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

Submission #181

Abstract

Many 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 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 period of time. OWebSync implements a fine-grained data synchronization model and leverages 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. Compared to other state-based approaches, OWebSync can achieve acceptable interactive performance with limited network overhead at a higher scale. This has been validated and evaluated in two industrial case studies.

1 Introduction

Web applications are 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 as HTML5 and JavaScript are becoming more powerful. The latter reason should be disappearing too with the arrival of WiFi, 4G and 5G ubiquitous wireless networks. 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.

Many native application-specific solutions and browser-plugins exist to tackle this problem in an ad-hoc solution. For example, many Google web apps can be used in offline mode. 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 prompt resynchronization for offline clients,
4. scale to tens (20-30) of online clients that concurrently edit a single shared document with interactive performance timings.

In his book on usability engineering [13], Nielsen states that remote interactions should take only one to two seconds to keep the user experience seamless. This is our basis for *prompt* synchronization with *interactive* performance. 10 seconds is the absolute maximum before users are leaving the web application. Many existing middlewares and frameworks exist to achieve this in the online setting. However, recovering from a failure often takes several tens of seconds or the applications need to implement complex conflict-resolution strategies themselves.

Many distributed NoSQL data systems, e.g. Amazon Dynamo [4], adopt synchronization based on Vector Clocks. This often leads to conflicts that need application-level resolving. Text-based versioning systems such as Git are not made to manage data structures and do not always guarantee valid data structures after synchronization. Code, XML or JSON documents can end up malformed and often require user-level resolution. Operational Transformation [20] is used for real-time synchronization in interactive web applications (e.g. in

Google Docs [25]) but is not resilient against message loss in case of long-time offline situations [9]. Operation-based Conflict-free Replicated Data Types [19] (CRDTs), as used in SMAC [5] and the JSON datatype of Kleppman [8], are also operation-based, but don't apply transformations to the operations. The operations are commutative and can arrive and be applied in a different order. However, this technique also requires a reliable message channel with exactly-once delivery. State-based (CRDTs) [19] 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 [29] for example, to achieve background, asynchronous synchronization between back-end data centers internally. Delta-state CRDTs [1] are an optimization of state-based CRDTs that reduces the network usage by only sending deltas based on the current version of the data at the client. However, this approach is slow to recover from long disconnections because it needs to fall back to the pure state-based approach. This approach also needs to keep track of different client versions in the metadata, which does not integrate well with a web application architecture.

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, prompt and fine-grained synchronization between online clients. The middleware supports prompt and seamless 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 [11] are used to enable seamless and prompt synchronization of state-based CRDTs and limit the amount of data that has to be transferred. It also doesn't require to store metadata about the different client versions - not in the data model, and not at the server. 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 [30, 33], 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.

[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.

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. 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 provide background information on Operational Transformation, Conflict-free Replicated Data Types and Merkle-trees, and motivate our approach of state-based CRDTs with Merkle-trees.

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.

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. Writing off all used materials is crucial for correct billing and inventory afterwards.

The company *eDesigners* offers a customer-facing multi-tenant web app 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 offline. When two users edit the same document, a conflict occurs, and the versions need to be merged. Edits that are independent of each other should both be applied to the template (e.g. 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. The previous section described the 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 removes 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 [19] are data structures that guarantee eventual consistency without the need for *explicit* conflict handling during synchronization by the application or the 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 OT. Concurrent operations in CmRDTs need to be 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 [18].

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 require little from the message channel, messages can get lost or arrive out of order 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. δ-CvRDTs [1] are a variant on state-based CRDTs with the advantage that in some cases only part of the state (a delta) needs to be sent for a correct synchronization. When a client performs an update, a new delta is generated which reflects the update. Each client keeps a list of deltas and remembers which clients have already acknowledged a delta. As soon as all clients have acknowledged a delta, the delta can be discarded because the update is now reflected in the state of all clients. If a client was offline for some reason and has missed too many deltas, the full state must be sent, just like normal state-based CRDTs.

δ-CRDTs have some problems when using them in web applications. Browser-based clients come and go with a large churn rate and it is often unclear if a client will come back online in the future (e.g. browser cache cleared). Keeping extra metadata for all those clients, to be able to synchronize only the required deltas, can result in a large storage or memory overhead to keep track of them at the server. One can always discard the metadata for clients that were offline and send the full state if they do come back online eventually. But this is of course not efficient when the state is large and that client already had most of the updates. Keeping metadata about all browser-based clients and their versions also doesn't match a stateless web application architecture.

A variant of δ-CRDTs, called Δ-CRDTs [22], is proposed as solution to this problem and is used in Legion [21]. Δ-CRDTs are comparable to δ-CRDTs, but instead of keeping track of the clients at the server, it includes extra metadata about concurrent versions of all clients in the data model (e.g. as vector clocks) to calculate the deltas dynamically. This solves the problem of keeping track of all clients at the server, but one might still need to synchronize the full state after long disconnects.

Merkle-trees. Merkle-trees [11] or hash-trees are used to quickly compare two large data structures. First each item in a data structure is hashed. 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 top-level hashes match, the data structures are equal. Otherwise, the tree can be descended using the mismatching hashes to find the mismatching items.

Approach. OWebSync uses state-based CRDTs, which require little from the message channel in comparison to operation-based approaches. No state about other clients or client-based versioning metadata needs to be stored. And even after long offline periods, the missed updates can be computed and synchronized seamlessly. 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 efficiently. This data structure and its building blocks is discussed in Section 3. Together with other architectural performance tactics and implementation-level optimizations we can achieve prompt and seamless synchronization in interactive multi-user web applications. 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 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) [19] and an Observed-Removed Map (ORMap) [19] 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 merge operation will always result in the value and timestamp of the latest update. The timestamp is only used when a conflict occurred, i.e. one or more clients have updated the value concurrently. This conflict resolution strategy boils down to a simple last-write-wins strategy.

Observed-Removed Map (ORMap). The Observed-Removed Map is implemented using an Observed-Removed Set (ORSet) as described by Shapiro et al. [19]. Internally, the ORSet contains two sets, the observed set and the removed set, to keep track of the items that are added to the set and which items are removed. A unique ID (UUID [10]) is added to each item to make it possible to add a removed item back to the set, since it will have a different ID when added again. The ORMap contains tuples with a value and an ID, just like an ORSet, and an extra key. 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 [16] 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 comparing the top-level hash only. Figure 1 shows the internal structure of an ORMap. 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. When the remove operation removes an item, only the ID needs to be kept internally and the whole sub-tree of the removed item can be discarded. The update

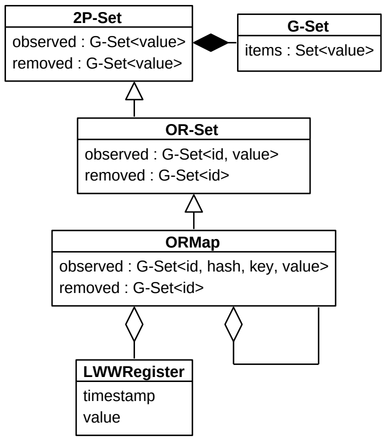


Figure 1: Class diagram of the CRDTs in OWebSync.

operation will update the value and the hashes. To merge two ORMaps, the union of the respective observed and removed set is taken, just like in a regular ORSet. Then, the hashes of the Merkle-tree are compared to check for changes in the children of the ORMap. When a mismatch is detected, the merge 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, i.e. a concurrent add and remove operation will result in the item being present in the set, since each add will get a new identifier. 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 internal 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 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. In fact, any data that can be modeled in a tree-like structure such as JSON, can tolerate eventual consistency and doesn't require constraints between the data, can use OWebSync for the synchronization. This data model is less suited for applications like online banking which requires constraints such as: "your balance can never be less than zero". Text-editing is also not a great fit, because there is not much structure in the data. If you would see text as a list of

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

(a) Conceptual representation of a single data object.

```
* drawings.drawing1.object36:
      uuid: 0a2f7bc2-129f-11e9-ab14-d663bd873d93
      hash: 7319eae53558516daafac19183f2ee34
      observed:
          - uuid: 23c1259a-129f-11e9-ab14-d663bd873d93
            hash: 65bdd1b610f629e54d05459c00523a2b
            key: "top"
          - uuid: 23c1259a-129f-11e9-ab14-d663bd873d93
            hash: 67507876941285085484984080f5951e
            key: "left"
          ...
      removed:
* drawings.drawing1.object36.top:
      uuid: 23c1259a-129f-11e9-ab14-d663bd873d93
      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.

characters, it would result in a tree with one top-level node (the document) and one layer with many child nodes (the characters). There won’t be much benefit in using a Merkle-tree. OWebSync also expects that no client is malicious.

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 remove 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 automatically logging out the user after a month of no usage.

An extra kind of conflict is possible when assigning different kinds of CRDTs to the same path. Then the merge-operation of the defined CRDTs cannot be used to resolve the conflict automatically. This is solved by posing an order on the possible CRDTs, e.g. LWWRegister $<$ ORMap. This means that when such a conflict occurs, the ORMap is selected as actual value, while the LWWRegister is discarded.

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 [32], 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 ([17,19]) and could be added in future work.

Adding new kinds of CRDTs to the data model is straightforward. An existing CvRDT can be used as is, except for an extra hash to be part of the Merkle-tree. For a CRDT that represents a leaf value (e.g. a Multi-Value Register [19]), the hash is simply the hash of that value. For CRDTs that can contain other values (e.g. a list [17]), a hash needs to be added that combines the hashes of all the children.

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, prompt 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 and storing all shared data. The server is only responsible for data synchronization and does not run application logic. 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 WebSockets [7]. All

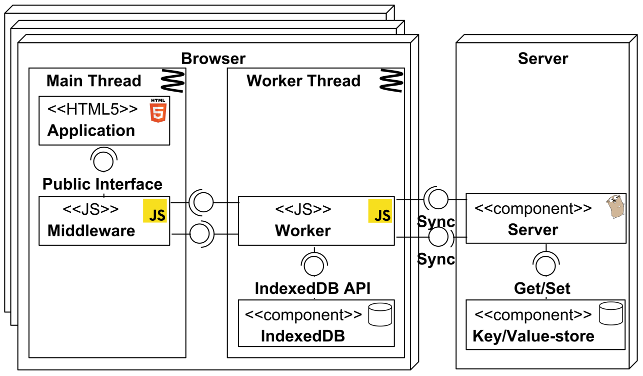


Figure 3: Overall architecture of the OWebSync middleware

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.
- `REMOVE(path)`: Remove the value or sub-tree at the 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 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);
```

The difference between the levels of hierarchy is as follows. The object at `"drawings.drawing1"` is fetched from disc and is represented as a JavaScript object in-memory. If there would be other drawings (e.g. drawing2), they won't be loaded. The access to `"d1.object36.color"` is just a plain JavaScript object access and has nothing to do with OWebSync. For performance reasons, it is best to always scope to the smallest possible object from the database, in this example that would be like this:

```
OWebSync.set("drawings.drawing1.object36.color",
    "#f00")
```

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:

- `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.
- `PUSH (path, CRDT)`: the sender sends the CRDT data structure at a given path and the receiver will merge it at the given path.
- `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 removed.

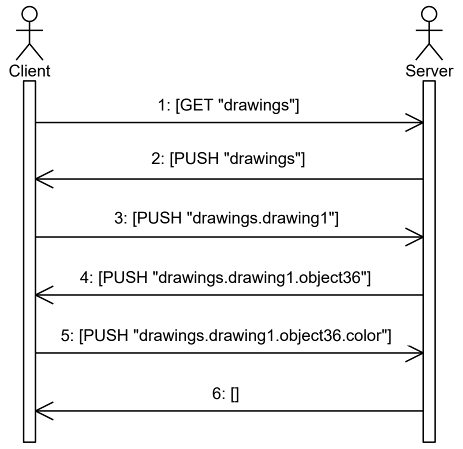
The protocol is initiated by a client, which will traverse the Merkle-tree of the CRDTs. The synchronization starts with the highest CRDT in the tree. The client will send a GET message to the server with the given path and hash value of the CRDT. If the server concludes that the hash of the path matches 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. This doesn't include the child CRDTs, only the metadata (key, UUID and hash) of the immediate children. The client must merge the new CRDT with the CRDT at its requested path. This merger process at the client might detect conflicting children in the tree by comparing the hashes. The client will then PUSH the CRDTs of those conflicting children to the server. The server then needs to merge those CRDTs. 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 leaves of the tree are 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. Per PUSH-message that is sent, the process descends one level in the Merkle-tree. It is therefore limited to the depth of the Merkle-tree.

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 UUID 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 a UUID.

Figure 4 shows an example for the eDesigners case study where the client 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.



Client
Server
1: [GET "drawings"]
2: [PUSH "drawings"]
3: [PUSH "drawings.drawing1"]
4: [PUSH "drawings.drawing1.object36"]
5: [PUSH "drawings.drawing1.object36.color"]
6: []

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

Performance optimization tactics. The main optimization tactic to achieve prompt 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, many messages are coming in while still doing the first synchronization. This results in many duplicated messages and doing a lot of duplicated work on sub-trees, since the top-level hash will only be up-to-date when the bottom of the 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 are sent 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 sub-trees.

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

Extra levels in the Merkle-tree. When the number of child values in an ORMap increases, all the metadata for those children (key, UUID and hash) needs to be sent each time during the synchronization to check for changes. This leads to very high network usage, since it cannot make use of the Merkle-tree efficiently. To solve this problem, we introduced extra, virtual, levels in the Merkle-tree. Whenever an ORMap needs to be transmitted which contains many children (i.e. hundreds), instead an extra Merkle-tree level is sent. This extra level combines the many children in groups of e.g. 10. This number can be adapted to the total number of children. As a result, 10 times less hashes will be sent, combined with the key-ranges the hashes belong to. The other party can verify the hashes and determine which ones are changed and then push the 10 children for which the combined hash didn't match. This improvement leads to a slight delay in synchronization time since it adds one extra round-trip, but when only a small part of the children is updated, it uses much less bandwidth.

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 four times on four representative JavaScript technologies for web-based data synchronization: our OWebSync platform, which uses state-based CRDTs with Merkle-trees, Yjs [33] which uses operation-based CRDTs, and ShareDB [30] which makes use of OT. We used Legion [21] for testing delta-CRDTs. Both Yjs (845 GitHub stars) and ShareDB (2129 GitHub stars) are widely-used open source technologies that are available on GitHub. Legion is not widely-used in production, but is currently the only implementation of delta-CRDTs in JavaScript to the best of our knowledge. We did not evaluate Google Docs, which uses OT, because it is text based, and can not be used to synchronize the JSON-documents used in the benchmark. Instead we opted for ShareDB.

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 [28] 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.

Our evaluation contains 48 benchmarks: 6 benchmarks to be executed by each of the 4 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 itself [25]. 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 operation-based synchronization middlewares.

In our performance evaluation, one iteration of a benchmark takes about 11 minutes. The first 3 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 the consistency of the performance metrics. In order to execute all 48 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, Legion and OWebSync did not fail to synchronize all updates. The synchronization times of the succeeded updates are illustrated in Figure 5.

Analysis of the results. For the benchmark with 8 clients and 100 objects, both Yjs and ShareDB synchronize the updates faster than OWebSync and Legion. For these two operation-based approaches, 99% is below 0.6 seconds. OWebSync needs about 2.3 seconds for synchronizing the 99th percentile and for Legion the 99th percentile is at 3.6 seconds. The reason for this is that OWebSync and Legion don't keep track of which updates are sent to which client. Hence, each time one wants to synchronize the data, a few extra round-trips are required to calculate which updates are needed. Yjs and ShareDB can just send the operations. On a faster network, i.e. with less latency, both Legion and OWebSync will synchronize faster than in this benchmark (since the time for a round-trip will be less), but will never be able to match the performance of operation-based approaches.

This changes when the scale of the benchmark increases. For the benchmark with 24 clients and 1000 objects, OWebSync and Legion become faster with a maximum of respectively 2.7 and 4.2 seconds for the 99th percentile. Yjs and ShareDB require tens of seconds for the 50th percentile and even hundreds of seconds for the 90th percentile. With this number of clients and this size of the data, keeping track of all the clients and maintaining exactly-once semantics of operation delivery, give a large overhead.

We can conclude that the synchronization times of OWebSync and Legion only slightly increase when scaling up to 1000 objects and 24 clients. Moreover, all the OWebSync benchmarks show consistent results during their 10 iterations, while Legion has some ill-performing outliers. However, the update times for Yjs and shareDB increase significantly when increasing the number of clients. For the Yjs and ShareDB benchmarks with 24 clients and 1000 objects, the results start to increase and fluctuate more as these technologies start to struggle with the scale of the benchmark.

Network trade-off. The trade-off for this scalable, prompt synchronization, is that OWebSync has the largest network usage of all tested technologies (Figure 6). The usage of Merkle-trees reduced the network usage with about a factor 8 in the worst case (1000 objects under a single node in the tree). Introducing extra levels in the Merkle-tree for nodes with many children lowered the bandwidth another factor 3. Even in the benchmark with 24 clients and 1000 objects, the used bandwidth is only 280 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 [31]. In this same benchmark, the server consumes about 6.7 Mbit/s, which is acceptable for a typical data center.

[2]Tables with the detailed performance results have been submitted as supplementary material. The raw logs of all 48 benchmarks, and the graphical analysis in boxplots of each iteration (to verify the consistency), are available on an anonymous Azure storage account: `https://owebsyncdata.blob.core.windows.net/logs/data.zip`

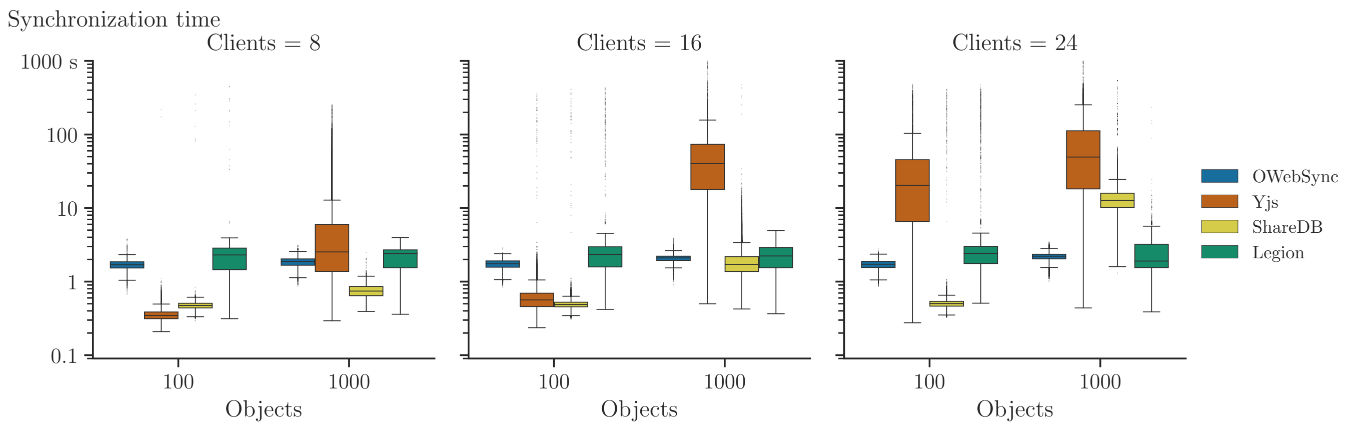


Figure 5: Aggregated boxplots containing the times to achieve full synchronization to all clients. Each boxplot contains all 10 iterations for each of the 24 benchmarks in the fully online situation. In order to compare technologies that have results of the same order of magnitude, as well as results in different orders of magnitude, we opted for a logarithmic Y axis.

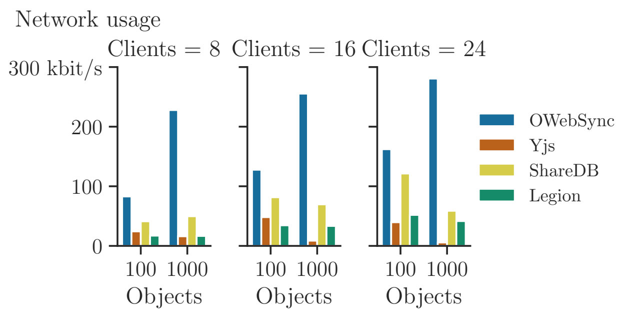


Figure 6: Network usage per client for each benchmark. Some technologies use less bandwith when the scale increases, because the time to synchronize increases even more.

The data structure has an important effect on the network usage. One might create a tree-structure with few nodes which have many children. This will make the Merkle-tree less useful, since the metadata of all the children needs to be exchanged to be able to determine which children are updated. The other possibility is that there are less children per node, but with an increased depth of the tree. This positively affects the network usage, as less metadata will need to be exchanged. However, synchronizing the whole tree will take more roundtrips as there are more levels in the tree to go through.

Interpretation and discussion. For interactive web applications, usability guidelines [13] [14] state that remote response times should typically be 1 to 2 seconds on 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]).

The mean synchronization time for the OWebSync benchmark with 24 clients and 1000 objects is around 2.2 seconds. The 99th percentile is at the border line of annoyance with 2.7s. 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 Dang et.al. [3] for Google Docs, which uses the same approach as ShareDB (OT).

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 (2 minutes in the actual test as the first minute is used as warm-up) for 1 minute using Pumba [28]. Again, Yjs only succeeds in synchronizing 10% to 23% of the updates for the benchmarks with 24 clients.

Analysis of the results. The boxplots of these benchmarks (Figure 7) show that OWebSync can synchronize all missed updates faster than Yjs, ShareDB and Legion. In summary, the time to synchronize all missed updates in case of network failure for OWebSync is between 2.1 and 5.8 seconds, which is acceptable for interactive online web applications. The other technologies need tens or hundreds of seconds to process all of the missed updates. Yjs and ShareDB need to replay all missed operations on the client that was offline. This is due to their operation-based nature. OWebSync only needs to merge the new state, which it does in exactly the same way as if the failure never happened. Legion could keep up with OWebSync in the online scenario, but now we see that resynchronization after a failure starts to take longer when the scale of the benchmark or the size of the dataset increases.

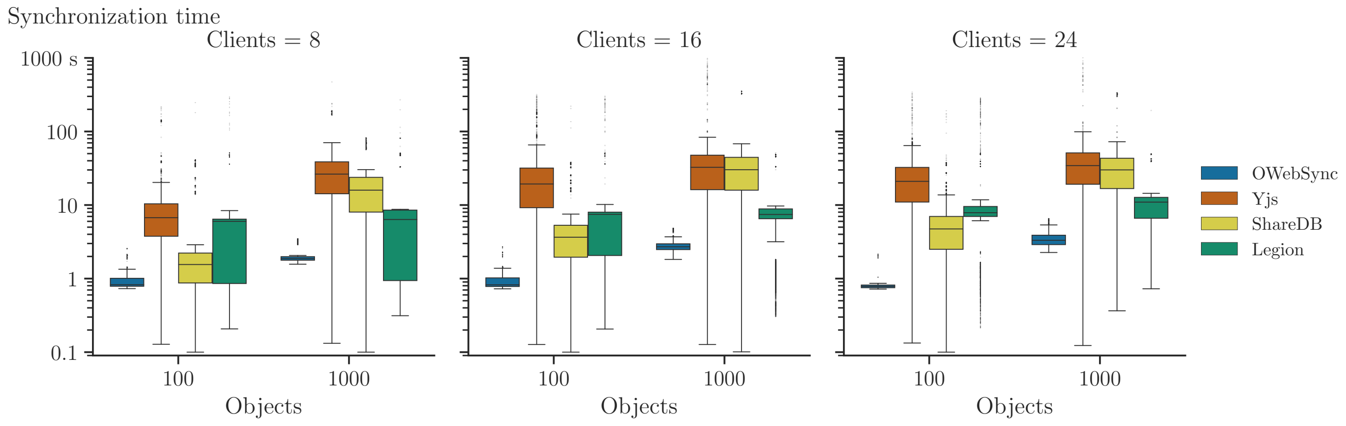
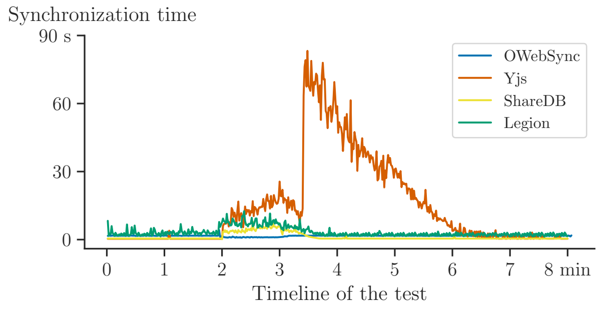
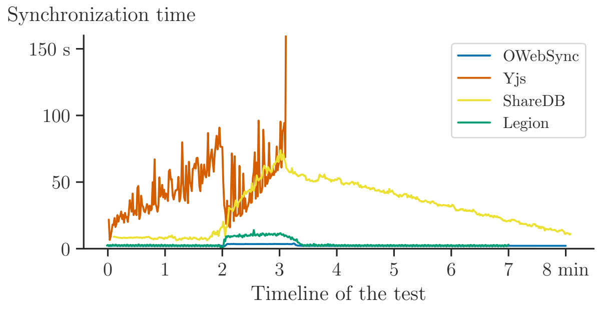


Figure 7: Boxplots of the time it takes for an update done during the failure scenario to be received by all clients. The time before a client notices the network is repaired is not taken into account.



(a) Evolution of the time to synchronize updates in the benchmark with 8 clients, 100 objects and network failure.



(b) Evolution of the time to synchronize updates for the benchmark with 24 clients, 1000 objects and network failure.

Figure 8: Mean time to synchronize updates after the network failure, without the time during the failure taken into account.

Timeline analysis of the benchmarks. The timeline graphs in Figure 8 show the resynchronization times on the y-axis, without the offline time during the failure, for each update done at a given moment during the benchmark timeline (x-axis). 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.

Figure 8a shows this for 8 clients and 100 objects. The first two minutes show consistent synchronization times for all four technologies. In this part of the benchmark, OWebSync and Legion are the slowest, as was the case for the online situation. After those two minutes, the network of one client is disrupted. One minute later, the network is repaired and the synchronization times drop as full synchronization is possible again. OWebSync returns immediately to the same performance as before the failure. ShareDB and Legion also achieve the original performance again, but this takes about half a minute. Yjs will block the synchronization of new updates to first synchronize missed updates from during the failure, and only then resumes normal synchronization. This leads to an extra peak in synchronization time right after the failure and it takes several minutes for Yjs to stabilize again.

In the benchmark with 24 clients and 1000 objects (Figure 8b), OWebSync still returns to the same performance as before the failure after about 20 seconds. Legion needs half a minute, but at that time, not all of the missed updates are fully synchronized yet. ShareDB takes several minutes to achieve this. Yjs clients can no longer handle the combination of ongoing updates and delayed updates, and start failing to synchronize after 4 minutes due to client-side load.

Summary. Our evaluation shows that the operation-based approaches work well in continuous online situations with a limited number of users. They can synchronize faster than OWebSync and Legion. However, when network disruptions occur, or when the number of users scales up, these technologies cannot achieve acceptable performance and need tens or hundreds of seconds to achieve synchronization.

Δ-CRDTs can improve the scalability in the online scenario, but take a longer time to synchronize after a network failure. OWebSync can then achieve much better performance in the order of seconds, which is still acceptable for interactive web applications. Table 1 summarizes the results in seconds of the large scale benchmark (24 clients, 1000 objects) for the average user (50th percentile) and most of the users (99th percentile) for both the online and offline setting.

In a setting with frequent offline situations, e.g. with mobile employees, OWebSync is the more appropriate technology and outperforms all other technologies. Multiple clients going offline will only widen this gap. In a well-connected online situation, e.g. with only LAN-connected employees in the back office, Legion offers better scalability in terms of network usage, but that is not really a resource constraint in such setting.

	online		offline	
	50%	99%	50%	99%
Yjs	74.9	509.7	34.0	442.5
ShareDB	13.8	129.9	30.1	321.7
Legion	1.9	4.2	9.1	44.7
OWebSync	2.2	2.7	3.4	5.8

Table 1: Summary of the synchronization times in seconds.

6 Other related work

The related work consists of three types of work: 1) concepts and techniques such as CRDTs and OT, 2) NoSQL data systems such as Dynamo and Cassandra, as well as 3) synchronization frameworks for web clients. The concepts and techniques were discussed in Section 2. In this section we focus on the relevant NoSQL data systems and on synchronization frameworks.

Distributed data systems and NoSQL. Based on the original Dynamo paper [4], many other open-source NoSQL systems have been developed for structured or semi-structured data, focusing on eventual consistency within or between data centers. CouchDB [24] and MongoDB [26] 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 [29] 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 [23] 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. Cimbiosys [15] is an application platform that supports content-based partial replication and synchronization with arbitrary peers. While it shares some of the goals of OWebSync, it is best suited to synchronize collections of media data (e.g. pictures, movies) and not for JSON documents with fine-grained conflict resolution.

Client-tier JavaScript-libraries for synchronization. Many JavaScript frameworks have appeared to enable synchronization between web browsers and server-side data systems. PouchDB [27] 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 [30] is a client-server framework to synchronize JSON documents and adopts OT 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 [12, 33] 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. Legion [21] is a framework for extending web applications with peer-to-peer interactions. It also supports client-server usage and uses delta-state CRDTs for the synchronization. Swarm.js [32] 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 seamless 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 prompt and fine-grained synchronization for online interactive web applications with continuous concurrent updates.

Our comparative evaluation shows that the operation-based approaches cannot achieve acceptable performance in case of network disruptions or larger scale settings and need tens or hundreds of seconds to achieve synchronization. Current state-based approaches using delta-CRDTs are more scalable than the operation-based ones, but cannot achieve timely synchronization after being offline. The state-based approach with Merkle-trees 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. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111(Supplement C):162 – 173, 2018.

[2] Tim Bray. The javascript object notation (json) data interchange format. RFC 7158, IETF, 2014.

[3] Quang-Vinh Dang and Claudia-Lavinia Ignat. 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, pages 548–553, Vienna, Austria, May 2016. IFIP.

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

[5] Jacob Eberhardt, Dominik Ernst, and David Bermbach. Smac: State management for geo-distributed containers. Technical report, Technische Universitaet Berlin, 2016.

[6] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, June 1989.

[7] Ian Hickson. The websocket api, w3c candidate recommendation. Technical report, 2012.

[8] Martin Kleppmann and Alastair R Beresford. A conflict-free replicated json datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.

[9] Santosh Kumawat and Ajay Khunteta. A survey on operational transformation algorithms: Challenges, issues and achievements. *International Journal of Computer Applications*, 3(12):30–38, July 2010.

[10] Paul Leach, Michael Mealling, and Rich Salz. A universally unique identifier (uuid) urn namespace. RFC 4122, 2005.

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

[12] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Yjs: A framework for near real-time p2p shared editing on arbitrary data types. In *Engineering the Web in the Big Data Era*, pages 675–678, Cham, 2015. Springer International Publishing.

[13] Jakob Nielsen. *Usability Engineering*. Nielsen Norman Group, 1993.

[14] Jakob Nielsen. Website response times. https://www.nngroup.com/articles/website-response-times/, 2010.

[15] Venugopalan Ramasubramanian, Thomas L Rodeheffer, Douglas B Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C Marshall, and Amin Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 261–276, 2009.

[16] Ronald Rivest. The md5 message-digest algorithm. RFC 1321, 1992.

[17] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

[18] Marc Shapiro, Nuno Perguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400, Berlin, Heidelberg, October 2011. Springer Berlin Heidelberg.

[19] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011.

[20] Chengzheng Sun and Clarence Ellis. 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, pages 59–68, New York, NY, USA, 1998. ACM.

[21] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. Legion: Enriching internet services with peer-to-peer interactions. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 283–292, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[22] Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-crdts: Making Δ-crdts delta-based. In *Proceedings of the 2Nd Workshop on the Principles and Practice*

of Consistency for Distributed Data, PaPoC '16, pages 12:1–12:4. ACM, 2016.

[23] Antidote. http://syncfree.github.io/antidote, 2014.

[24] Couchdb. https://couchdb.apache.org, 2005.

[25] Google docs. https://support.google.com/docs/answer/2494822, 2018.

[26] Mongodb. https://www.mongodb.com/, 2009.

[27] Pouchdb. https://pouchdb.com, 2013.

[28] Pumba. https://github.com/alexei-led/pumba, 2016.

[29] Riak. http://docs.basho.com/riak/kv, 2010.

[30] Sharedb. https://github.com/share/sharedb, 2013.

[31] Speedtest.net. http://www.speedtest.net/reports/united-states/2018/Mobile/, 2018.

[32] Swarm.js. https://github.com/gritzko/swarm, 2013.

[33] Yjs. https://github.com/y-js/yjs, 2014.