript library in the client and the middleware is then available in the global scope of the web page. One can then load data and edit data using typical JavaScript paths. An example from the eDesigners case study:

```
let d1 = await OWebSync.get("drawings.drawing1");
d1.object36.color = "#f00";
OWebSync.set("drawings.drawing1", d1);
```

***Synchronization protocol.*** The synchronization protocol between client and server consists of three key messages, that the client can send to the server and vice versa:

1. GET(path, hash): the receiver returns the CRDT at a given path if the hash is different from its own CRDT at the given path.
2. PUSH (path, CRDT): the sender sends the CRDT data structure at a given path and the receiver will merge it at the given path.
3. REMOVE(path, uuid): removes the CRDT at a given path if the unique identifier (uuid) of the value is matching the given uuid. As such, a newer value with a different uuid will not be deleted.

The protocol is initiated by a client, which will traverse the Merkle-tree of the CRDTs. The synchronization starts with the highest CRDT in the tree. The client will send a GET message to the server with the given path and hash value of the CRDT. If the server concludes that the hash of the path matches the client's hash, the synchronization stops. All data is consistent at that time.

If the hash does not match, the server returns a PUSH message with the CRDT that is located at the PATH requested by the client. The client has to merge the new CRDT with the CRDT at its requested location. This merger process at the client might detect conflicting children in the tree by comparing the hashes. The client will then PUSH that child to the server with the CRDT of the client. The server then needs to merge this CRDT. If a child does not exist yet, an empty child is created and a GET message is sent.

The process continues by traversing the tree and exchanging PUSH and GET messages until the leaf of the tree is reached. The CRDT in this leaf is a register and can be merged immediately. All parents of this leaf are now updated such that finally the top-level hash of client and server match. If the top-level hashes do not match, other updates have been done in the meantime, and the process is repeated.

If during a merger process, a child seems to be removed at one side, but not at the other side, a REMOVE message is sent to the other party so that it can remove that value and add the ID to the removed set of the correct ORMap. Alternatively, this additional third message type of REMOVE could be avoided if a PUSH of the parent would be sent instead. However, the push of a parent with many children would cause a serious overhead compared to a REMOVE message with only a path and uuid.

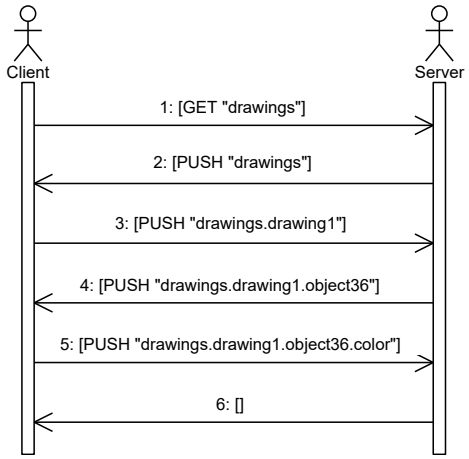



**Figure 4.** Synchronization protocol when the client has made an update. With every PUSH message, the respective CRDT is send. E.g. for message 4, the first CRDT in Figure 2b is send.

Figure 4 shows an example for the eDesigners case study where the client has changed the color of an object. If the client had made multiple changes, e.g. he also changed the height, the start of the synchronization protocol would be the same, except that the height will also be included in message five.

***Performance optimization tactics.*** The main optimization tactic to achieve fluent synchronization for interactive applications is the reduction of network traffic by the Merkle-trees. However, there are additional tactics needed to further improve synchronization time. The protocol discussed above leads to many messages between clients and server. To reduce the chattiness and overhead of the synchronization protocol between the many clients and server, different optimization tactics are applied by the client and the server.

*Message batching.* In the basic protocol explained above, all messages are sent to the other party as soon as a mismatch of a hash in the Merkle-tree is detected. This leads to lots of small messages (GET, PUSH, and REMOVE) being sent out, and as a consequence, a lot of messages are coming in while still doing the first synchronization. This results in a lot of duplicated messages and doing a lot of duplicated work on subtrees, since the root hash will only be up-to-date when the bottom of tree is correctly synchronized, and not when another synchronization round is already busy somewhere halfway in the tree. To solve this problem, all messages are grouped in a list and are sent out in batch after a full pass of a whole level of the tree has occurred. At the other side, the messages are processed one by one, and all resulting messages are again grouped in a list, and then send out after the incoming batch was fully iterated. If no further messages are resulting from the processing of a batch, an empty list is sent to the other party. This ends the synchronization. As a result, a lot less messages are sent between a client and server, and only one synchronization per client is occurring at the same time, resulting in no duplicated messages and no duplicated work on subtrees.

*Parallel processing of message batches.* Message batching eliminated the parallel processing of many small messages that could lead to a lot of duplicated work on subtrees. However, because it processes the messages in a batch one by one, there is no more parallel processing at all and the synchronization time increases significantly. To solve this problem, the messages in one batch are processed in parallel.

## 5 Performance evaluation

The performance evaluation will focus on situations where all clients are continuously online, as well as on situations where the network is interrupted. For online situations, we are especially interested in the time it takes to distribute and apply an update to all other clients that are editing the same data. For the offline situation we are especially interested in

how long it takes for all clients to get back in sync with each other after the network disruption.

The performance evaluation in this paper is performed using the eDesigners case study, as this scenario has the largest set of shared data and objects between users. The eWorkforce case study has less shared data with less concurrent updates as technicians typically work on their own data island and the data contains less objects with less frequent changes. To compare performance, we implemented the eDesigners case study three times on three representative JavaScript technologies for web-based data synchronization: our OWebSync platform, which uses state-based CRDTs with Merkle-trees, Yjs [29] which uses operation-based CRDTs, and ShareDB [26] which makes use of Operational Transformation. Both Yjs (773 Github stars) and ShareDB (1964 Github stars) are widely-used open source technologies that are available on GitHub.

***Benchmark setup.*** Both the clients and the server are deployed as separate Docker containers on a set of VMs in our OpenStack private cloud. A VM has 4 GB of RAM and 4 vCPUs and can hold up to 3 client containers. A client container contains a browser which loads the client-side OWebSync middleware from the server. The middleware server is deployed on a separate VM. The monitoring server that captures all performance data is also deployed on a separate VM. Pumba [24] is used to artificially increase the latency between the containers to an average of 100 ms with 50 ms jitter which resembles the latency of a bad 4G network. To have a fair comparison, all three technologies (OWebSync, Yjs and ShareDB) use web sockets with identical time-outs (10 seconds) to detect silent network failure.

Our evaluation contains 36 benchmarks: 6 benchmarks to be executed by each of the 3 technologies, in both a continuous online setting as well as in a disconnected situation. These 6 benchmarks vary in number of clients and data size: 8, 16, or 24 clients are performing continuous concurrent updates on 100 or 1000 objects in a single shared document. Each client performs a random write on a shared object every second. We thus use at most 24 clients, which are editing the same document concurrently. In comparison, Google Docs (the most popular collaborative editing system today) supports a maximum of 100 concurrent users according to Google [21] itself. But in practice, latency starts to increase significantly when the number of users exceeds 10 [3]. Our performance results show the same problem for the other web-based synchronization middleware in our comparison.

In our performance evaluation, one iteration of a benchmark takes about 11 minutes. The first three minutes are used to populate the database, to perform the initial synchronization, and to execute a minute of warm-up. Then we measure the performance of 8 minutes of continuous updates. Finally, we wait until all updates are synchronized with a maximum of 15 minutes.

To ensure stability and consistency of the benchmark results on our private cloud, we first validated the performance results by repeating the benchmark 100 times. This resulted in about 13 hours of recorded data to validate consistency of the performance metrics. In order to execute all 36 benchmarks for this paper, we reduced the number of iterations to 10. This showed the same consistency and stability of the performance results. The 10 iterations take in total 110 minutes and provide us with 80 minutes of data[2] for each benchmark (initialization time and warm up period excluded) in which each client makes one update every second.

***Performance of continuous online updates.*** The following performance measurements quantify the statistical division of the time it takes to synchronize a single update to all other clients in the case of a continuous online situation.

First of all, Yjs failed to synchronize all updates within the 15 minute waiting time for the benchmarks with 24 clients, as well as for the benchmark with 16 clients and 1000 objects. The success rate of the actually synchronized updates was between 13% and 37%. ShareDB and OWebSync did not fail to synchronize all updates.

The synchronization times of the succeeded updates are illustrated in Figure 5 and Table 1. Figure 5 contains the boxplots of all update times in the 10 iterations of each of the 18 benchmarks in the online setting. In order to compare results of the same order of magnitude, as well as results in different orders of magnitude, we opted for a logarithmic Y axis. Table 1 contains the synchronization times (aggregated for all 10 iterations of a benchmark) and their standard deviation over the 10 iterations. The presented results start from the 50th percentile (i.e.half of the succeeded updates are synchronized within the given seconds) to 100% (i.e. all succeeded updates are synchronized within the given seconds).

*Analysis of the results.* For the benchmark with 8 clients and 100 objects, both Yjs and ShareDB synchronize the updates faster than OWebSync. 99.9% is below 0.75 seconds. OWebSync needs about 2.56 seconds for synchronizing the 99.9th percentile, and 2.68 for synchronizing 100% of the updates. This is when Yjs and ShareDB first start to struggle. 100% synchronization requires respectively 22.58 and 103.24 seconds, but also includes a huge standard deviation to achieve this. For OWebSync this deviation on the results is much less (0.11). For the benchmark with 24 clients and 1000 objects OWebSync becomes the fastest with a maximum of only 6.21 seconds for the time to synchronize all updates. Yjs and ShareDB require tens of seconds for the 50th percentile and even hundreds of seconds for the 90th percentile.

We can conclude that the synchronization time of OWebSync only slightly increases when scaling up to 1000 objects and 24 clients. However, the update times for Yjs and

[2]The raw logs of all 36 benchmarks, and the graphical analysis in boxplots of each iteration, are available on an anonymous Azure storage account for verification: https://owebsyncdata.blob.core.windows.net/logs/data.zip

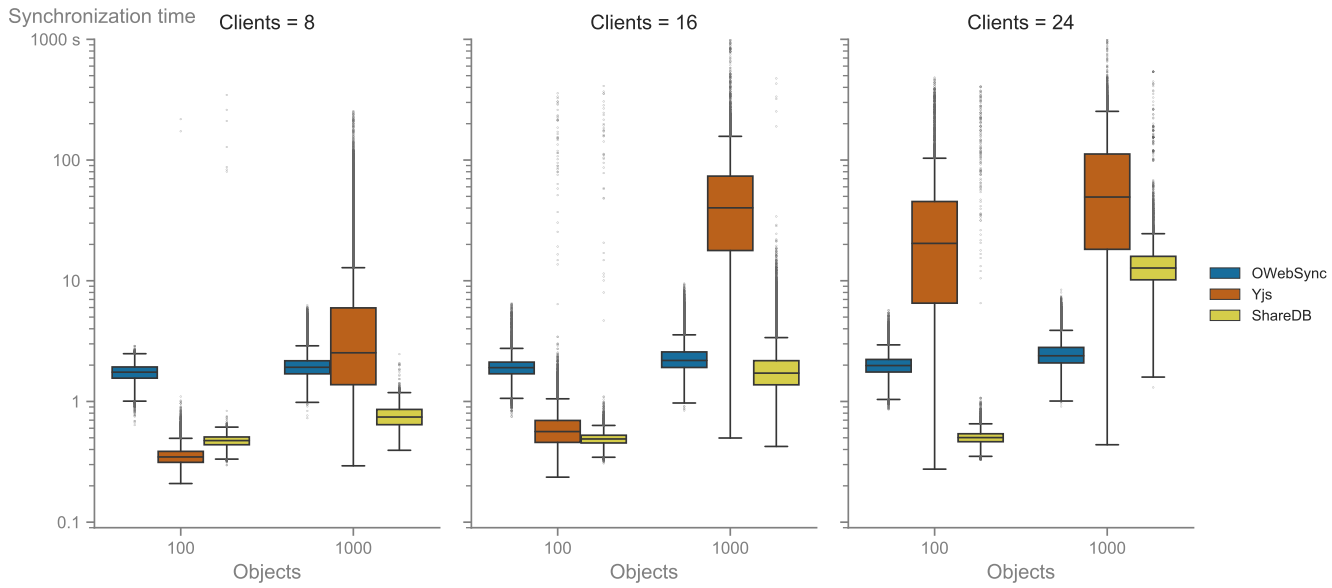



**Figure 5.** Aggregated boxplots containing the times to achieve full synchronization to all clients. Each boxplot contains all 10 iterations for each of the 18 benchmarks in the fully online situation. In order to compare technologies that have results of the same order of magnitude, as well as results in different orders of magnitude, we opted for a logarithmic Y axis.

| | 100 objects | | | 1000 objects | | |
|---|---|---|---|---|---|---|
| | 8 clients | 16 clients | 24 clients | 8 clients | 16 clients | 24 clients |
| *OWebsync* | | | | | | |
| 50% | 1.75 ± 0.01 | 1.91 ± 0.02 | 1.98 ± 0.03 | 1.92 ± 0.04 | 2.23 ± 0.24 | 2.39 ± 0.08 |
| 90% | 2.08 ± 0.01 | 2.32 ± 0.04 | 2.48 ± 0.07 | 2.56 ± 0.20 | 3.07 ± 0.50 | 3.31 ± 0.17 |
| 95% | 2.16 ± 0.01 | 2.48 ± 0.09 | 2.67 ± 0.10 | 2.92 ± 0.35 | 3.46 ± 0.63 | 3.70 ± 0.24 |
| 99% | 2.33 ± 0.03 | 3.09 ± 0.61 | 3.17 ± 0.27 | 3.97 ± 0.64 | 4.44 ± 1.08 | 4.60 ± 0.47 |
| 99.9% | 2.56 ± 0.06 | 3.88 ± 1.28 | 3.88 ± 0.54 | 5.02 ± 0.71 | 5.50 ± 1.38 | 5.64 ± 0.75 |
| 100% | 2.68 ± 0.11 | 4.22 ± 1.37 | 4.30 ± 0.59 | 5.34 ± 0.79 | 5.97 ± 1.39 | 6.21 ± 1.03 |
| *Yjs* | | | | | | |
| 50% | 0.35 ± 0.01 | 0.57 ± 0.01 | 31.01 ± 6.51 | 3.38 ± 1.88 | 50.43 ± 12.19 | 74.89 ± 26.98 |
| 90% | 0.44 ± 0.02 | 0.85 ± 0.02 | 82.39 ± 12.14 | 18.83 ± 10.76 | 116.32 ± 27.81 | 227.64 ± 114.30 |
| 95% | 0.49 ± 0.02 | 0.97 ± 0.03 | 96.75 ± 10.82 | 37.39 ± 24.96 | 148.59 ± 35.30 | 302.73 ± 103.52 |
| 99% | 0.58 ± 0.03 | 1.30 ± 0.06 | 157.74 ± 42.73 | 71.61 ± 52.21 | 291.72 ± 105.64 | 509.70 ± 218.52 |
| 99.9% | 0.75 ± 0.08 | 2.18 ± 0.37 | 327.62 ± 24.81 | 89.80 ± 63.14 | 595.14 ± 255.30 | 641.43 ± 245.58 |
| 100% | 22.58 ± 65.20 | 214.64 ± 108.46 | 386.38 ± 38.35 | 93.11 ± 64.98 | 773.57 ± 257.50 | 706.56 ± 219.51 |
| *ShareDB* | | | | | | |
| 50% | 0.47 ± 0.00 | 0.49 ± 0.00 | 0.50 ± 0.00 | 0.74 ± 0.01 | 1.70 ± 0.04 | 13.81 ± 3.45 |
| 90% | 0.54 ± 0.00 | 0.56 ± 0.00 | 0.57 ± 0.00 | 0.96 ± 0.02 | 2.83 ± 0.04 | 46.04 ± 55.22 |
| 95% | 0.56 ± 0.00 | 0.58 ± 0.01 | 0.59 ± 0.01 | 1.01 ± 0.01 | 3.81 ± 0.09 | 105.18 ± 155.23 |
| 99% | 0.58 ± 0.01 | 0.62 ± 0.02 | 0.64 ± 0.01 | 1.10 ± 0.01 | 6.92 ± 0.35 | 129.86 ± 155.56 |
| 99.9% | 0.62 ± 0.03 | 0.78 ± 0.12 | 37.94 ± 34.66 | 1.22 ± 0.03 | 13.07 ± 1.16 | 199.65 ± 160.70 |
| 100% | 103.24 ± 122.04 | 222.05 ± 101.09 | 295.25 ± 69.22 | 1.31 ± 0.06 | 20.88 ± 5.17 | 297.72 ± 166.33 |

**Table 1.** Synchronization times and their standard deviation for the succeeded updates in all 10 iterations of each benchmark, starting from the 50% percentile (i.e. half of the updates are fully synchronized after the given seconds) to 100% (i.e. all succeeded updates are fully synchronized after the given seconds).

shareDB increase significantly when going beyond 8 clients or when synchronizing 1000 objects. For the Yjs and ShareDB benchmarks with 24 clients and 1000 objects, the results start to increase and fluctuate more and more as these technologies start to struggle with the scale of the benchmark.

All the OWebSync benchmarks show more consistent results during their 10 iterations. This is clearly visible in Table 1 with the distribution of the synchronization times of OWebSync. The standard deviation on the synchronization times is clearly limited in all benchmarks. This applies to all percentiles. For the OWebSync benchmark with 24 clients and 1000 objects, the separate boxplots of the 10 iterations actually show consistent medians, third quarters, maxima and outliers (Figure 6).

The trade-off for this scalable, fluent synchronization, is that OWebSync has the largest network usage of all tested technologies (Figure 7). The usage of Merkle-trees reduced the network usage with about a factor 8 in the worst case. Even in the benchmark with 24 clients and 1000 objects, the used bandwidth is less then 800 kbit/s per client. This is much less than the available bandwidth which is on average 27 Mbit/s on a mobile network in the US [27]. In this same benchmark, the server consumes about 20 Mbit/s, which is acceptable for a typical data center.

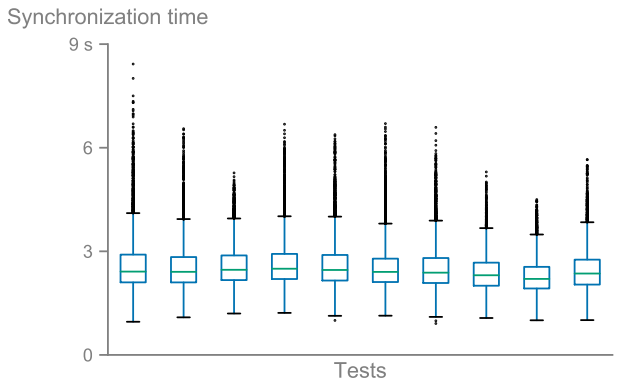



**Figure 6.** Boxplots of 10 iterations of the OWebSync benchmark with 1000 objects and 24 clients.

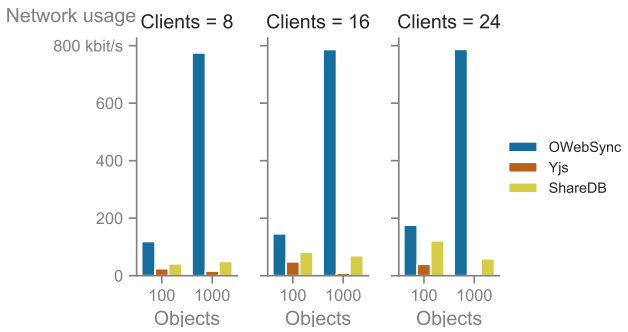



**Figure 7.** Network usage per client for each benchmark.

*Interpretation and discussion.* For interactive web applications, usability guidelines [12] state that a direct interaction should occur within 0.1 seconds. Remote response times should typically be 1 to 2 seconds in average. 3 to 5 seconds is the absolute maximum before users are annoyed. The user is often leaving the web application after 10 seconds of waiting time. We start from these numbers to assess the update propagation time between users in a collaborative interactive online application with continuous updates. We are interested in the waiting time for a user to receive an update from another online user. These numbers should be achieved not only for the average user (the mean synchronization time) but also for the 99th percentile (i.e. *most of the users* [4]).

Table 1 shows that the average result for the OWebSync benchmark with 24 clients and 1000 objects is around 2 to 2.5 seconds. The average synchronization time of the 99th percentile is at the border line of annoyance with 4.60s. Yjs and ShareDB operate with sub-second synchronization times when sharing 100 objects between 8 writers. When the number of objects and writers increases, the synchronization time raises to tens of seconds for the 50th percentile, and hundreds of seconds for the 99th percentile. This is in line with the observations of [3] for Google Docs, which uses the same approach as ShareDB (Operational Transformation).

***Performance in disconnected scenarios.*** We now present the performance analysis for the case when the network between one client and the server goes down. In these benchmarks, we have an analogous benchmark setup. However, during the 11 minute execution, we start dropping all messages after 3 minutes for 1 minute using Pumba [24]. Again, Yjs only succeeds in synchronizing 10% to 23% of the updates for the benchmarks with 24 clients.

*Analysis of the results.* In summary, the resynchronization time of OWebSync in case of network failure is between 1 and 4 seconds, which is acceptable for interactive online web applications. The boxplots of these benchmarks (Figure 8 and Table 2) show that OWebSync can synchronize all missed updates faster than Yjs or ShareDB. These technologies need tens or hundreds of seconds to process the updates. This is due to their operation-based nature. Yjs and ShareDB need to replay all missed operations on the client that was offline. OWebSync only needs to merge the new state, which it does in exactly the same way as if the failure never happened.

*Timeline analysis of the benchmarks.* The overall impact on all clients is minimal as can be seen in the timeline graph in Figure 9 for the benchmark with 8 clients and 100 objects. This graph shows the total synchronization time on the Y-axis, for each update done at a given moment during the benchmark timeline (X-axis). The first two minutes show consistent synchronization times for all three technologies. In this benchmark, OWebSync is the slowest as was the case for the online situation. After those two minutes, the network of one client is disrupted. This is 3 minutes after

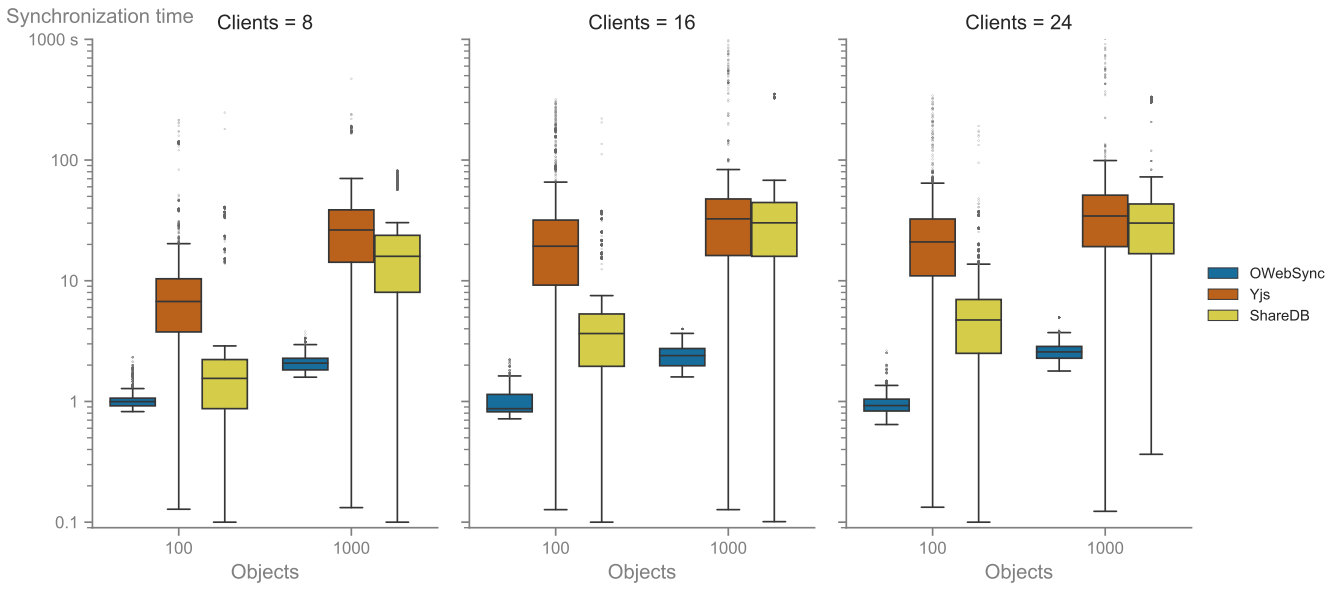
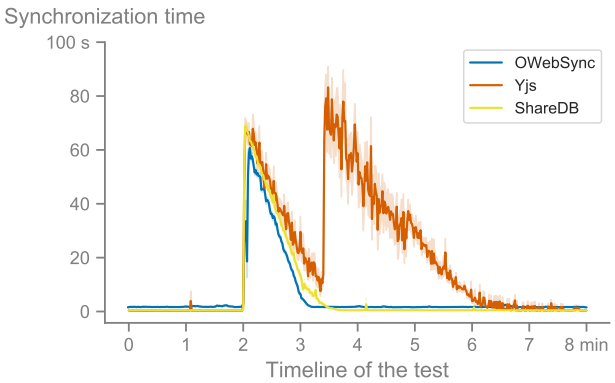


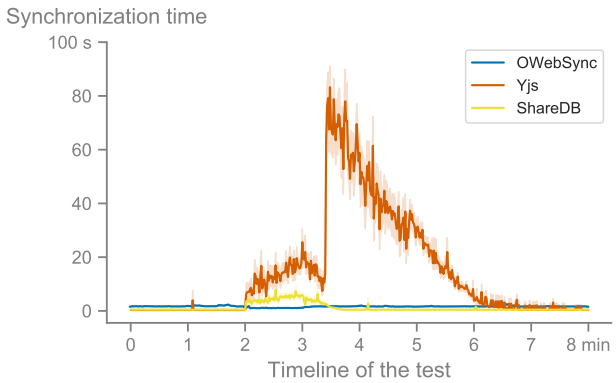

**Figure 8.** Boxplots of the time it takes for an update done during the failure scenario to be received by all clients.

| | 100 objects | | | 1000 objects | | |
|---|---|---|---|---|---|---|
| | 8 clients | 16 clients | 24 clients | 8 clients | 16 clients | 24 clients |
| *OWebsync* | | | | | | |
| 50% | 1.02 ± 0.14 | 0.96 ± 0.16 | 0.96 ± 0.19 | 2.04 ± 0.24 | 2.33 ± 0.41 | 2.60 ± 0.43 |
| 90% | 1.40 ± 0.35 | 1.00 ± 0.17 | 0.99 ± 0.19 | 2.54 ± 0.29 | 2.45 ± 0.42 | 2.81 ± 0.44 |
| 95% | 1.71 ± 0.15 | 1.57 ± 0.37 | 1.15 ± 0.34 | 2.76 ± 0.31 | 3.16 ± 0.44 | 2.83 ± 0.42 |
| 99% | 1.82 ± 0.23 | 1.78 ± 0.27 | 1.83 ± 0.37 | 2.88 ± 0.25 | 3.32 ± 0.38 | 3.43 ± 0.58 |
| 99.9% | 1.86 ± 0.21 | 1.86 ± 0.22 | 1.86 ± 0.39 | 2.99 ± 0.28 | 3.32 ± 0.37 | 3.50 ± 0.53 |
| 100% | 1.86 ± 0.21 | 1.86 ± 0.22 | 1.86 ± 0.40 | 3.00 ± 0.30 | 3.32 ± 0.37 | 3.56 ± 0.53 |
| *Yjs* | | | | | | |
| 50% | 10.82 ± 8.97 | 41.31 ± 32.28 | 20.67 ± 2.69 | 26.49 ± 1.91 | 31.80 ± 9.46 | 33.97 ± 3.73 |
| 90% | 43.79 ± 49.14 | 82.55 ± 61.10 | 41.42 ± 7.67 | 46.47 ± 2.55 | 52.82 ± 14.72 | 61.92 ± 7.60 |
| 95% | 47.28 ± 50.06 | 131.57 ± 71.05 | 58.36 ± 11.57 | 77.61 ± 50.76 | 112.43 ± 113.54 | 115.17 ± 123.67 |
| 99% | 89.76 ± 72.86 | 178.91 ± 90.47 | 134.72 ± 66.76 | 154.39 ± 46.72 | 414.49 ± 250.11 | 442.50 ± 184.87 |
| 99.9% | 99.74 ± 80.70 | 211.78 ± 86.39 | 246.19 ± 55.59 | 185.87 ± 79.13 | 645.16 ± 291.23 | 738.65 ± 183.96 |
| 100% | 102.93 ± 83.96 | 242.26 ± 76.40 | 266.94 ± 58.20 | 201.46 ± 108.08 | 674.31 ± 294.22 | 771.66 ± 186.74 |
| *ShareDB* | | | | | | |
| 50% | 1.56 ± 0.12 | 3.70 ± 0.34 | 4.75 ± 0.25 | 16.00 ± 0.89 | 30.20 ± 1.19 | 30.05 ± 0.74 |
| 90% | 20.73 ± 14.40 | 6.32 ± 0.49 | 8.27 ± 0.35 | 65.99 ± 6.81 | 53.28 ± 1.19 | 51.65 ± 1.11 |
| 95% | 31.57 ± 8.80 | 17.18 ± 10.11 | 8.70 ± 0.35 | 67.67 ± 6.79 | 167.03 ± 135.97 | 319.09 ± 8.77 |
| 99% | 31.77 ± 8.77 | 26.08 ± 8.55 | 23.36 ± 9.45 | 68.98 ± 6.79 | 169.51 ± 136.06 | 321.68 ± 8.77 |
| 99.9% | 50.80 ± 41.58 | 31.01 ± 9.91 | 36.69 ± 22.56 | 69.30 ± 6.83 | 171.14 ± 135.16 | 322.22 ± 8.77 |
| 100% | 67.41 ± 75.16 | 82.51 ± 76.05 | 117.07 ± 64.98 | 69.33 ± 6.83 | 171.98 ± 134.54 | 322.27 ± 8.77 |

**Table 2.** Synchronization times for missed updates during the network disruption, starting from 50th percentile (i.e. half of the missed updates are fully synchronized between all clients after the given seconds) to 100% (i.e. all missed updates are fully synchronized after the given seconds). The synchronization time does not include offline time.

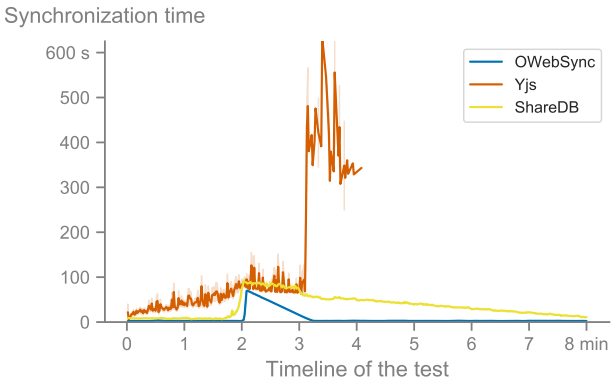



**(a)** The total time to synchronize updates, including the time offline. The peak at 120 seconds is due to the 1 minute failure.

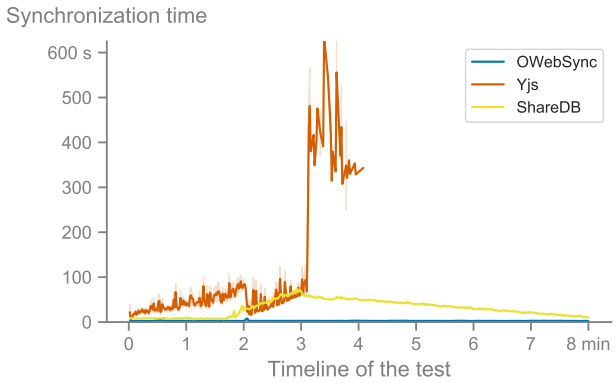



**(b)** The time to synchronize updates after the network failure, thus without the time during the failure taken into account.

**Figure 9.** Evolution of the time to synchronize updates in the benchmark with 8 clients, 100 objects and network failure.




**(a)** The total time it takes to synchronize updates, including the time offline. The peak at 120 seconds is due to the 1 minute failure.




**(b)** The time it takes to synchronize updates, not including the time offline.

**Figure 10.** Evolution of the time to synchronize updates for the benchmark with 24 clients, 1000 objects and network failure.

the start of the benchmark, but the first minute is used as warm-up period and is not shown in the graphs. One minute later, the network is repaired and the synchronization times drop as full synchronization is possible again. OWebSync quickly returns to the same performance as before the failure. ShareDB also achieves the original performance again, but takes a bit more time to achieve this. Yjs on the other hand will block the synchronization of new updates to first synchronize missed updates from during the failure, and only then resumes normal synchronization. In the benchmark with 24 clients and 1000 objects (Figure 10), OWebSync still quickly returns to the same performance as before the failure. However, ShareDB now takes much longer to achieve this. Yjs clients can not longer handle the combination of ongoing updates and delayed updates, and start failing to synchronize after 4 minutes due to client side load.

OWebSync is only slightly influenced by a network disruption, as can be seen in Figure 9b and Figure 10b which show the resynchronization times without the offline time during the failure. This means that for an update done 20 seconds before the end of the failure, and which got synchronized 22 seconds later, the resynchronization time is 2 seconds.

For OWebSync, this graph stays mostly flat for updates done during the failure. For the benchmark with 8 clients, it even drops to values lower than during normal online synchronization, because synchronization after a failure is faster. This is due to the fact that the synchronization starts immediately after the client notices that the failure is over, instead of waiting until the previous synchronization is done like in the normal scenario. The other technologies, Yjs and ShareDB, have an increased synchronization time for updates done during the failure. In the benchmark with 24 clients and 1000 objects, these operation-based technologies also still perform slower for updates done after the failure.

*Summary.* Our evaluation shows that the operation-based approaches work well in continuous online situations with a limited number of users. However, when network disruptions occur, or when the number of users scales up, these

technologies can not achieve acceptable performance and need tens or hundreds of seconds to achieve synchronization. The state-based approach of OWebSync can achieve much better performance in the order of seconds, which is still acceptable for interactive web applications.

## 6 Related work

The related work consists of three types of work: 1) concepts and techniques such as CRDTs and Operational Transformation, 2) distributed data systems such as Dynamo and Cassandra, as well as 3) synchronization frameworks for clients such as PouchDB, Swarm.js, Yjs and ShareDB. The concepts and techniques were discussed in Section 2. In this section we focus on the distributed data systems and on synchronization frameworks.

*Distributed data systems and NoSQL.* With the venue of NoSQL systems, a lot of new storage solutions have appeared that offer *eventual consistency* between different distributed nodes, within or even across data centers. Their focus is often to provide availability of read and write operations over strong consistency in the context of network partitions. These are typical systems of which the replicated data copies are stored on multiple servers across or within data centers, and that support concurrent updates on the different data copies, even when the nodes are disconnected.

Dynamo [4] is a highly-available key-value store at Amazon. The focus is to support high-availability of write operations. Applications should always be able to write on the local copy. In case conflicts occur between different versions (in the case of network partitions), the reconciliation occurs when the data item is read later. Using syntactic reconciliation, Dynamo can resolve the conflict between a newer version and an older version if a newer version is clearly derived from the older version. In case concurrent writes occurred, Dynamo relies on the application to merge the two versions (semantic reconciliation). Dynamo can thus not merge two versions of complex objects that are stored as values in the key-value store.

Based on the original Dynamo paper, a lot of other open-source NoSQL systems have been developed for structured or semi-structured data. Cassandra [9, 19] supports fine-grained versioning of cells in a wide-column store. It therefore uses timestamps for each row-column cell, and adopts a last-write-wins strategy to join two cells. CouchDB [20] and MongoDB [22] focus on semi-structured document storage, typically in a JSON format. CouchDB offers coarse-grained versioning per document and stores multiple versions of the document. Applications need to resolve the conflicts between the versions. Moreover, it also does not support fine-grained conflict detection or merging within two JSON documents. Riak [25] is a server-side key-value store like Amazon Dynamo, but also supports more fine-grained data structures such as state-based CRDTs (registers, counters, sets and maps). It does not support client-side data replicas, Merkle-trees for synchronization, or long-term offline usage. Antidote [18] is a research project to develop a geo-replicated database over world-wide data centers. It adopts operation-based commutative CRDTs for highly-available transactions. It supports partial replication but assumes continuous online connections as the default operational situation.

*Client-tier JavaScript-libraries for synchronization.* A lot of JavaScript frameworks have appeared to enable synchronization between web browsers and server-side data systems. PouchDB [23] is a client-side JS library that can replicate data from and to a CouchDB server. Local data copies are stored in the browser for offline usage. PouchDB only supports conflict detection and resolution at the coarse-grained level of a whole document. ShareDB [26] is a client-server framework to synchronize JSON documents and adopts Operational Transformation as synchronization technique between the different local copies. ShareDB can thus not be used in extended offline situations. In case of short network disruptions it can store the operations on the data in memory and resend them when the connection restores. The offline operations are lost when the browser session is closed. Yjs [11, 29] is a JavaScript Framework for synchronizing structured data and supports maps, arrays, XML and text documents. All data types also use operation-based CRDTs for synchronization. Swarm.js [28] is a JavaScript client library for the Swarm database and uses a Replicated Object Notation (RON). RON is based on operation-based CRDTs with a partially ordered log for synchronization after offline situations. It currently only supports sets and basic values like string and int. Swarm.js also focuses on peer-to-peer architectures like chat applications and decentralized CDNs, while OWebSync focuses on client-server line-of-business applications.

## 7 Conclusion

This paper presented a web middleware that supports the fluent synchronization of both online and offline clients that are concurrently editing shared data sets.

Our OWebSync middleware implements a data model that combines state-based CRDTs with specific enhancements based on Merkle-trees. Due to the enhancements in our data model and performance tactics in our supporting middleware architecture, we were able to achieve fluent and fine-grained synchronization for online interactive web applications with continuous concurrent updates.

Our comparative evaluation shows that the operation-based approaches can not achieve acceptable performance in case of network disruptions or larger scale settings, and need tens or hundreds of seconds to achieve synchronization. The state-based approach of OWebSync can achieve better performance in the order of seconds, which is still acceptable for interactive web applications.

# References


[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2015. *Efficient State-Based CRDTs by Delta-Mutation.* Springer International Publishing, 62–76. https://doi.org/10.1007/978-3-319-26850-7_5

[2] Tim Bray. 2014. *The javascript object notation (json) data interchange format.* RFC 7158. IETF. https://www.rfc-editor.org/rfc/rfc7158.txt

[3] Quang-Vinh Dang and Claudia-Lavinia Ignat. 2016. Performance of real-time collaborative editors at large scale: User perspective. In *Internet of People Workshop, 2016 IFIP Networking Conference (Proceedings of 2016 IFIP Networking Conference, Networking 2016 and Workshops).* IFIP, Vienna, Austria, 548–553. https://doi.org/10.1109/IFIPNetworking.2016.7497258

[4] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, Vol. 41(6). ACM, ACM, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[5] Jacob Eberhardt, Dominik Ernst, and David Bermbach. 2016. *SMAC: State Management for Geo-Distributed Containers.* Technical Report. Technische Universitaet Berlin.

[6] C. A. Ellis and S. J. Gibbs. 1989. Concurrency Control in Groupware Systems. *SIGMOD Rec.* 18, 2 (June 1989), 399–407. https://doi.org/10.1145/66926.66963

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

[8] Santosh Kumawat and Ajay Khunteta. 2010. A survey on operational transformation algorithms: Challenges, issues and achievements. *International Journal of Computer Applications* 3, 12 (July 2010), 30–38. https://doi.org/10.5120/787-1115

[9] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[10] Ralf Merkle. 1982. Method of providing digital signatures. (1982). US patent 4309569. The Board Of Trustees Of The Leland Stanford Junior University.

[11] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2015. Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types. In *Engineering the Web in the Big Data Era.* Springer International Publishing, Cham, 675–678.

[12] Jakob Nielsen. 2012. Response time limits. (2012).

[13] Ronald Rivest. 1992. *The MD5 Message-Digest Algorithm.* RFC 1321. https://www.rfc-editor.org/rfc/rfc1321.txt

[14] Marc Shapiro, Nuno Perguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems (Lecture Notes in Computer Science)*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.), Vol. 6976. Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

[15] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://hal.inria.fr/inria-00555588

[16] Chengzheng Sun and Clarence Ellis. 1998. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work (CSCW '98).* ACM, New York, NY, USA, 59–68.

[17] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. 2017. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17).* International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 283–292. https://doi.org/10.1145/3038912.3052673

[18] 2014. Antidote. http://syncfree.github.io/antidote. (2014).

[19] 2009. Apache Cassandra. https://cassandra.apache.org. (2009).

[20] 2005. CouchDB. https://couchdb.apache.org. (2005).

[21] 2018. Google Docs. https://support.google.com/docs/answer/2494822. (2018).

[22] 2009. MongoDB. https://www.mongodb.com/. (2009).

[23] 2013. PouchDB. https://pouchdb.com. (2013).

[24] 2016. Pumba. https://github.com/alexei-led/pumba. (2016).

[25] 2010. Riak. http://docs.basho.com/riak/kv. (2010).

[26] 2013. ShareDB. https://github.com/share/sharedb. (2013).

[27] 2018. Speedtest.net. http://www.speedtest.net/reports/united-states/, last accessed on 20/09/18. (2018).

[28] 2013. Swarm.js. https://github.com/gritzko/swarm. (2013).

[29] 2014. Yjs. https://github.com/y-js/yjs. (2014).