OWebSync: A web-based middleware for fluent and efficient data synchronization of distributed web clients

Kristof Jannes, Bert Lagaisse, and Wouter Joosen

imec-DistriNet, Dept. of Computer Science, KU Leuven, Belgium
kristof.jannes,bert.lagaisse,wouter.joosen@cs.kuleuven.be

Abstract. A lot of enterprise software services are adopting a fully web-based architecture for both internal line-of-business applications and for online customer-facing applications. Although wireless connections are becoming more ubiquitous and faster, mobile employees and customers are however not always connected. Nevertheless, continuous operation of the software services is expected.
This paper presents OWebSync: a web-based middleware framework for the continuous synchronization of online web clients and web clients that have been offline for a longer time period. OWebSync implements a fine-grained data synchronization model and leverages upon Merkle trees and convergent replicated data types to achieve the required performance both for online interactive clients, and for resynchronizing clients that have been offline.
OWebSync is validated and evaluated in two industrial use cases.

Keywords: Data synchronization · Offline web applications.

1 Introduction

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

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

1. support fine-grained and concurrent updates by distributed web clients on local copies of shared data,
2. operate conflict-free in both online and offline situations,
3. achieve continuous synchronization for online interactive clients and fluent resynchronization for offline clients.

A lot of distributed NoSQL data systems, e.g. Amazon Dynamo [11], adopt synchronization mechanisms based on Vector Clocks. This often lead to conflicts that need application-level resolving. Text-based versioning such as GIT does not always guarantee consistent data structures after synchronization. Code, XML or JSON documents can end up malformed and often require user-level resolution. Operational Transformation [20] approaches are often used for real-time synchronization (e.g. in Google Docs) but are not resilient against message loss in case of long-time offline situations [13]. Commutative Conflict-free Replicated Data Types [19] are also operation-based, but don't apply transformations to the operations. As such, the operations are commutative and can arrive and be applied in a different order. However, this technique also suffers when operations are lost. State-based Convergent Replicated Data Types [19] are resilient against message loss, but have often been considered as problematic with regards to the amount of data that has to be transfered between all distributed entities.

In this paper we present OWebSync, 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 near-realtime synchronization between online clients. Moreover, the middleware supports fluent resynchronization when clients were offline for a longer time, e.g. in case of client crashes or server crashes. OWebSync leverages state-based conflict-free replicated data types to support synchronization between clients and server even in the case of message loss. Merkle-trees [15] are used to enable more fluent synchronization of state-based CRDTs and limit the amount of data that has to be transfered. More specifically, OWebSync provides generic, reusable JSON [10] 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 evaluation shows that all clients receive updates within the timespan of seconds, even when tens of clients are editing hundreds of shared data items.

This paper is structured as follows. Section 2 provides two motivating case studies and then provides the rationale and more background on synchronization mechanisms such as CRDTs. Section 3 describes the generic, reusable JSON-based data types of OWebSync. Section 4 presents the deployment and runtime architecture of OWebSync. Section 5 evaluates performance in online and offline situations. We discuss related work in Section 6 and conclude in Section 7.

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

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

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

eDesigners offers a customer-facing multi-tenant web service to create, edit and apply graphical templates for mass communication based on the customer's company style. Templates can be edited by multiple users at the same time, even when they are offline. When two users edit the same document, a conflict occurs when the versions need to be merged. Edits that are independent of each other should both be applied to the template. For example, one edit can change the color of an object, another edit the size. When two users edit the same property of the same object, only one value can be saved. This should be resolved automatically.

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

Operational Transformation (OT). Operational Transformation [12] is a technique that is often used to synchronize concurrent edits on a shared document. For example, two clients can edit the text 'ABC' concurrently, where one client inserts '*' at position 1, and another client deletes the character at position 1.

The former results in 'A*BC', the latter in 'AC'. To achieve the correct state ('A*C'), the first client needs to transform the incoming operation of the other client to a deletion at position 2. This means the operation needs to be transformed to the current local state. The problem is that the transformation of the incoming operations of other clients on the local current state can get very complex, and that messages can get lost, or can arrive in the wrong order.

Conflict-free Replicated Data Types (CRDTs). CRDTs [19] are data structures that guarantee eventual consistency without the need for *explicit* conflict handling during synchronization by the application or user. Conflict-free thus means that conflicts are resolved automatically in a systematic and deterministic way, such that the application or user doesn't have to deal with conflicts themselves. There are two kinds of CRDTs: operation-based (Commutative Replicated Data Types) and state-based (Convergent Replicated Data Types).

Commutative Replicated Data Types (CmRDTs). CmRDTs make use of operations to reach consistency, just like Operational Transformation (OT). But the operations in CmRDTs are commutative and can be applied in any order. This way, there is no central server needed to apply a transformation on the operations. As with OT, CmRDTs need a reliable message channel so that every message reaches every replica exactly once [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 there is a least-upper-bound operation between two states. The least-upper-bound is the state that is larger or equal to both states according to the partial order. To merge two states, the least-upper-bound is computed and the result is the new state. CvRDTs don't require anything from the message channel, messages can get lost without a problem, since the whole state is always communicated. The main disadvantage is the fact that the state can get quite large, and needs to be communicated every time.

Merkle-trees. Merkle-trees [15] or hash-trees are used to quickly compare two large data structures. Each item in a data structure is hashed, and then the hashes are combined in a hash on top them, often in a binary way by combining two hashes from a lower level into a single hash at the higher level. This continues until the root of the tree is created with the top-level hash. Two data structures can now be compared starting from the two top-level hashes. If the root hashes match, the data structures are equal. Otherwise the tree can be descended using the mismatching hashes to find the mismatching items.

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

3 Convergent replicated data types with Merkle-trees

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

The CvRDT consist of two specific CvRDTs: a Last-Write-Wins Register (LWWRegister) [19] and an Observed-Removed Map (ORMap) [19]. 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 to the value. The data structure supports three operations: reading the value, updating the value and merging an LWWRegister with another one. The update operations also automatically updates the timestamp. The merging operation will always result in the value and timestamp of the latest update. The other value is lost.

Observed-Removed Map (ORMap). The Observed-Removed Map is typically implemented using an Observed-Removed Set (ORSet) with as contained data a tuple that contains a key and a value. We add an additional hash of the value that will be used to construct the Merkle-tree.

An ORSet is constructed with two *grow-only sets*. A grow-only set is also a CvRDT representing a set to which one can only add items. Such set can easily be merged with other grow-only sets by simply creating a union. The ORSet contains a grow-only set for the added items (observed set) and a grow-only set for the removed items (removed set).

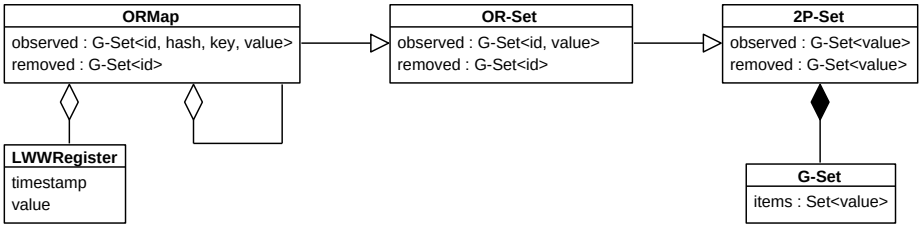

Fig. 1. Class diagram of the CRDTs that are used, including the CRDTs on which the ORMap is based. ORMap extends ORSet, which extends a Two-Phase Set (2P-Set). The 2P-Set contains two Grow-Only Sets (G-Set).

4 Web-based middleware architecture for synchronization

In this section we describe the deployment and execution architecture of the OWebSync middleware and 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 synchronization for online interactive clients.

Overall architecture. The middleware architecture is depicted in Figure 2 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. However, access control on the data is also supported and enforced at the server. Both the clients and server have a key-value store to make data persistent on disk. The many clients and server communicate using only web-based HTTP traffic. All communication messages between client and server are sent and received using asynchronous workers inside the client and server subsystems. We first further elaborate on the client-tier subsystem with the public middleware API for applications, and then describe the client-server communication protocol for synchronization in detail.

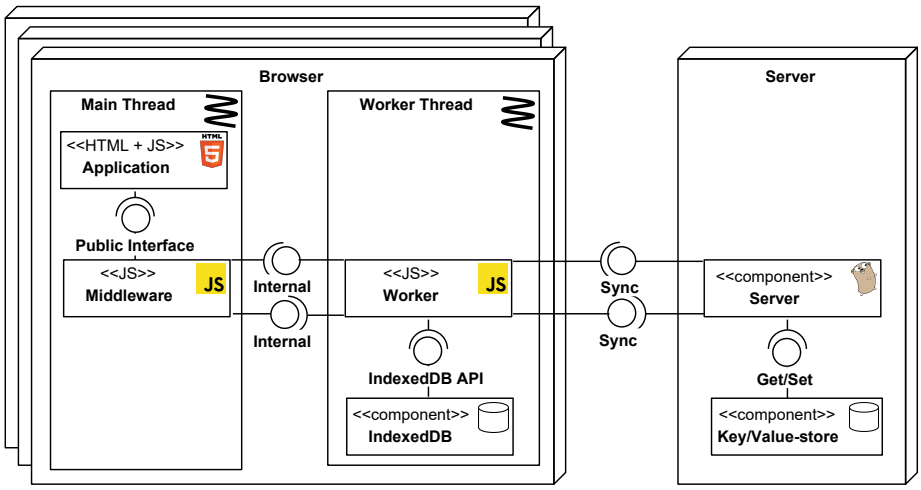

Fig. 2. Overall architecture of the OWebSync middleware

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 a GET, but every time the value changes, the callback is executed.
- SET(path, value): Create or update a value at a given path.

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

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

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

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

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

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

The process continues by traversing the tree and exchanges PUSH and GET messages until the leaf of the tree is reached. The CRDT in this leaf is a register and can be merged. 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 meanwhile, and the process is repeated.

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

Performance optimization tactics. The protocol 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 server.

Message batching. In the basic protocol explained above, all messages are sent to the other party as soon as a mismatch is detected. This leads to lots of small messages (GET, PUSH, and REMOVE) being sent out, and as a consequence, a lot of messages are coming in while still doing the first synchronization. This results in a lot of duplicated messages and doing a lot of duplicated work on subtrees. To solve this problem, all messages are grouped in a list and are sent out in batch after a full pass of the tree has occurred. At the other side, the messages are processed one by one, and all resulting messages are again grouped in a list, and then send out after the incoming batch was fully iterated. If no further messages are resulting from the processing of a batch, an empty list is sent to the other party. This ends the synchronization. As a result, a lot less messages are sent between a client and server, and only one synchronization is occurring at the same time, resulting in no duplicated messages and no duplicated work on subtrees.

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

HTTP websockets. The communication between client and server is initiated by a single HTTP message for the initial GET operation. All following messages in the synchronization protocol are sent over WebSockets within the context of the original HTTP message and connection. As such, a lot of overhead of HTTP headers and setting up TCP connections can be avoided.

5 Performance evaluation

The performance evaluation will focus on situations where all clients are continuously online, as well as on situations where clients go offline. We will perform most of the evaluation using the eDesigners case study, as this scenario has the largest set of shared data and objects between designers. Hence this serves as a worst case scenario. 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.

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 following failure scenarios. We assess situations where clients as well as the server can be offline. During this server disruption the clients continue doing updates on their local data copy.

Benchmark and test 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 can hold up to 3 client containers that generate load. 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.

We evaluated 9 benchmarks with different parameters regarding the scale of the tests: 8, 16, or 24 clients performing continuous concurrent updates on 10, 100 or 1000 objects. These objects are fully shared and replicated between the clients. Each client performs a random write on a shared object every second. Each test takes about 11 minutes. The first three minutes are used to populate the database, to perform the initial synchronization, and to execute a minute of warm-up. Then we measure the performance of 8 minutes of continuous updates. Each of the 9 benchmarks consists of 100 tests executed sequentially, and each benchmark thus runs for 1100 minutes.

Performance of continuous online updates. The following performance measurements quantify the statistical division of the time it takes to synchronize all clients in the case of continuous updates in an online situation. We also evaluate the progress over time to reach a certain percentage of the clients. This assesses how fast the first clients are getting an update compared to the last clients, and how this progresses in between.

For all of the 9 benchmarks we present the median synchronization time to achieve full synchronization in function of the number of clients (Figure 10) and in function of the number of objects (Figure 9). From the 9 benchmarks, we selected 3 of which we present the detailed performance: 1) a small scale scenario with 8 concurrent writers on 10 shared objects (Figure 6), 2) a medium scale scenario with 16 concurrent writers on 100 shared objects (Figure 5) and 3) our larger-scale scenario has 24 clients on 1000 shared objects that are all being edited concurrently (Figure 3).

Figure 4 shows the progress of synchronization over multiple clients, from the first client that receives the update to the last one. For brevity, we only provide this graph for the larger-scale scenario, as this has the most clients and objects to synchronize. As can be seen, there is a continuous, linear progress of the updates, and the last receivers do not cause major delays in general. However, as can be seen in the boxplots, during some tests, some updates took up to 4-5 seconds to achieve full synchronization.

All the benchmarks show consistent results during their 1100 minute run. The median time to get full synchronization is consistent over all 100 tests, in each benchmark. This is also clearly visible in the table in Figure 7 with the average means over all 100 tests in all 9 benchmarks. The standard deviation on the average mean[1] is clearly very limited. For each benchmark, the boxplots of the 100 tests actually show consistent medians, third quarters and maxima (Figure 3, 5 and 6).

[1] the average of the 100 means of the 100 tests in one benchmark

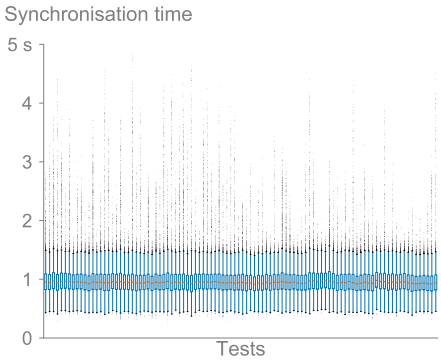

Fig. 3. Boxplots of 100 test executions in the larger-scale benchmark.

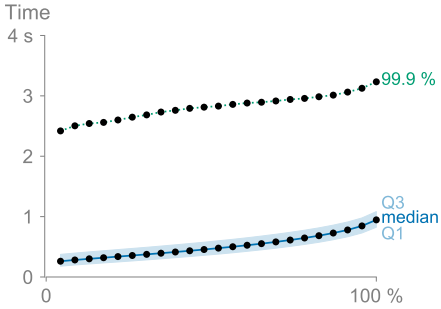

Fig. 4. Progress of synchronization in all 100 tests of the larger-scale benchmark.

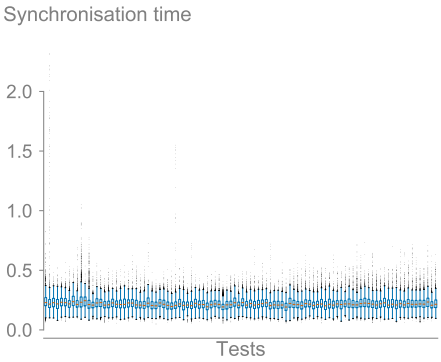

Fig. 5. Boxplots of 100 test executions in the medium-scale benchmark.

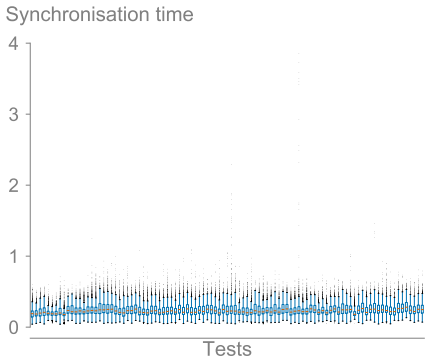

Fig. 6. Boxplots of 100 test executions in the small-scale benchmark

nb	8 *clients*	16 *clients*	24 *clients*
10	0.22 ± 0.02	0.21 ± 0.01	0.38 ± 0.05
100	0.19 ± 0.01	0.21 ± 0.01	0.33 ± 0.03
1000	0.40 ± 0.07	0.80 ± 0.05	0.94 ± 0.01

Fig. 7. For each benchmark, the average median synchronization time and standard deviation over 100 tests.

nb	8 *clients*	16 *clients*	24 *clients*
10	0.69 ± 0.05	0.75 ± 0.15	1.36 ± 0.21
100	0.40 ± 0.19	0.49 ± 0.21	1.48 ± 0.81
1000	1.22 ± 0.74	2.50 ± 0.97	2.53 ± 0.81

Fig. 8. For each benchmark, the average synchronization time of the 99.9 percentile and standard deviations.

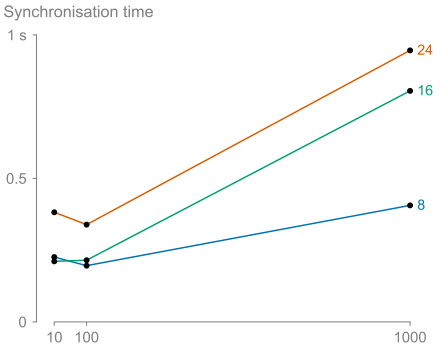

Fig. 9. Overview of the median synchronization times for the 9 benchmarks, in function of the number of shared objects

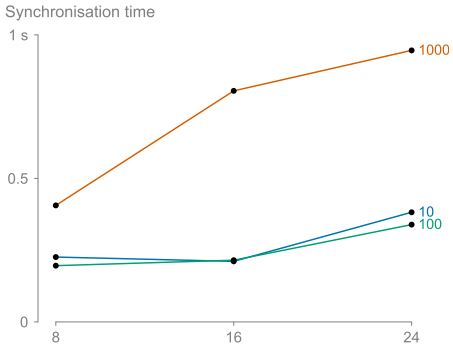

Fig. 10. For the 9 benchmarks, overview of the median synchronization times in function of the number of clients

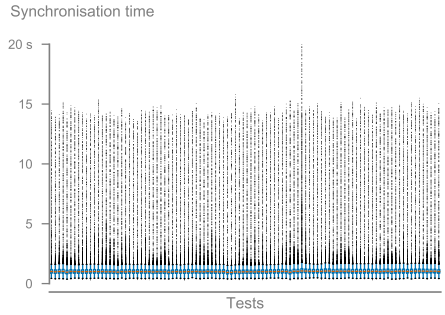

Fig. 11. Boxplots of 100 executions of the larger-scale performance test with a crash of a single client in each test.

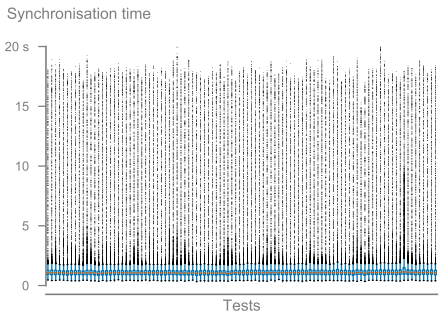

Fig. 12. Boxplots of 100 executions of the larger-scale performance test with a server crash in each test.

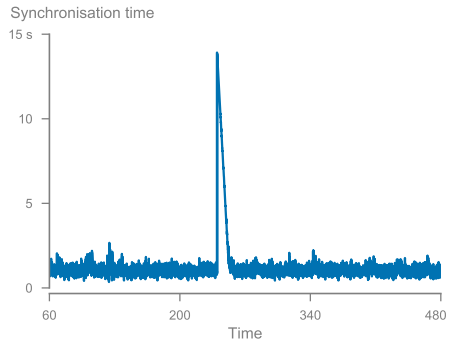

Fig. 13. Synchronization timeline of a single test with a client crash

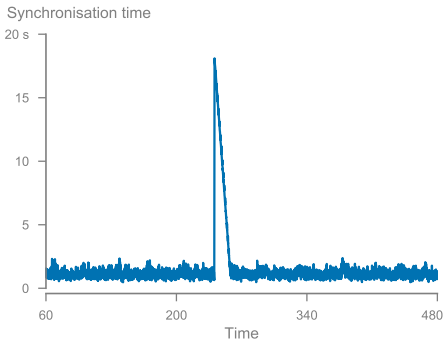

Fig. 14. Synchronization timeline of a single test with a server crash.

For interactive web applications, usability guidelines [17] state that a direct interaction should occur within 0.1 seconds. Remote response times should typically be 1 to 2 seconds in average. 3 to 5 seconds is the absolute maximum before users are annoyed. The user is often leaving the web application after 10 seconds of waiting time.

We start from these numbers to assess the update propagation time between users in a collaborative interactive online application with continuous updates. Local edits by a user on the local copy are of course within the 0.1 seconds limit for direct local interactions. We want to know 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 99.9 percentile (i.e. *most of the users* [11]). Figure 3 and the tables in Figure 7 and 8 show that the average mean for the larger-scale benchmark is well within 1 second. The third quarter is around 1.1 second, and the maximum without outliers is 1.5 seconds. The average synchronization time of the 99.9 percentile is at the border line of annoyance with $2.5 \pm 0.8s$. This is mainly due to the large number of objects that are shared between the concurrent writers, and not the number of concurrent writers. The synchronization time for 16 or 24 writers is very similar. The same applies to 10 or 100 objects. This can also be seen in Figure 10 and Figure 9.

In summary, over all benchmarks, the waiting time for an update of an online user was always below 5 seconds, even for the most extreme outliers. For those outliers, we are at the border line of user annoyance, but never even close the 10 seconds limit of user waiting time.

Failure scenarios. As the second part of this evaluation, we provide the performance measurements when a client or server goes offline. In these benchmarks, we have the same test setup, but in each 8 minute test we let a client or server crash after 4 minutes for 10 seconds and then restart it. For brevity, and to maximize the visibility of the impact, we only present this test for the larger scale benchmark with 24 clients and 1000 objects. The boxplots of the benchmarks with 100 tests (Figure 11 and 12) both show consistent results over all 100 tests. Both the mean synchronization times and the outliers are in the same ranges over all 100 tests during a 1100 minute benchmark.

In case of a client crash, the average offline time over the 100 tests was $12.502 \pm 0.322s$. The time to resynchronize the client after the offline period was $2.028 \pm 0.184s$. The statistical division of the synchronization times of all the updates by all the clients is displayed in Figure 15. The impact on the synchronization time of other clients was very minimal. The mean synchronization time was similar to the situation where all clients are online (about 1.005s). Of course the average synchronization time of the 99.9 percentile is much higher due to the offline client, as the synchronization time includes the 12 seconds down time. However, up to the 95 percentile, the synchronization times of updates in this failure scenario are in line with the fully online scenario.

In case of the server crash, all clients are offline and the average offline time over all tests was $11.675 \pm 0.248s$. This timespan includes the restart of the server

after the crash. The mean synchronization time over all updates was 1.067s, and the 95 percentile was still around 1.998s (Figure 15).

After the server restart, the median resynchronization time for all clients was $3.674 \pm 0.645s$. The 99.9 percentile of the resynchronization times was $6.039 \pm 0.652s$. Hence, after a server crash, most of the users were resynchronized in about 6 seconds.

In summary, the resynchronization time in case of a client crash or server crash is acceptable for interactive online web applications. Moreover, the overall impact on all clients is minimal as can be seen in the timeline graphs in Figure 13 and 14. After a short peak in synchronization times of updates, mainly during the crashes, everything quickly turns back to the average synchronization times.

stats	client offline	server offline
mean	$1.005 \pm 0.023s$	$1.067 \pm 0.021s$
90%	$1.403 \pm 0.196s$	$1.553 \pm 0.171s$
95%	$1.739 \pm 1.047s$	$1.998 \pm 0.518s$
99%	$9.684 \pm 2.978s$	$13.371 \pm 0.608s$
99.9%	$14.124 \pm 3.159s$	$17.506 \pm 0.634s$

Fig. 15. Synchronization times of all updates in the failure scenarios.

6 Related work

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

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

Dynamo [11] for example is a highly-available key-value store developed and used by Amazon in their data centers. The focus is to support high-availability of write operations. Applications should always be able to write on the local copy. In case conflicts occur between different versions (in the case of network partitions), the reconciliation occurs when the data item is read later. Using syntactic reconciliation, Dynamo can resolve the conflict between a newer version and older version if a newer version is clearly derived from the older version. In

case concurrent writes occurred, Dynamo relies on the application to merge the two versions (semantic reconciliation). Dynamo can thus not merge two versions of complex objects that are stored as values in the key-value store.

Based on the original Dynamo paper, a lot of other open-source NoSQL systems have been developed for structured or semi-structured data. Cassandra [2] [14] supports fine-grained versioning of cells in a wide-column store. It therefore uses timestamps for each row-column cell, and adopts a last-write-wins strategy to join two cells. CouchDB [3] and MongoDB [4] 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 [6] 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 [1] is a research project to develop a geo-replicated database over world-wide data centers. It adopts operation-based commutative CRDTs for highly-available transactions. It supports partial replication but assumes continuous online connections as the default operational situation.

Client-tier JavaScript-libraries for synchronization. A lot of JavaScript frameworks have appeared to enable synchronization between web browsers and server-side data systems. PouchDB [5] 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 [7] is a client-server framework to synchronize JSON documents and adopts Operational Transformation as synchronization technique between the different local copies. ShareDB can thus not be used in 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 [16, 9] 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 [8] 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.

7 Conclusion

This paper presents a web middleware that supports the fluent synchronization of both online and offline clients that are concurrently editing shared data sets. We proposed a JSON-based data model for the web middleware and a supporting

synchronization architecture. The solution is resilient for message loss and out-of-order messages.

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

Our benchmarks demonstrate that we were able to scale up to scenarios with 24 clients concurrently updating 1000 objects while still achieving fluent interactive synchronization.

References

1. Antidote. http://syncfree.github.io/antidote
2. Apache cassandra. https://cassandra.apache.org
3. Couchdb. https://couchdb.apache.org
4. Mongodb. https://www.mongodb.com/
5. Pouchdb. https://pouchdb.com
6. Riak. http://docs.basho.com/riak/kv
7. Sharedb. https://github.com/share/sharedb
8. Swarm.js. https://github.com/gritzko/swarm
9. Yjs. https://github.com/y-js/yjs
10. Bray, T.: The javascript object notation (json) data interchange format. RFC 7158 (2014), https://www.rfc-editor.org/rfc/rfc7158.txt
11. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: ACM SIGOPS operating systems review. vol. 41, pp. 205–220. ACM (2007). https://doi.org/10.1145/1294261.1294281
12. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. SIGMOD Rec. **18**(2), 399–407 (Jun 1989). https://doi.org/10.1145/66926.66963
13. Kumawat, S., Khunteta, A.: A survey on operational transformation algorithms: Challenges, issues and achievements. International Journal of Computer Applications **3**(12), 30–38 (2010). https://doi.org/10.5120/787-1115
14. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review **44**(2), 35–40 (2010)
15. Merkle, R.: Method of providing digital signatures
16. Nicolaescu, P., Jahns, K., Derntl, M., Klamma, R.: Yjs: A framework for near real-time p2p shared editing on arbitrary data types. In: Engineering the Web in the Big Data Era. pp. 675–678. Springer (2015)
17. Nielsen, J.: Response time limits. Accessed cited 21Aug (2012)
18. Shapiro, M., Perguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems. Lecture Notes in Computer Science, vol. 6976, pp. 386–400. Springer (Oct 2011)
19. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of convergent and commutative replicated data types. Research Report RR-7506 (Jan 2011), https://hal.inria.fr/inria-00555588
20. Sun, C., Ellis, C.: Operational transformation in real-time group editors: issues, algorithms, and achievements. In: Proceedings of the 1998 ACM conference on Computer supported cooperative work. pp. 59–68. CSCW '98, ACM (1998)