OWebSync: A Web Middleware with State-Based Replicated Data Types and Merkle-Trees for Seamless Synchronization of Distributed Web Clients

Anonymous Author(s)
Submission Id: 75

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 often offline due to expected or unexpected network disruptions. Nevertheless, continuous operation of the software 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 state-based Convergent Replicated Data Types to achieve the required performance, both for online interactive clients, as well as for resynchronizing clients that have been offline.

In comparison with operation-based and state-based middleware solutions, OWebSync is especially better in operating in and recovering from offline settings and network disruptions. Compared to operation-based solutions, OWebSync also scales better to tens of concurrent editors on a single semi-structured document. Compared to other state-based approaches, it doesn't require to transmit the full state, and also doesn't store metadata on the server about the client versions. OWebSync 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 a software service while in an airplane.

Collaborative web applications with concurrent edits on shared data should offer *prompt* synchronization with *interactive* performance. The research of Nielsen on usability engineering [24] states that remote interactions should take only one to two seconds to keep the user experience seamless. Users are annoyed after a 5 second waiting period and 10 seconds is the absolute maximum before users are leaving the web application.

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 shared document with interactive performance,
5. tolerate hundreds of clients over time without inflating and polluting the data with versioning metadata.

Several client-side frameworks exists for synchronization of semi-structured data. They are either operation-based, state-based or delta-state based. Operation-based approaches distribute the updates as operations to all replicas. Operational Transformation (used in Google Docs [45]) is a popular operation-based technique for real-time synchronization in web applications, but it is not resilient against message loss or out-of-order messages. It also requires a central server transforming the operations for other clients to deal with concurrent changes. Commutative Replicated Data Types [31], as used by SwiftCloud [26, 34], Yjs [22, 23, 54] and Automerge [16, 39] are also operation-based. Again, updates need to be propagated, as operations, to all clients using a reliable, exactly-once, message channel. However, no transformation is required because operations are required to be commutative. State-based Convergent Replicated Data Types [31] are resilient against message loss, but a large amount of data has to be transferred between all replicas. This approach is often used to achieve asynchronous background synchronization between data centers, e.g. in Riak [50], and is less suited for interactive collaborative applications. Delta-state based CRDTs [1, 33], as used by Legion [32, 46], need much less of the message channel. They

use vector clocks to calculate delta-updates, which require one entry per client per object in the server-side metadata. This doesn't integrate well with a stateless web application architecture and the dynamic nature of the web. It is often uncertain if a web client will ever connect to a server again.

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 contribution of this paper is a data-synchronization middleware with the following advances compared to state-of-the-art frameworks:

1. prompt and seamless resynchronization when clients were offline due to expected or unexpected network disruptions, while maintaining interactive synchronization in the online settings,
2. minimization of data-transfer of state-based CRDTs by using Merkle-trees [21],
3. no metadata is stored on the server about the different client versions and identifiers.

At the implementation level, OWebSync provides a generic, reusable JSON [6] based data type that web applications can leverage to model their application data. One can nest several map structures into each other to build a complex tree-structured data model. These data types support fine-grained and conflict free synchronization of all items in the tree-based JSON document.

Our comparative evaluation shows that all online clients receive updates from other clients within the time span of seconds, even when tens of clients are editing hundreds of shared objects in a single document. This makes our solution suitable for online, interactive and collaborative applications. Compared to operation-based middlewares [39, 51, 54], OWebSync is especially better in recovering from offline situations, even with silent network disruptions. Compared to state-based approaches, it can reduce the network usage and the storage of server-side metadata per client.

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 synchronization architecture. Section 5 compares and evaluates performance in online and offline situations using OWebSync and other state-of-the-art synchronization frameworks. We discuss related work in Section 6 and then we conclude.

[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.

2 Motivation, Background and Approach

This section 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 offline settings due to expected or unexpected network disruptions. 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.

2.1 Case studies

The motivation and requirements have emerged from two industrial case studies from our applied research projects, that have also been used for the evaluation of the middleware. The first case study is an online software service from eWorkforce, 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. eWorkforce has two kinds of employees that use the online software service: the help desk operators at the office and the technicians on the road. The help desk 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 their 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.

This case study needs to support long term offline usage, with quick synchronization when coming online, especially for last minute changes to the work plan of the technicians. The help desk software needs to be operative at all times, even without connection to the central database, because customers can call for support and schedule interventions.

eDesigners. 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 changes the color of an object, another edit changes 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.

This case study requires that the application is always available, even on an airplane. Updates always need to be possible, even when offline. When coming back online, the updates need to be synchronized promptly without requiring the user or the application to manually resolve conflicts. When online, the performance should be interactive, especially when two users are working on the same template next to each other.

2.2 Background, principles and approach

The previous section described the overall goal of OWebSync. This section now describes 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 and Conflict-free Replicated Data Types.

Operational Transformation. OT [11] is a technique that is often used to synchronize concurrent edits on a shared document. OT works by sending the operations to the other replicas. The operations are not necessarily commutative, which means they cannot be applied immediately on other replicas. A concurrent edit might conflict with another operation, e.g. the location could have changed. Therefore, a central server is used to transform the operations for the different replicas so that the resulting operations maintain the original semantics. The problem is that the transformation of the incoming operations of other clients on their local current state can get very complex. Messages can also get lost or can arrive in the wrong order. Hence, OT is not resilient against message loss in offline situations [18].

Conflict-free Replicated Data Types. CRDTs [30, 31] are data structures designed for replication that guarantee eventual consistency without *explicit* coordination with other replicas. Conflict-free 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 manually. There are two kinds of CRDTs: operation-based or Commutative Replicated Data Types (CmRDT) and state-based or Convergent Replicated Data Types (CvRDT).

Commutative Replicated Data Types. CmRDTs [30] 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. Causally ordered delivery is required in some cases.

Convergent Replicated Data Types. CvRDTs [30] 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, 2] are a variant of state-based CRDTs with the advantage that in some cases only part of the state (a delta) needs to be sent for 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 and has missed too many deltas, then the full state must be sent, just like with 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.

A variant of δ-CRDTs, called Δ-CRDTs [33], is proposed as solution to this problem. 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 (vector clocks) to calculate the deltas dynamically. This solves the problem of keeping track of the deltas for clients at the server, but it still needs client identifiers and version numbers inside the vector clocks for each object, and each client that made a change.

Another approach to optimize δ-CRDTs is using join decompositions [12, 13]. This approach doesn't extend CRDTs with additional metadata that needs to be garbage collected. Instead, it can efficiently calculate a minimal delta to synchronize. While this improves the network usage compared to normal δ-CRDTs, it still requires clients to keep track of their neighbours. When there is no such data available (e.g. after a network partition), it needs to fallback to a state-based approach. However, it only requires sending the full state in a single direction (compared to bidirectionally in normal state-based CRDTs). A digest-driven approach is also supported, which will send a smaller digest of the actual state. However, for many CRDTs, such digest doesn't exist and for large, nested data, this digest would still be very large.

Merkle-trees. Merkle-trees [21] 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. Our use cases require both interactive performance for the online clients, as well as fast bidirectional resynchronization when a client was offline. The current state-of-the-art solutions suffice for the online clients, at least for small scale settings with 10 clients. But the performance of all of them degrade quickly when they need to synchronize with a client which was offline for some time or with a new client which they never saw. Keeping metadata about all browser-based clients doesn't match the characteristics of the web, where clients come and go at a fast rate.

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, unlike delta-state approaches. And even after long offline periods, the missed updates can be computed and synchronized seamlessly. To limit the overhead of full 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 are discussed in Section 3. Together with other architectural performance tactics, we can achieve prompt synchronization in interactive multi-user web applications. This is discussed in Section 4.

3 The OWebSync Data Model

This section describes 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) [31] and an Observed-Removed Map (ORMap) [31] 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. 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.

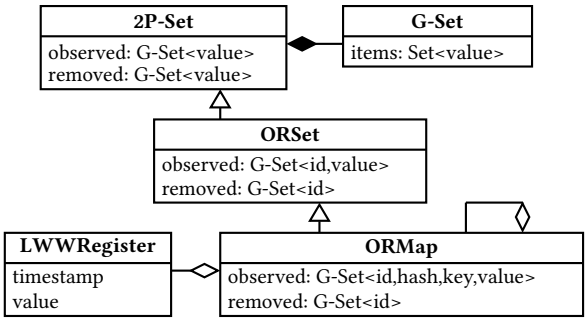

Figure 1. Class diagram of the CRDTs in OWebSync.

Observed-Removed Map. The Observed-Removed Map is implemented using an Observed-Removed Set (ORSet) as described by Shapiro et al. [31]. 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 [20]) 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 [28] 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 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`.

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, that can tolerate eventual consistency and that 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 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 doesn't 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.

Another kind of conflict occurs when assigning different types 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.

Another conflict is a concurrent remove and update of a child CRDT. The CRDT proposed here maintains a remove-wins semantic. This means that updates done to children, are discarded when the parent is removed concurrently.

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 [53], 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 ([29, 31]) 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 [31]), the hash is simply the hash of that value. For CRDTs that can contain other values (e.g. a list [29]), a hash needs to be added that combines the hashes of all the children.

4 Web-based synchronization architecture

This section describes 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 a client and a server subsystem. 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 doesn't 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 [15]. All communication messages between client and server are sent and received using asynchronous workers inside the client and server subsystems. We first 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.

```
{
    "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: 0eac2a3a-546f-11e9-8647-d663bd873d93
            hash: 675078769412850854849840 80f5951e
            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.

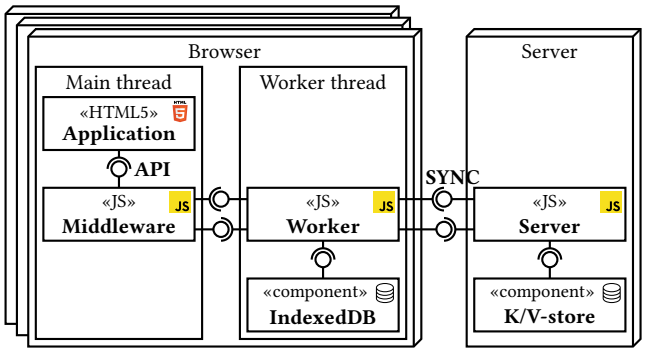

Figure 3. Overall architecture of the OWebSync middleware

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 `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 doesn't 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.

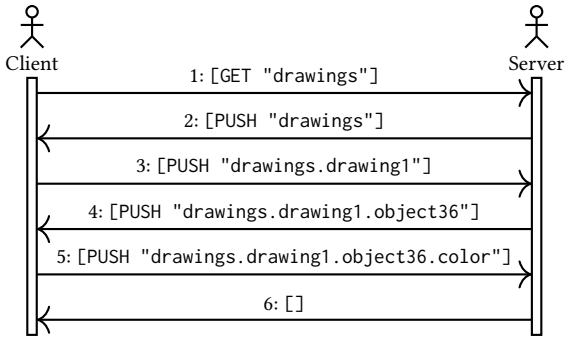

Figure 4. Synchronization protocol when the client made an update. With every PUSH message, the respective CRDT is sent. E.g. for message 4, the first CRDT in Figure 2b is sent.

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 doesn't 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. The number of messages (and thus the length of the synchronization protocol) is therefore limited to the maximum 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.

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 concurrently, 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, fewer 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.

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

the time it takes for all clients to get back in sync with each other after the network disruption, and in the time it takes to restore normal interactive performance.

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 fewer shared data with fewer concurrent updates as technicians typically work on their own data island and the data contains fewer objects with less frequent changes. To compare performance, we implemented the eDesigners case study five times on five representative JavaScript technologies for web-based data synchronization: our OWebSync platform, which uses state-based CRDTs with Merkle-trees, Yjs [54] and Automerge [39] which use operation-based CRDTs, and ShareDB [51] which makes use of OT. We used Legion [32] for testing delta-CRDTs. Both Yjs (917 GitHub stars) and ShareDB (2323 GitHub stars) are widely-used open source technologies that are available on GitHub. Automerge is the implementation of the JSON data type of Kleppmann and Beresford [17]. 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 test. Instead we opted for ShareDB. We use Fabric.js [43] for the graphical interface.

Test setup. Both the clients and the server are deployed as separate Docker containers on a set of VMs in the Azure [40] public cloud. A VM has 4 vCPU cores and 8 GB of RAM (Azure Standard A4 v2) and can hold up to 3 client containers. A client container contains a browser which loads the client-side middleware from the server. The middleware server is deployed on a separate VM (Azure Standard F4s v2). The monitoring server that captures all performance data is also deployed on a separate VM. The Linux `tc` tool [3] is used to artificially increase the latency between the containers to an average of 60 ms with 10 ms jitter, which resembles the latency of a 4G network in the US. Other countries are pushing latencies down to 30 ms [48].

Our evaluation contains three benchmarks with different configurations. One benchmark represents the continuous online scenario where clients are making updates for 10 minutes and stay online the whole time. The second benchmark is the offline scenario where the network connection between the clients and the server is disrupted during the test. In total, we executed 60 tests for those two benchmarks: 6 tests to be executed by each of the 5 technologies, in both a continuous online setting as well as in a disconnected situation. These 6 tests 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 data set. One such object was shown in Figure 2a in Section 3 and has 7 attributes. Each client edits one object, which leads to two random writes on a shared object, every second (x and y position). In reality, a single update in the user-interface can lead to several writes to the data store, e.g. updating a gradient color would lead to 5 writes in Fabric.js [43].

We 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 [45]. But in practice, latency starts to increase significantly when the number of users exceeds 10 [8]. Our performance results show the same problem for ShareDB, which uses the same technique. In our performance evaluation, one iteration of a test takes about 10 minutes. Before each test, the database is populated and the initial synchronization will be performed. The first minute is used to execute a warm-up. Then we measure the performance of 9 minutes of continuous updates. To ensure stability and consistency of the test results, all tests are repeated 10 times [2].

The third and last benchmark is used to measure the total size of the data set over a longer time (2 hours). Every 10 minutes, 5 new client browsers will start making changes. After those 10 minutes, the browsers are shut down and replaced by others. After 2 hours, about 60 browsers of clients are introduced into the system. This benchmark simulates the eDesigners case study over the course of a few years, several employees and external consultants will have worked on the template using different browsers on their devices (desktop, laptop, tablet). In the meantime they might have cleared their browser cache, used an incognito session or switched to a new device. This scenario is used to verify how well the 5 frameworks will perform over time.

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. The synchronization times of the succeeded updates are illustrated in Figure 5.

Analysis of the results. For the test with 8 clients and 100 objects, all operation-based approaches (ShareDB, Yjs and Automerge) synchronize the updates faster than the state-based approaches (Legion and OWebSync). For these three operation-based approaches, 99% is below 0.3 seconds. Legion needs about 1.0 second for synchronizing the 99th percentile and OWebSync needs 1.3 seconds. The reason for this is that Legion and OWebSync don't keep track of which updates have been sent to which client. Hence, each time the data is synchronized, a few extra round-trips are required to calculate which updates are needed. ShareDB, Yjs and Automerge can just send the operations. On a faster network, with less latency, both Legion and OWebSync will be able

[2]Tables with the detailed performance results and the raw logs and data of all 60 tests are available on an anonymous Azure storage account: https://owebsyncdata.blob.core.windows.net/logs/data.zip

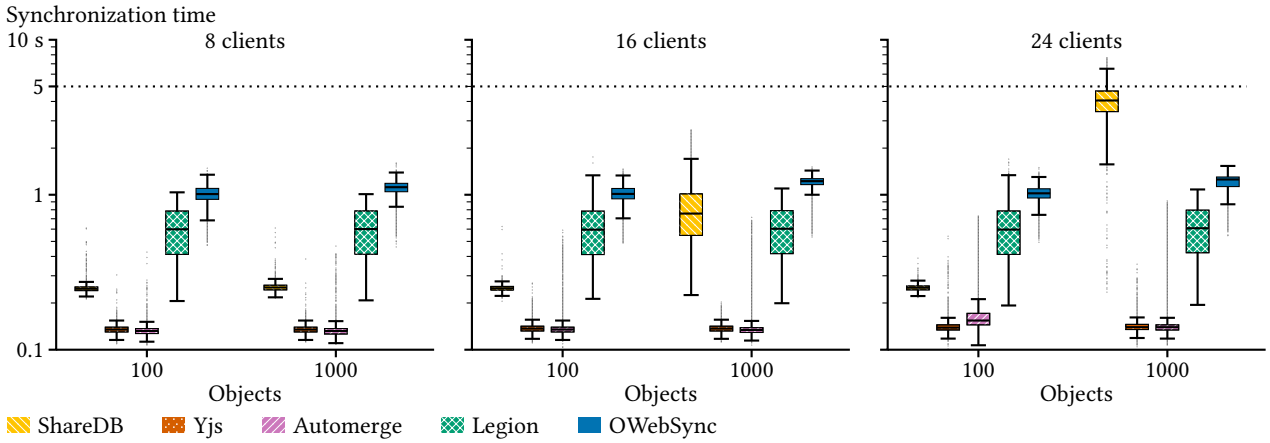

Figure 5. Aggregated boxplots containing the times to achieve full synchronization to all clients. Each boxplot contains all 10 iterations for each of the 30 tests in the fully online situation. In order to compare technologies that have results of the same order of magnitude, as well as results in different orders of magnitude, we opted for a logarithmic Y axis.

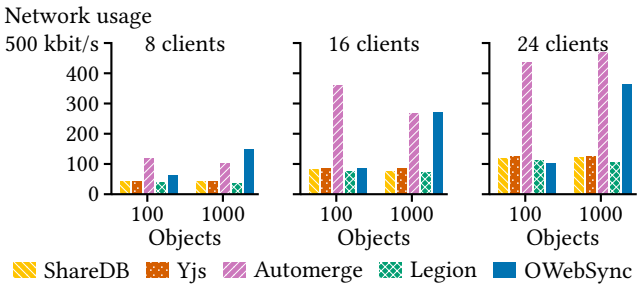

Figure 6. Network usage per client for each test. Some technologies use less bandwidth when the scale increases, because the time to synchronize increases even more.

to synchronize faster than in this test (since the round-trip time will be less). But even with this high latency in this benchmark, OWebSync performs within the guidelines of 1-2 seconds for interactive performance. For the test with 24 clients and 1000 objects, ShareDB has raised to 7.7 seconds for the 99th percentile. The server cannot keep up with transforming the incoming operations. Since the operations in Yjs and Automerge are commutative and don't need a transformation, the server doesn't become a bottleneck here.

Network trade-off. The trade-off for this scalable, prompt synchronization, is that OWebSync has a rather large network usage compared to the other tested technologies (Figure 6). Only Automerge requires more bandwidth, because it stores the whole history and uses long text-based UUIDs as client identifiers, compared to just integers with Legion. The usage of Merkle-trees reduced the network usage of OWebSync with about a factor 8 in the worst case (1000 objects under a single node in the tree), compared to normal state-based CRDTs. Introducing extra levels in the Merkle-tree for nodes with many children lowered the bandwidth with another factor 3. Even in the test with 24 clients and 1000 objects, the used bandwidth is only 360 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 [52]. The server consumes about 8.7 Mbit/s, which is acceptable for a typical data center. 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. This can be seen in Figure 6 by comparing the network usage of the tests with 100 objects to the tests with 1000 objects. The other possibility is that there are fewer 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 round-trips as there are more levels in the tree to go through.

Interpretation and discussion. For interactive web applications, usability guidelines [24, 25] 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 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* [9]).

The 99th percentile for the synchronization time for the OWebSync test with 24 clients and 1000 objects is below 1.5 seconds. ShareDB operates with sub-second synchronization times when sharing 100 objects between 8 writers. But when the number of objects and writers increases, the synchronization time raises to 7.7 seconds for the 99th percentile.

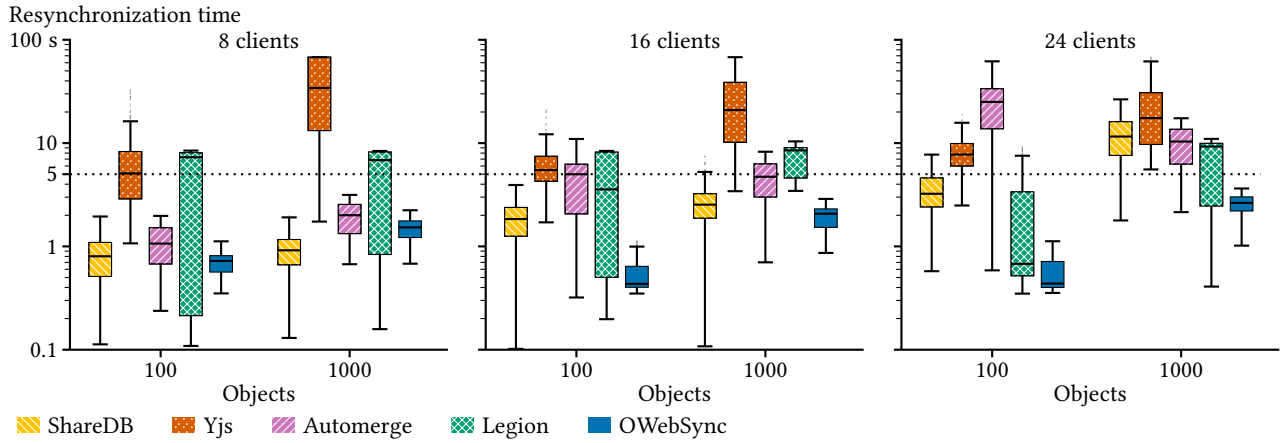

Figure 7. Boxplots of the time it takes for an update done during the failure scenario to be received by all clients. The time before a client notices the network connection is reestablished is not taken into account. Note that the median here means that only 50% of all missed updates are synchronized to all clients.

This is in line with the observations of Dang and Ignat [8] for Google Docs, which uses the same approach as ShareDB (OT). The other technologies stay well below 5 seconds in the online scenario and can be called interactive.

Performance in disconnected scenarios. We now present the performance analysis for the case when the network between the clients and the server is disrupted. In these tests, we have an analogous test setup. However, during the 10 minute execution, we start dropping all messages after 3 minutes (2 minutes in the graphs as the first minute is used as warm-up) for 1 minute. We evaluate the time that is needed to achieve full bidirectional synchronization of all concurrent updates on all clients during the network disruption. We also evaluate the time that is needed to restore normal interactive performance in the online setting after the disruption.

Analysis of the results. The boxplots of these tests (Figure 7) show that OWebSync can synchronize all missed updates faster than ShareDB, Yjs, Automerge and Legion. Note that at the median of the boxplots, only 50% of the missed updates is synchronized. Only at the upper whisker, all of the missed updates are fully synchronized. Then, each user is fully up-to-date with everything that was updated during the network disruption. In the large scale scenario with 24 clients and 1000 updates, the time to synchronize all missed updates in case of network failure is 3.5 seconds for the 99th percentile for OWebSync, which is acceptable for interactive web applications. The other technologies need more than 5 seconds to only synchronize half of the missed updates, meaning that users will become annoyed. The operation-based approaches need several tens of seconds to synchronize all of the missed updates because they need to replay all missed operations on the clients that were 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 network disruptions starts to take longer when the scale of the test or the size of the data set increases.

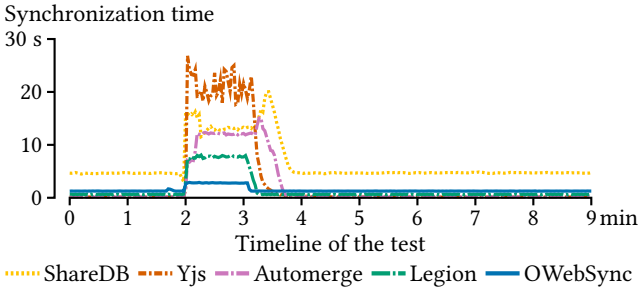

Figure 8. Mean time to synchronize updates after the network disruption for the test with 24 clients, 1000 objects.

Timeline analysis of the tests. The timelines in Figure 8 show the resynchronization times on the y-axis, without the offline time during the network disruption, for each update done at a given moment during the test timeline (x-axis). This means that for an update done 20 seconds before the end of the disruption, and which got synchronized 22 seconds later, the resynchronization time is 2 seconds.

In the test with 24 clients and 1000 objects (Figure 8), OWebSync quickly returns to the same performance as before the network disruption. Legion needs more time to synchronize the missed updates, but also quickly returns to the same performance. The operation-based approaches take much longer to synchronize missed updates, and take tens of seconds to return to the original performance. ShareDb and Automerge need more than half a minute to return to the same interactive performance as before. This means that in a setting with frequent disconnections, the user won't be able to gain interactive performance, since even when coming

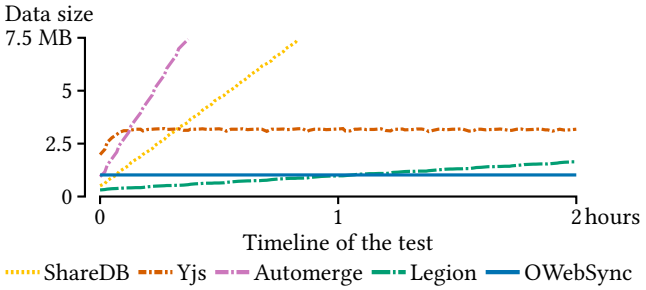

Figure 9. Evolution of the total data size on the server.

back online, those technologies cannot achieve prompt and interactive synchronization immediately.

Total size of the data model. All other technologies used in the evaluation use some form of client identifiers and version numbers to keep track of changes (e.g. vector clocks in Legion). This means that the size of the data set will grow over time, especially in highly dynamic settings like the web. Figure 9 shows the total data size on the server over time while several users are joining and leaving. The size of the data set on the server remains constant over time when using OWebSync. The other techniques grow with the number of clients and the number of operations. In the dynamic setting of the web, keeping track of all clients using version vectors with client identifiers will eventually inflate and pollute the metadata. Users can clear the browser cache, browse incognito or visit the web application on multiple devices including someone else's device for one time. By storing those client identifiers in the data model on the server, the performance will decrease over time. Yjs is an exception and stops growing fast in size after a few minutes. This is because Yjs will garbage collect old operations after 100 seconds [54]. While this limits the total size of the data, this operation is not safe and clients that were offline for a longer time might end up in an inconsistent state or lose data.

The first two benchmarks are performed on a clean data set, meaning that the size of the data on the server is still small. If we would start the tests after e.g. 5 hours of warm-up, the results for the other technologies would be worse. We performed the evaluation in a worst case scenario for OWebSync, with clean data sets for the other frameworks.

Summary. Our evaluation shows that the operation-based approaches work well in continuous online situations with a limited number of users. Operational Transformation cannot be used with many clients as the server eventually becomes a bottleneck. Operation-based approaches can synchronize updates faster than state-based approaches like Legion and OWebSync. However, when network disruptions occur, these technologies cannot achieve acceptable performance and need tens of seconds to achieve synchronization. Delta-state CRDTs, as used in Legion, can recover faster from network disruptions than operation-based approaches, but still need more than 5 seconds (8.6 s for the 99th percentile) to synchronize missed updates, which cannot be called interactive anymore. Moreover, the size of the data set will increase with the number of updates and the number of clients. OWebSync can achieve much better performance in the order of seconds, which is still acceptable for interactive web applications. In a setting with frequent offline situations, e.g. for mobile employees, OWebSync is the more appropriate technology and outperforms all other technologies. Over time, OWebSync can continue to deliver the same prompt and interactive performance, as no client identifiers or version vectors are stored. Table 1 summarizes the results in seconds of the large scale test (24 clients, 1000 objects) for the average user (50th percentile) and most of the users (99th percentile) for both the online and offline setting.

Table 1. Summary of the synchronization times in seconds for 24 clients and 1000 objects.

	online		offline	
	50%	99%	50%	99%
ShareDB	4.45	7.69	12.67	25.10
Yjs	0.14	0.17	20.21	109.15
Automerge	0.14	0.20	11.59	18.90
Legion	0.64	1.03	7.61	8.56
OWebSync	1.34	1.49	2.87	3.53

6 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 synchronization frameworks between data centers and 3) synchronization frameworks for replication to the client.

Concepts and techniques. The concepts and techniques like OT and CRDTs were discussed in Section 2. Other text-based versioning systems such as Git [44] are not made to manage data structures and do not always guarantee valid data structures after synchronization. Code, XML or JSON can end up malformed and often require user-level resolution.

We now discuss some other extensions to CRDTs. Conflict-free Partially Replicated Data Types [7] allow to replicate only part of a CRDT. This helps with bandwidth and memory consumption, as well as security. OWebSync allows to replicate any arbitrary sub-tree of the whole CRDT tree. Hybrid approaches combining operation-based and state-based CRDTs are also possible as demonstrated by Bendy [4]. For data that can tolerate staleness, one can make use of state-based CRDTs, while for data with interactive performance requirements, operation-based CRDTs can be used. This dynamic decision is only made between the servers, and not on the clients. For clients, only operation-based CRDTs are available, since they will never make enough updates to justify plain state-based CRDTs.

Distributed data systems and NoSQL systems. Based on the original Dynamo paper [9], many other open-source NoSQL systems have been developed for structured or semi-structured data, focusing on eventual consistency within or between data centers. Dynamo uses multi-value registers to maintain multiple versions of the data and expects application level resolution of conflicts. Cassandra [19, 41] supports fine-grained versioning of cells in a wide-column store. It uses wall-clock timestamps for each row-column cell, and adopts a last-write-wins strategy to merge two cells. CouchDB [42] and MongoDB [47] focus on semi-structured document storage, typically in a JSON format. CouchDB offers only coarse-grained versioning per document and stores multiple versions of the document. Applications need to resolve the version conflicts. Moreover, it also doesn't support fine-grained conflict detection within two JSON documents.

Several commercial database systems allow to use CRDTs as the underlying data model: e.g. Riak [50], Akka [37] and Redis [5]. Next to those commercial products, several research projects have emerged. Antidote [38] 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 for clients. SMAC [10] uses an operation-based CRDT storage system for state management tasks for distributed container deployments. DottedDB [14] uses node-wide dot-based clocks to find changes that need to be replicated, without the need for explicit tombstones. It doesn't support replication to the clients, or offline edits.

Client-tier frameworks for synchronization. Many client-side frameworks have appeared to enable synchronization between native clients. Cimbiosys [27] 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. SwiftCloud [26, 34–36] is a distributed object database with fast reads and writes using a causally-consistent client-side local cache and operation-based CRDTs. Metadata used for causality in the form of vector clocks are assigned by the data centers, so the size of the metadata is bound by the number of data centers, and not by the number of updates or the number of clients. The cache is limited in size and the data is only partially available, limiting what data can be read and updated during offline operation. Because it uses operation-based CRDTs, it needs a reliable exactly-once message channel, which is implemented by using a log.

Next to frameworks for native clients, there are several JavaScript frameworks made for synchronization between distributed web clients. Legion [32, 46] is a framework for extending web applications with peer-to-peer interactions. It also supports client-server usage and uses delta-state based CRDTs for the synchronization. Automerge [16, 39] is a JavaScript library for data synchronization adopting the operation-based JSON data type of Kleppman [17]. It uses vector clocks which grow in size with the number of clients. PouchDB [49] is a client-side JavaScript 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 [51] 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 is restored. The offline operations are lost when the browser session is closed. Yjs [22, 23, 54] 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 [53] is a JavaScript client library for the Swarm database, based on operation-based CRDTs with a partially ordered log for synchronization after offline situations. 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. None of these JavaScript frameworks support all of the following: fine-grained conflict resolution, interactive updates when online and fast resynchronization after being offline, as well as being scalable to tens of concurrently online clients and hundreds of writers over time.

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 Merkle-trees, which allows to quickly find differences in the data set and synchronize them to other clients. Apart from the regular CRDT structure and the hashes of the Merkle-tree, no other metadata needs to be stored. Other approaches use client identifiers and version numbers, or the full history to track updates, which will pollute the metadata and decrease performance over time.

The comparative evaluation shows that the operation-based approaches cannot achieve acceptable performance in case of network disruptions and need tens of seconds to achieve synchronization. Current state-based approaches using delta-state CRDTs are faster to recover than the operation-based ones, but cannot achieve prompt synchronization of missed updates. The state-based approach with Merkle-trees of OWebSync can achieve better performance in the order of seconds for both online updates and missed offline updates, 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. In *Networked Systems*. Springer International Publishing, Cham, 62–76. https://doi.org/10.1007/978-3-319-26850-7_5

[2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta state replicated data types. *J. Parallel and Distrib. Comput.* 111, Supplement C (2018), 162 – 173. https://doi.org/10.1016/j.jpdc.2017.08.003

[3] Werner Almesberger. 1999. Linux network traffic control – implementation overview.

[4] Carlos Bartolomeu, Manuel Bravo, and Luís Rodrigues. 2016. Dynamic Adaptation of Geo-replicated CRDTs. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16)*. ACM, New York, NY, USA, 514–521. https://doi.org/10.1145/2851613.2851641

[5] Cihan Biyikoglu. 2017. Under the Hood: Redis CRDTs (Conflict-free Replicated Data Types).

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

[7] Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. 2015. Conflict-free partially replicated data types. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 282–289.

[8] 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

[9] 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, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

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

[11] 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

[12] Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero, and João Leitão. 2019. Efficient Synchronization of State-based CRDTs. In *Proceedings of the 35th IEEE International Conference on Data Engineering*.

[13] Vitor Enes, Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2016. Join Decompositions for Efficient Synchronization of CRDTs After a Network Partition: Work in Progress Report. In *First Workshop on Programming Models and Languages for Distributed Computing (PMLDC '16)*. ACM, New York, NY, USA, Article 6, 3 pages. https://doi.org/10.1145/2957319.2957374

[14] R. J. T. Gonçalves, P. S. Almeida, C. Baquero, and V. Fonte. 2017. DottedDB: Anti-Entropy without Merkle Trees, Deletes without Tombstones. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 194–203. https://doi.org/10.1109/SRDS.2017.28

[15] Ian Hickson. 2012. *The WebSocket API, W3C Candidate Recommendation*. Technical Report. https://www.w3.org/TR/2012/CR-websockets-20120920/

[16] Martin Kleppman and Alastair R Beresford. 2018. Automerge: Real-time data sync between edge devices. http://martin.kleppmann.com/papers/automerge-mobiuk18.pdf

[17] 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.

[18] 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

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

[20] Paul Leach, Michael Mealling, and Rich Salz. 2005. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. https://www.rfc-editor.org/rfc/rfc4122.txt

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

[22] 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.

[23] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2016. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. In *Proceedings of the 19th International Conference on Supporting Group Work (GROUP '16)*. ACM, New York, NY, USA, 39–49. https://doi.org/10.1145/2957276.2957310

[24] Jakob Nielsen. 1993. *Usability Engineering*. Nielsen Norman Group. https://www.nngroup.com/books/usability-engineering/

[25] Jakob Nielsen. 2010. Website Response Times. https://www.nngroup.com/articles/website-response-times/.

[26] Nuno Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. 2014. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*. IEEE, 30–33.

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

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

[29] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel and Distrib. Comput.* 71, 3 (2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006

[30] 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.

[31] 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

[32] 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

[33] Albert van der Linde, João Leitão, and Nuno Preguiça. 2016. 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)*. ACM, New York, NY, USA, Article 12, 4 pages. https://doi.org/10.1145/2911151.2911163

[34] Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro, and Nuno Preguiça. 2013. *SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine.* Research Report RR-8347. INRIA. https://hal.inria.fr/hal-00870225

[35] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. 2015. *Write Fast, Read in the Past: Causal Consistency for Client-side Applications.* Research Report RR-8729. Inria. https://hal.inria.fr/hal-01158370

[36] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. 2015. Write Fast, Read in the Past: Causal Consistency for Client-Side Applications. In *Proceedings of the 16th Annual Middleware Conference (Middleware '15)*. ACM, New York, NY, USA, 75–87. https://doi.org/10.1145/2814576.2814733

[37] 2018. Akka. https://doc.akka.io/docs/akka/current/distributed-data.html.

[38] 2014. Antidote. http://syncfree.github.io/antidote.

[39] 2017. Automerge. https://github.com/automerge/automerge.

[40] 2019. Azure. https://azure.microsoft.com.

[41] 2009. Apache Cassandra. https://cassandra.apache.org.

[42] 2005. CouchDB. https://couchdb.apache.org.

[43] 2011. Fabric.js. https://github.com/fabricjs/fabric.js.

[44] 2005. Git. https://git-scm.com/.

[45] 2018. Google Docs. https://support.google.com/docs/answer/2494822.

[46] 2016. Legion. https://github.com/albertlinde/Legion.

[47] 2009. MongoDB. https://www.mongodb.com/.

[48] 2019. opensignal.com. https://www.opensignal.com/reports/2019/01/usa/mobile-network-experience.

[49] 2013. PouchDB. https://pouchdb.com.

[50] 2010. Riak. http://docs.basho.com/riak/kv.

[51] 2013. ShareDB. https://github.com/share/sharedb.

[52] 2018. Speedtest.net. http://www.speedtest.net/reports/united-states/2018/Mobile/.

[53] 2013. Swarm.js. https://github.com/gritzko/swarm.

[54] 2014. Yjs. https://github.com/y-js/yjs.