OWebSync: Seamless Synchronization of Distributed Web Clients

Kristof Jannes, Bert Lagaisse and Wouter Joosen

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 middleware for data synchronization in interactive groupware with fast resynchronization of offline clients and continuous, interactive synchronization of online clients. To automatically resolve conflicts, OWebSync implements a fine-grained data synchronization model and leverages state-based Conflict-free Replicated Data Types. This middleware uses Merkle-trees embedded in the tree-structured data and virtual Merkle-tree levels to achieve the required interactive performance. Our comparative evaluation with available operation-based and delta-state-based middleware solutions shows that OWebSync is especially better in operating in and recovering from offline settings and network disruptions. In addition, OWebSync scales more efficiently over time, as it does not store version vectors or other meta-data for all past clients.

Index Terms—CRDTs, Groupware, Web browsers, Eventual Consistency

1 Introduction

Web applications are the default architecture for many online software services, both for internal line-of-business applications such as Customer Relationship Management (CRM), billing, and Human Resources (HR); as well as for customer-facing services. Browser-based service delivery fully abstracts the heterogeneity of the clients, solving the deployment and maintenance problems that come with native applications. Nevertheless, native applications are still used when rich and highly interactive GUIs are required, or when applications must function offline for a long 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. In reality, connectivity is often missing for minutes to hours. Mobile employees can be working in cellars or tunnels, and customers sometimes want to use a web-based service on an airplane.

Interactive groupware applications, such as collaborative web applications with concurrent edits on shared data, should offer prompt data *synchronization* with *interactive* performance when online. We use the term synchronization here to describe the process of keeping data of multiple replicas eventually consistent by means of replication.

This paper focuses on prompt and seamless synchronization when clients were offline due to network disruptions, while maintaining interactive synchronization in the online setting. The research of Nielsen on usability engineering [1] states that remote interactions should take only 1-2 seconds to keep the user experience seamless and interactive. Users are annoyed after a 5 second waiting period and 10 seconds is the absolute maximum before users leave the application.

Several client-side frameworks for synchronization of semi-structured data exist. They support fine-grained and concurrent updates on local copies of shared data and operate conflict-free in online and offline situations. However, there is no generic, fully web-based middleware solution that can be used by interactive web applications to:

1) achieve continuous and interactive synchronization for online clients and prompt resynchronization for offline clients,
2) scale to tens of online clients that concurrently edit a document with interactive performance,
3) tolerate hundreds of clients over time without inflating the data with versioning metadata.

State-of-the-art data synchronization frameworks are either operation-based, state-based or delta-state-based. Operation-based approaches distribute the updates as operations to all replicas. Operational Transformation, as used in Google Docs [2], 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 requires a central server transforming the operations for other clients to deal with concurrent changes. Commutative Replicated Data Types [3], [4], as used in SwiftCloud [5], [6], Yjs [7], [8], [9] and Automerge [10], [11], are also operation-based. Again, updates must be propagated, as operations, to all clients using a reliable, exactly-once, message channel. However, no transformation is needed because concurrent operations are commutative. State-based Convergent Replicated Data Types [4] are resilient against message loss, but have often been considered as problematic since the full state has to be transferred between all replicas each time. However, it is used for background synchronization between data centers, e.g. in Riak [12]. Merkle Search Trees [13] are proposed as a solution to the high bandwidth usage. It uses Merkle-trees [14] to replicate a basic key-value store like in Dynamo [15]. The solution works in large systems with low rates of updates for asynchronous background

• *The authors are with imec-DistriNet, KU Leuven, 3001 Leuven, Belgium. E-mail: {kristof.jannes, bert.lagaisse, wouter.joosen}@cs.kuleuven.be*

synchronization between backend servers; it is not suited for interactive groupware. Delta-state-based Conflict-free Replicated Data Types [16], as used in Legion [17], [18], need less of the message channel than the operation-based approaches. However, they use vector clocks to calculate delta-updates, which require one entry per writing client per object in the server-side metadata. This does not integrate well with the dynamic nature of the web, where it is often uncertain if a client will ever connect to a server again.

In this paper, we present OWebSync, a generic web middleware for data synchronization in browser-based applications and interactive groupware. It supports offline usage with fast resynchronization, as well as continuous and interactive synchronization between online clients. OWebSync provides a generic, reusable data type, based on JSON [19], that web application developers 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 by leveraging state-based Conflict-free Replicated Data Types (CRDTs). OWebSync solves the scalability issue that comes with operation-based approaches, where server-side metadata will grow linearly over time with the number of clients present in the system at some point. It reduces the required bandwidth by combining several tactics such as Merkle-trees embedded in the tree-structured data, virtual Merkle-tree levels, and message batching. As such, OWebSync can achieve the interactive performance of operation-based approaches, while maintaining the inherent *robustness* of state-based approaches.

This paper is structured as follows. Section 2 provides two motivating case studies and provides background on synchronization mechanisms such as CRDTs. Section 3 describes the underlying data model based on CRDTs and Merkle-trees. Section 4 presents the deployment and synchronization architecture together with two performance optimization tactics. 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.

2 Motivation and Background

This section explains the motivation of the goal and approach of OWebSync. First, we present two case studies of online software services for 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.

2.1 Case studies

The motivation and requirements emerged from two case studies from our applied research projects with industry, 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, eDesigners, offers a web-based design environment for graphical templates that are applied to mass customer communication.

2.1.1 *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 must be able to indicate which materials they used for a job. Since they are always on the road, a stable internet connection is not always available. Moreover, they often work in offline mode when they work in basements to install hardware. Writing off all used materials is crucial for correct billing and inventory afterwards.

This case study requires support for 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 must be operational at all times, even without connection to the central database, as customers can call for support and schedule interventions.

2.1.2 *eDesigners*

The company eDesigners offers a customer-facing multi-tenant web application 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, updates must always be possible, even offline when working on an airplane. When coming back online, the updates should be synchronized promptly without requiring the user or the application to manually resolve conflicts. When working online, the performance should be interactive, especially when two users are working on the same template next to each other.

2.2 Background

The previous section described the overall goal. In this section, we discuss the advantages and problems of state-of-the-art techniques such as Operational Transformation and Conflict-free Replicated Data Types (CRDTs).

2.2.1 *Operational Transformation*

OT [20] is a technique that is often used to synchronize concurrent edits on a shared document. It 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. 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 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 and long-lasting offline situations [21].

2.2.2 Conflict-free Replicated Data Types

CRDTs [4], [22] 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 does not 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 [22] make use of operations to reach consistency, just like OT. Concurrent operations in CmRDTs must be commutative and can be applied in any order. This way, there is no central server necessary 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 [22] 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. This means that there is a partial order defined over the possible states and 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, which will be 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 sent. However, this state can get large, and needs to be communicated every time.

Delta-state CvRDTs. δ-CvRDTs [23], [24], [25] 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, it 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 the client already had most of the updates.

A variant of δ-CRDTs, called Δ-CRDTs [16], is proposed as a 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, as 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 [26], [27]. This approach does not 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 neighbors. When there is no such data available, e.g. after a network partition, it needs to fall back 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 does not exist and for large, nested data, this digest would still be large.

2.3 Principles

We now introduce two state-based CRDTs and Merkle-Trees. We will use these as building blocks in the next section for our data model.

2.3.1 LWWRegister

A Last-Write-Wins Register (LWWRegister) [4] is a CvRDT that contains exactly one value (string, number or boolean) together with a timestamp of the last change. This timestamp will be used to merge another replica of this LWWRegister. The value associated with the highest timestamp is kept, while the other value is discarded. This conflict resolution strategy boils down to a simple last-write-wins strategy.

2.3.2 ORSet

An Observed-Removed Set (ORSet), as described by Shapiro et al. [4], [28], is a set CvRDT. Internally, the ORSet contains two sets: the observed set and the removed set. When an item is added to the set, it is added to the observed set together with a unique tag. When that item is removed, the associated tag is added to the removed set, and the item itself is removed from the observed set. This allows an item to be removed and added multiple times. All items present in the observed set, but not in the removed set are currently present in the set. The conflict resolution of the ORSet 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. To merge a local replica of an ORSet with another replica, the union of the respective observed and removed set is taken, and removed items are removed from the observed set.

2.3.3 Merkle-trees

Merkle-trees [14] or hash-trees are used to quickly compare two large data structures. Merkle-trees are trees where each node contains a hash. The values of the leaf nodes are hashed and each hash in an internal node is the hash of the hashes of all its immediate children. 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. Sub-trees which are already equal will have equal hashes at their top nodes, so they do not need further verification.

3 The OWebSync Data Model

This section describes the data model of OWebSync that will be used for synchronization. The data model is a CvRDT for the efficient replication of JSON data structures and applies Merkle-trees internally to quickly find data changes.

3.1 Approach

OWebSync uses state-based CRDTs, which require little from the message channel compared 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. The CvRDT consist of two types: a LWWRegister and an ORMap extended with a Merkle-tree. The LWWRegister is used to store values in the leaves of the tree, and is implemented as described by Shapiro et al. [4]. The ORMap is a recursive data structure that represents a map containing a mapping from strings to other ORMaps or LWWRegisters.

3.2 Observed-Removed Map

The ORMap is implemented starting from a state-based Observed-Removed Set [28]. The items in the observed set are key-value pairs, where a key is a string, and the value is a reference to another CvRDT. Concurrent edits to different keys can be made without a problem. Edits to the same key and tag will be delegated to the child CRDT: either another ORMap or a LWWRegister. If two different replicas add the same key, they will get a different tag. This situation is difficult to resolve, and we opted to merge the two values, keeping only the lexicographical greatest tag. As a result, a single replica of an ORMap has at most one value for a key.

We made two extensions to this basic ORMap to make state-based synchronization more efficient. First, we extended this data structure with a Merkle-tree using the object's logical tree-structure. This means that we keep an extra hash for all items in the observed set. When the child is a LWWRegister, the hash is the hash 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 its children, lexicographically ordered on the unique tags. This way, when one value in a register changes, all the hashes of the parents will also change, so that a change can be detected by only comparing the top-level hash. Second, we do not store a child CRDT inside the observed set, instead we only store the tag, key and hash of that CRDT. The child CRDTs can be stored elsewhere using its path as a unique key.

Alg. 1 and 2 show the specification of the OWebSync ORMap with our two extensions. It supports several operations to query, update and merge this data structure. The

```
1:  $KV$                                   ▷ Key-value store
2:  $p_0..p_n$          ▷ Array representation of a path in the tree
3:  state:
4:      $O \leftarrow \emptyset$      ▷ Observed set with tuples $(key, tag, hash)$
5:      $R \leftarrow \emptyset$                        ▷ Removed set with $tag$s
6:      $PATH$                              ▷ The path of this ORMap
7:  query: GET $(p_0..p_n)$
8:      if $\exists o \in O : o.key = p_0$ then
9:          $c \leftarrow KV.\text{GET}(PATH + o.key)$
10:         return $c.\text{GET}(p_1..p_n)$
11:     return $\perp$
12: update: SET $(p_0..p_n, value)$
13:     if $\exists o \in O : o.key = p_0$ then
14:         $c \leftarrow KV.\text{GET}(PATH + o.key)$
15:         $c.\text{SET}(p_1..p_n, value)$
16:         $o.hash \leftarrow c.hash$
17:     else
18:         if $\text{LEN}(p_0..p_n) = 1 \wedge$ IS_PRIMITIVE$(value)$ then
19:             $c \leftarrow$ NEW_LWWREGISTER$(PATH + p_0)$
20:         else
21:             $c \leftarrow$ NEW_ORMAP$(PATH + p_0)$
22:         $c.\text{SET}(p_1..p_n, value)$
23:         $O \leftarrow O \cup \{(p_0, \text{UNIQUE}(), c.hash)\}$
24: update: REMOVE $(p_0..p_n)$
25:     if $\exists o \in O : o.key = p_0$ then
26:         if $\text{LEN}(p_0..p_n) = 1$ then
27:             $O \leftarrow O \setminus \{o\}$
28:             $R \leftarrow R \cup \{o.tag\}$
29:         else
30:             $c \leftarrow KV.\text{GET}(PATH + o.key)$
31:             $c.$REMOVE$(p_1..p_n)$
32:             $o.hash \leftarrow c.hash$
33: update: REMOVE_WITH_TAG $(p_0..p_n, tag)$
34:     if $\text{LEN}(p_0..p_n) = 0$ then
35:         if $\exists o \in O : o.tag = tag$ then
36:             $O \leftarrow O \setminus \{o\}$
37:             $R \leftarrow R \cup \{o.tag\}$
38:     else if $\exists o \in O : o.key = p_0$ then
39:         $c \leftarrow KV.\text{GET}(PATH + o.key)$
40:         $c.$REMOVE_WITH_TAG$(p_1..p_n, tag)$
41:         $o.hash \leftarrow c.hash$
42: (Continues in Alg. 2)
```

Alg. 1. Simplified implementation of the state, query- and update operations of an ORMap with a Merkle-tree for synchronization.

`GET` operation is equal to the one in a basic ORMap. There is always at most one single object in the observed set with a specific key. The `SET` and `REMOVE` operation are also similar, but require updating the hash to keep the Merkle-tree up-to-date. The `REMOVE_WITH_TAG` operation removes a single element, based on the tag instead of the key.

The `MERGE` operation is modified to make use of the Merkle-tree. It accepts a path in the tree, and the ORMap of that path from the remote replica. The received ORMap only contains the metadata of its children, and not the actual child CRDTs. The `MERGE` will detect which branches of the tree are changed, and returns all paths of those branches. In a next step, the synchronization protocol will use those re-

```
43: merge: MERGE ($p_0..p_n$, $remote$)
44:     $N \leftarrow \emptyset$                ▷ paths that need synchronization
45:     if LEN($p_0..p_n$) $= 0$ then
46:         $R \leftarrow R \cup remote.R$
47:         $O \leftarrow \{o \in O : o.tag \notin R\}$
48:         for all $ro \in remote.O : ro.tag \notin R$ do
49:             if $\exists o \in O : o.key = ro.key$ then
50:                 if $o.tag \neq ro.tag$ then
51:                     if $ro.tag > o.tag$ then
52:                         $o.tag \leftarrow ro.tag$
53:                     else
54:                         $N \leftarrow N \cup \{PATH\}$
55:                 if $o.hash \neq ro.hash$ then
56:                     $N \leftarrow N \cup \{PATH + o.key\}$
57:             else
58:                 $N \leftarrow N \cup \{PATH + ro.key\}$
59:     else
60:         if $\exists o \in O : o.key = p_0$ then
61:             $c \leftarrow KV.$GET$(PATH + o.key)$
62:             $N \leftarrow N \cup c.$MERGE$(p_1..p_n, remote)$
63:             $o.hash \leftarrow c.hash$
64:         else
65:             if LEN($p_0..p_n$) $= 1$
66:                 $\wedge$ TYPEOF($remote$) $=$ LWWRegister then
67:                 $c \leftarrow$ NEW_LWWREGISTER$(PATH + p_0)$
68:             else
69:                 $c \leftarrow$ NEW_ORMAP$(PATH + p_0)$
70:             $N \leftarrow N \cup c.$MERGE$(p_1..p_n, remote)$
71:             $O \leftarrow O \cup \{(p_0, c.tag, c.hash)\}$
72:     return $N$
```

Alg. 2. Simplified implementation of the merge operations of an ORMap with a Merkle-tree for synchronization.

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

Fig. 1. JSON data structure of the eDesigners case study.

```
- drawing1.object36:
  {
    tag: 0a2f7bc2-129f-11e9-ab14-d663bd873d93,
    hash: 7319eae53558516daafac19183f2ee34,
    observed: [
      {
        key: "top",
        tag: 23c1259a-129f-11e9-ab14-d663bd873d93,
        hash: 65bdd1b610f629e54d05459c00523a2b
      },
      {
        key: "left",
        tag: 0eac2a3a-546f-11e9-8647-d663bd873d93,
        hash: 67507876941285085484984080f5951e
      },
      ...
    ],
    removed: []
  }
- drawing1.object36.top:
  {
    tag: 23c1259a-129f-11e9-ab14-d663bd873d93,
    hash: 65bdd1b610f629e54d05459c00523a2b,
    timestamp: 789778800000,
    value: "100"
  }
```

Fig. 2. Internal structure of two key-values that represent `object36` and the property `top` of the JSON data structure in Fig. 1. We use a pseudo-JSON notation here, however, in practise these two key-values are stored in a binary format in a key-value store.

turned paths to descend in the tree and continue the MERGE in these branches. Only the returned paths are merged further, the other branches of the tree do not need further processing. By splitting up this operation per level in the tree, only the updated registers and parent ORMaps need to be sent over the network, improving both the bandwidth usage as well as saving computation power as not all CRDTs need to be merged. We explain this synchronization protocol in more detail in Section 4. We use a key-value store to store the CRDTs, called KV in the specification.

Proof sketch. A state-based object is a CvRDT when the states of that object forms a monotonic join semilattice. This means that there is partial order defined over the states, and a least upper bound (LUB) operation on two states which results in the smallest state that is larger or equal to the two given states according to the partial order [4], [22]. The partial order of the modified ORMap defined here is similar to the ORSet [28], which contains two grow-only sets. As an optimization, removed items are only present in the removed-set and are removed from the observed-set, however, conceptually they are still part of the observed-set when determining the partial order. The LUB operation, equal to the MERGE operation in Alg. 2, takes the union of the respective observed- and removed-set. Again using the optimization that removed items are not actually stored in the observed-set anymore. Two complications added here are the key and the hash. When a key is present in both ORMaps, with a different tag, the LUB operation will only keep the largest tag, and merge the two values according to the rules of the child CRDT. The other tag is considered removed. When the hash differs, the two values are merged according to the rules of the child CRDT, after which the hashes will become equal. A new item in the remote observed-set is not immediately added to the local observed-set, instead the path of that branch is returned. Later, MERGE will be called with the child, and the item is added to the observed-set. This addition is delayed to make it possible to infer the type of the child CRDT.

Example. As an example, we illustrate the conceptual representation of an application data object in the eDesigners case study, as well as the resulting underlying CRDTs in the OWebSync data model. Fig. 1 present a JSON data structure of a drawing with one rectangle object. Fig. 2 represents the internal structure of two CRDTs in that JSON structure. First, the key under which the CRDT is stored in a key-value store is listed, then the internal value of the CRDT. There is an ORMap stored in the key-value store for key `""` (the root of the tree), `drawing1` and `drawing1.object36`. Only `drawing1.object36` is shown in the figure. For all the leaf-values, there is a LWWRegister stored under the respective keys, for conciseness, only `drawing1.object36.top` is shown. The application developer only needs to know about the conceptual JSON representation, the middleware will automatically translate this data model and its operations to the underlying CRDTs and maintain the Merkle-tree and the internal CRDT structure. When a user modifies the `top` property, its hash and the hash of all the parents will change. The MERGE procedure will be called with $p_0..p_n$ empty and return the branches that have changed: {`drawing1`}. MERGE will now again be called with p_0 = `"drawing1"` and the respective remote CRDT, and will return {`drawing1.object36`}. This process will continue until it reaches a leaf value.

3.3 Considerations and discussion

The data model is best suited for semi-structured data that is produced and edited by concurrent users. Any data that can be modeled in a tree-like structure such as JSON and that can tolerate eventual consistency, can use OWebSync for the synchronization. Examples are the data items in the case studies: graphical templates, a set of tasks or used materials for a task. This data model is less suited for applications such as online banking which requires constraints on the data 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 and one level with many child nodes: the characters. There is no benefit in using a Merkle-tree here.

Developers have two choices. They can either pass a JSON object, and every JSON map will be mapped to an ORMap, and the leaf values to LWWRegisters. Or they can *stringify* an object, so that the full object is mapped to a LWWRegister. As a result, changes will be atomic.

The timestamps in the LWWRegisters are provided by the clients and we do not consider malicious clients. We also assume loosely synchronized clocks. If this assumption is not met, an accidental fault resulting in a clock several years in the future, might make edits to this LWWRegister impossible. Users can resolve this manually by removing that register and creating a new one with the same key in the ORMap, loosing concurrent changes. We do not consider this an important drawback, as most personal devices these days automatically synchronize their time with the internet.

In the current data model, the ORMap keeps the tags 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 problem. Even when you remove a large sub-tree of several levels deep, only the tag of its root is kept in the parent. One strategy to clean up tombstones could be to remove those older than e.g. 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 logging out the user after a month of no usage. Delta-state-based CRDTs can avoid tombstones by encoding the causal context as a compact version vector. However, this version vector grows in size with the number of clients that make changes to this ORMap. We opted for tombstones, which can be garbage collected after a sufficiently long time, because we target a dynamic environment such as the web. Web clients come and go, without long term presence.

Another kind of conflict occurs when two different types of CRDTs are assigned concurrently at the same position in the JSON structure. In this case, the merge-operation of the defined CRDTs cannot be used to resolve the conflict. 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.

Beside primitive values and maps, the JSON specification also contains ordered lists. This is currently not supported by OWebSync. We focused on the initial key data structures: last-write-wins registers and maps. Keeping a total numbered order, as lists do, is rarely needed. Unique IDs in a map are better suited in a distributed setting. In the case studies, the ordering of items in a set was based on application-specific properties such as dates, times or other values, instead of an auto-incremented number of a list. However, CvRDTs for ordered lists exist [4], [29] and could be added in future work. Adding new kinds of CRDTs to the data model is straight-forward. 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 [4]), the hash is simply the hash of that value. For CRDTs that can contain other values, the hash must combine the hashes of all the children.

4 Architecture and Synchronization

This section describes the deployment and execution architecture of OWebSync as well as the synchronization protocol. This middleware architecture is key to support the data- 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.

4.1 Overall architecture

The middleware architecture is depicted in Fig. 3 and consists of a client and a server subsystem. The client-tier middleware API is fully implemented in JavaScript and runs completely in the browser, without add-ins or plugins. The server is a light-weight process, which listens to incoming web requests. The server is only responsible for data synchronization, it does not run application logic. Both client and server have a key-value store to persist data, and they communicate using only web-based HTTP traffic and WebSockets [30]. All communication messages are sent and received inside the client and server subsystems using asynchronous workers. The tags in the ORMap are UUIDs [31] and we use the MD5 [32] algorithm for hashing. We first elaborate on the client-tier subsystem with the public middleware API for applications, and then describe the client-server synchronization protocol.

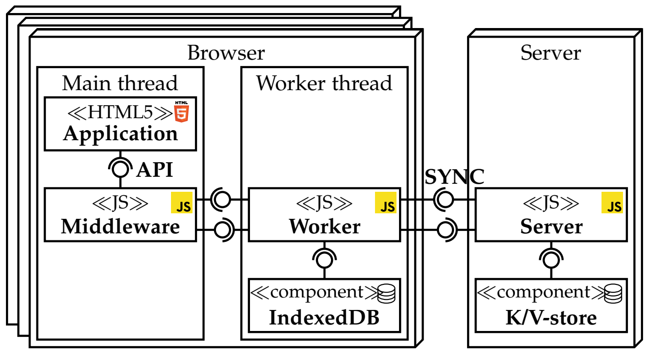

Fig. 3. Overall architecture of the OWebSync middleware

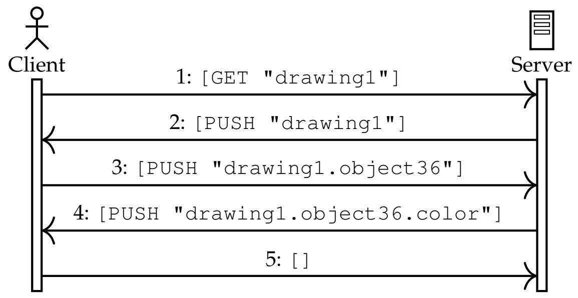
Client
Server
1: [GET "drawing1"]
2: [PUSH "drawing1"]
3: [PUSH "drawing1.object36"]
4: [PUSH "drawing1.object36.color"]
5: []

Fig. 4. Synchronization protocol when another client made an update to the color. A GET message only sends the path and hash value, a PUSH message also sends the respective CRDT. E.g. for message 3, the first CRDT in Figure 2 is sent.

4.2 Client-tier middleware and API

The public programming API of the middleware is located at the client-tier, and web applications are developed as client-side JavaScript applications that use this 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)`: set or update a value.
- `REMOVE(path)`: remove a value or branch.

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("drawing1")
d1.object36.color = "#f00"
OWebSync.set("drawing1", d1)
```

The object at `"drawing1"` is fetched from disc and is represented as a JavaScript object in memory. If there would be other drawings (e.g. drawing2), these will not be loaded. The access to `"d1.object36.color"` is just a plain JavaScript object access and does not involve OWebSync. For performance reasons, it is best to always scope to the smallest possible object from the database, in this example that would be:

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

4.3 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, tag)`: removes the CRDT at a given path if the unique identifier of the value is matching the given `tag`. As such, a newer value with a different `tag` will not be removed.

```
1: $KV$                                              ▷ Key-value store
2: sync: SYNC ($msgs$)
3:     $resp \leftarrow$ []
4:     for all $msg \in msgs$ do
5:         if $msg \equiv$ GET($path, hash$) then
6:             $c \leftarrow KV$.GET($path$)
7:             if $c.hash \neq hash$ then
8:                 $resp$.APPEND(PUSH($path, c$))
9:         else if $msg \equiv$ PUSH($path, crdt$) then
10:            $c \leftarrow KV$.GET("")
11:            $paths \leftarrow c$.MERGE($path$.SPLIT("."), $crdt$)
12:                                               ▷ Procedure from Alg. 2
13:            for all $p \in paths$ do
14:                if $KV$.HAS($p$) then
15:                    $c \leftarrow KV$.GET($p$)
16:                    $resp$.APPEND(PUSH($p, c$))
17:                else
18:                    $resp$.APPEND(GET($p, \perp$))
19:        else if $msg \equiv$ REMOVE($path, tag$) then
20:            $c \leftarrow KV$.GET("")
21:            $c$.REMOVE_WITH_TAG($path$.SPLIT("."), $tag$)
22:    return $resp$                              ▷ Procedure from Alg. 1
```

Alg. 3. Specification of the synchronization protocol, using the ORMap specified in Section 3. Some details are abstracted for conciseness.

The protocol, depicted in Alg. 3, is initiated by a client, which will traverse the Merkle-tree of the CRDTs. The synchronization starts with the CRDT in the root of 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. An empty message is send to signal this to the other side. All data is consistent at that time. If the hash does not match, the server returns a `PUSH` message with the CRDT that is located at the path requested by the client. This does not include the child CRDTs, only the metadata (key, tag, and hash) of the immediate children. The client must merge the new CRDT with the CRDT at its requested path. The specification is listed in Alg. 2. The `MERGE` operation returns a set with all changed paths. Those paths are the paths of the conflicting CRDTs that still need to be synchronized. The client will then `PUSH` the CRDTs belonging to those paths to the server. The server then needs to merge those CRDTs. If a child does not yet exists, 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 length of the synchronization protocol is therefore limited to the maximum depth of the Merkle-tree.

The third type of message, REMOVE, is not strictly necessary, but can improve the bandwidth requirements. If during this synchronization process between a client and the server, a child is 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 tag 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 tag.

Fig. 4 shows an example of 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 four.

4.4 Performance optimization tactics

The main optimization tactic to achieve prompt synchronization for interactive groupware is the reduction of network traffic by the Merkle-trees. However, there are additional tactics needed to further improve synchronization time. To reduce the overhead of the synchronization protocol between the many clients and the server, two optimization tactics are applied by both the client and the server.

4.4.1 Virtual Merkle-tree levels

When the number of child values in an ORMap increases, all the metadata for those children (key, tag, 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 fewer 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 did not 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 and reduces the computation time.

4.4.2 Message batching

In the basic protocol, 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 synchronized. 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 are only 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 to end the synchronization. As a result, fewer messages are sent between a client and server, and only one synchronization round per client is occurring at the same time, resulting in no duplicated messages and work on sub-trees.

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 case study 5 times on 5 representative JavaScript technologies for web-based data synchronization: our OWebSync platform, which uses state-based CRDTs with Merkle-trees, Yjs [9] and Automerge [11] which use operation-based CRDTs, and ShareDB [33] which makes use of OT. We used Legion [17] for testing delta-state CRDTs. Both Yjs (2698 GitHub stars) and ShareDB (3768 GitHub stars) are widely used open source technologies available on GitHub. Automerge is the implementation of the JSON data type of Kleppmann and Beresford [34]. Legion is the implementation of Δ-CRDTs of van der Linde [16], [17]. We did not evaluate Google Docs, which uses OT, as 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 [35] 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 [36] public cloud. A VM has 4 vCPU cores and 8 GB of RAM (Standard A4 v2) and holds up to 3 client containers. A client container contains a browser that loads the client-side middleware from the server. The middleware server is deployed on a separate VM (Standard F4s v2). The monitoring server that captures all performance data is deployed on a separate VM. VMs in Azure have their clocks synchronized with the host machine, which is synchronized with the internal Microsoft time servers. The Linux `tc` tool [37] 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 [38].

Our evaluation contains three benchmarks. The first 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. The third and last benchmark is used to measure the total size of the data set over a longer time period.

5.1 Performance of continuous online updates

The first benchmark represents the continuous online scenario where clients are making updates for 10 minutes and stay online the whole time. In total, we executed 30 tests for this benchmark: 6 tests to be executed by each of the 5 technologies. These 6 tests vary in the 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 Fig. 1 in Section 3 and has 7 attributes. The total depth of the three is four (root – `drawing1` – `object36` – `top`). The root has one child, while `drawing1` has either 100 or 100 children. And these children have itself each 7 children. This is the shape of the tree defined by the application, however, because `drawing1` has many children, OWebSync will transparently add an extra layer in the tree to reduce the network usage. This increases the total depth of the tree to 5.

Each client edits one object, which leads to two random writes, the x and y position, on a shared object every second. We use at most 24 clients, which are editing the same document concurrently. In comparison, Google Docs, which is the most popular collaborative editing system today, supports a maximum of 100 concurrent users according to Google itself [2]. But in practice, latency starts to increase significantly when the number of users exceeds 10 [39]. Our performance results show the same problem for ShareDB, which uses the same technique. In our performance evaluation, one iteration of a test takes 10 minutes. Before each test, the database is populated and the initial synchronization is performed. The first minute is used for warm-up. To ensure the stability of the test results, all tests are repeated 10 times.

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 updates are illustrated in Fig. 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 do not 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 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 do not need a transformation, the server does not become a bottleneck. These tests show that state-based CRDTs, which are currently only used for background synchronization between servers, can also be used in interactive groupware. This improvement is due to the use of Merkle-trees embedded in the data structure, the use of virtual Merkle-tree levels for large objects, and message batching.

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 (Fig. 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 in 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, virtual, 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 [40]. 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 Fig. 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.

CPU usage. We show the CPU usage for the experiment with 24 clients and 1000 objects in Table 1. The average client-side CPU usage for OWebSync is 9%, which is similar to Legion and ShareDB, and about half of Yjs and Automerge. The server-side CPU usage for OWebSync is higher, 33%, as it essentially needs to run the same synchronization protocol for every client. It still performs better than ShareDB, which uses OT and needs to transform all operations on the server before they are sent to the clients. The operation-based approach of Yjs and Automerge, and the delta-state-based approach of Legion, are more efficient, as the server can keep track of which client needs which updates. Automerge performs worse than expected, but we assume that this is because it stores the whole history and uses long text-based UUIDs as client identifiers.

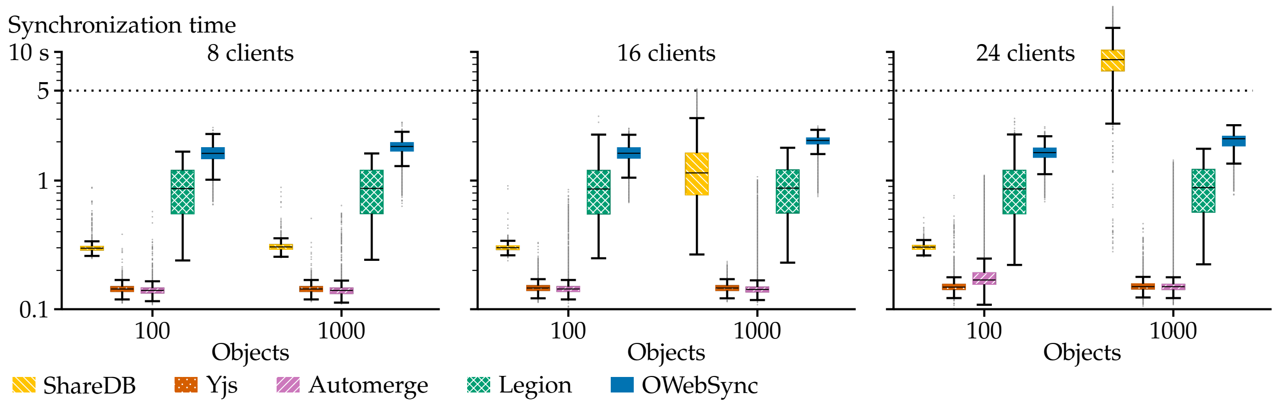

Fig. 5. Aggregated boxplots containing the times to synchronize an update to all other clients in the online scenario. Each boxplot contains all 10 iterations for each of the 30 tests in the fully online situation. To compare technologies that have results of the same order of magnitude, as well as results in different orders of magnitude, we opted for a logarithmic Y-axis.

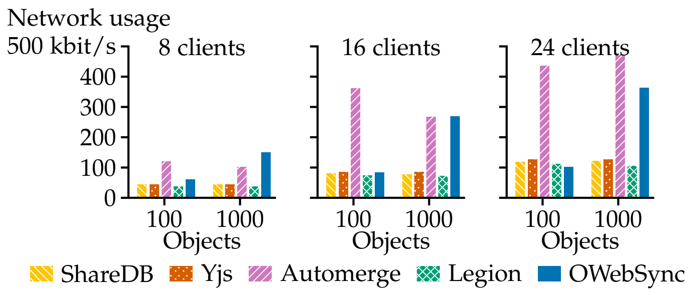

Fig. 6. Network usage per client for each test in the online scenario.

Interpretation and discussion. For interactive web applications and groupware, usability guidelines [1], [41] state that remote response times should 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* [15]). The 99th percentile for the synchronization time of 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. This is in line with the observations of Dang et al. [39] for Google Docs, which also uses OT. The other technologies stay well below 5 seconds in the online scenario and can be called interactive.

5.2 Performance in disconnected scenarios

We now present the performance analysis 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 for 1 minute (shown at 2 minutes in the graphs as the first minute is used as a warm-up). This 1-minute network disruption will lead to many conflicting operations, which will automatically be resolved by the middleware. During the disruption, there will be 1440 offline updates in the largest experiment with 24 clients. A longer offline period will not change much for OWebSync since only the state is kept and the same client moving the same object twice will result in the same amount of state to be sent. Operation-based approaches will take longer when the time increases since they have to send all operations anyway.

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, shown in Fig. 7, show that OWebSync can synchronize all missed updates faster than ShareDB, Yjs, Automerge, and Legion. Note that these boxplots are different from the previous figure. At the median of these boxplots, only 50% of the missed updates are synchronized. Only at the upper whisker, most of the missed updates are fully synchronized. Whiskers have a maximum of 1.5 times the interquartile range.

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 must 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 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.

Timeline analysis of the tests. The timelines in Fig. 8 show the resynchronization times on the y-axis, without the offline time during the network disruption, for each update

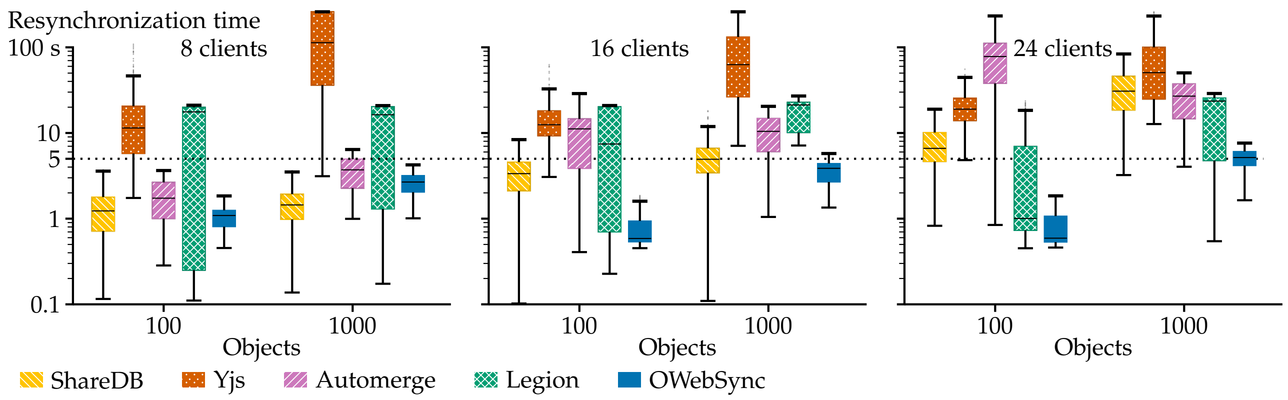

Fig. 7. Boxplots of the time it takes for an update during the failure scenario to be received by all clients. The time before a client notices the network connection is reestablished is not taken into account. Note that the median here means that only 50% of all missed updates are synchronized to all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

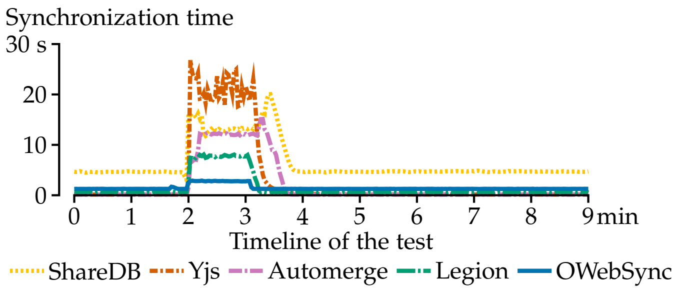

Fig. 8. Mean time to synchronize updates in case of a network disruption between minute 2 and 3 for the test with 24 clients, 1000 objects.

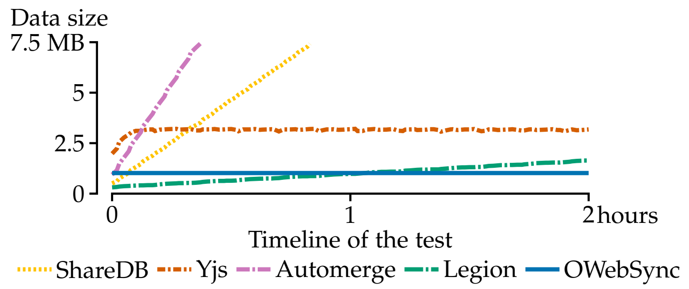

Fig. 9. Evolution of the total data size on the server.

done at a given moment during the test timeline. This means that for an update done 20 seconds before the end of the disruption, and which got synchronized to all other clients 22 seconds later, the resynchronization time is 2 seconds.

In the test with 24 clients and 1000 objects, 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 will not be able to regain interactive performance. When coming back online, those technologies cannot achieve prompt and interactive synchronization immediately.

5.3 Total size of the data model

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.

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. Fig. 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. Other techniques grow with the number of clients and operations. In the dynamic setting of the web, keeping track of all clients with version vectors and 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 [9]. 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 evaluated a worst-case scenario for OWebSync, with clean data sets for the other frameworks.

TABLE 1
Synchronization time and CPU usage for 24 clients and 1000 objects.

	Synchronization time [s]				CPU [%]	
	online		offline		client	server
	50%	99%	50%	99%		
ShareDB	4.45	7.69	12.67	25.10	10	101
Yjs	0.14	0.17	20.21	109.15	20	10
Automerge	0.14	0.20	11.59	18.90	22	54
Legion	0.64	1.03	7.61	8.56	9	5
OWebSync	1.34	1.49	2.87	3.53	9	33

5.4 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 8 seconds to synchronize missed updates, which cannot be called interactive anymore. Moreover, the size of the data set will increase with both 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 groupware. In a setting with frequent offline situations, e.g. for mobile employees, OWebSync is the most appropriate technology and outperforms all other frameworks. Over time, OWebSync can continue to deliver the same interactive performance, as no client identifiers or version vectors are stored. Table 1 summarizes the results in seconds of the large scale test with 24 clients and 1000 objects for the average user (50th percentile) and most of the users (99th percentile) for both settings.

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 [42] 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 [43] allow to replicate only part of a CRDT. This helps with bandwidth and memory consumption, as well as security and privacy [44]. OWebSync allows replicating any arbitrary subtree of the whole CRDT tree. Hybrid approaches combining operation-based and state-based CRDTs are also possible as demonstrated by Bendy [45]. 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. A garbage collection technique can be used to reduce the memory usage of operation-based CRDTs by defining a join-protocol for dynamic environments [46]. But this only treats transient network disruptions where clients will come back online eventually, which is not necessarily the case for web clients. Strong Eventually Consistent Replicated Objects (SECROs) [47] are similar to operation-based CRDTs, but do not impose restrictions on commutativity of operations. However, by doing so, they need a global total order and cannot tolerate network disruptions.

Distributed data systems and NoSQL systems. Based on the Dynamo paper [15], 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 conflict resolution. Cassandra [48], [49] 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 [50] and MongoDB [51] 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 version conflicts manually. It also does not support fine-grained conflict detection within two documents.

Several commercial database systems allow to use CRDTs as the underlying data model: e.g. Riak [12], Akka [52] and Redis [53]. Besides those commercial products, several research projects have emerged. Merkle Search Trees (MSF) [13] implement a key-value store like Dynamo using a state-based CRDT and a Merkle-tree. It builds the Merkle-tree on top of the flat data structure, while OWebSync will make use of the tree-like structure of the data to build the Merkle-tree. MSF is targeted to asynchronous background synchronization between backend servers, and not for interactive groupware with replication to the clients. Antidote [54] 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 and supports partial replication but assumes continuous online connections as the default operational situation for clients. SMAC [55] uses an operation-based CRDT storage system for state management tasks for distributed container deployments. DottedDB [56] uses node-wide dot-based clocks to find changes that need to be replicated, without the need for explicit tombstones. It does not 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 [57] 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 [5], [6], [58] 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. Hence, 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.

Besides the frameworks for native clients, there are several JavaScript frameworks made for synchronization between distributed web clients. Legion [17], [18] 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 [10], [11] is a JavaScript library for data synchronization adopting the operation-based JSON data type of Kleppman [34]. It uses vector clocks which grow in size with the number of clients. PouchDB [59] 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 [33] 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 [7], [8], [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 [60] 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. In contrast with OWebSync, 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 a shared data set. Our OWebSync middleware implements a generic data model, based on JSON, that combines state-based CRDTs with Merkle-trees. This 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. Existing approaches use client identifiers and version numbers, or even 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 resynchronization. Current state-based approaches using delta-state-based CRDTs are faster to recover than the operation-based ones, but cannot achieve prompt resynchronization 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, making it suitable for interactive web applications and groupware.

References

[1] J. Nielsen, *Usability Engineering*. Nielsen Norman Group, 1993. [Online]. Available: https://www.nngroup.com/books/usability-engineering/

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

[3] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *2009 29th IEEE International Conference on Distributed Computing Systems*, June 2009, pp. 395–403.

[4] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," Inria – Centre Paris-Rocquencourt ; INRIA, Research Report RR-7506, Jan. 2011. [Online]. Available: https://hal.inria.fr/inria-00555588

[5] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça, "Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine," INRIA, Research Report RR-8347, Oct. 2013. [Online]. Available: https://hal.inria.fr/hal-00870225

[6] N. Preguiça, M. Zawirski, A. Bieniusa, S. Duarte, V. Balegas, C. Baquero, and M. Shapiro, "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, 2014, pp. 30–33.

[7] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, "Yjs: A framework for near real-time p2p shared editing on arbitrary data types," in *Engineering the Web in the Big Data Era*. Cham: Springer International Publishing, 2015, pp. 675–678.

[8] ——, "Near real-time peer-to-peer shared editing on extensible data types," in *Proceedings of the 19th International Conference on Supporting Group Work*, ser. GROUP '16. NY, USA: ACM, 2016, pp. 39–49.

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

[10] M. Kleppman and A. R. Beresford. (2018) Automerge: Real-time data sync between edge devices. [Online]. Available: http://martin.kleppmann.com/papers/automerge-mobiuk18.pdf

[11] "Automerge," https://github.com/automerge/automerge, 2017.

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

[13] A. Auvolat and F. Taïani, "Merkle Search Trees: Efficient State-Based CRDTs in Open Networks," in *SRDS 2019 - 38th IEEE International Symposium on Reliable Distributed Systems*. Lyon, France: IEEE, Oct. 2019. [Online]. Available: https://hal.inria.fr/hal-02303490

[14] R. Merkle, "Method of providing digital signatures," 1982, uS patent 4309569. The Board Of Trustees Of The Leland Stanford Junior University.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS operating systems review*, vol. 41(6). NY, USA: ACM, 2007, pp. 205–220.

[16] A. van der Linde, J. a. Leitão, and N. Preguiça, "Δ-crdts: Making δ-crdts delta-based," in *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '16. NY, USA: ACM, 2016, pp. 12:1–12:4.

[17] A. van der Linde, P. Fouto, J. a. Leitão, N. Preguiça, S. Castiñeira, and A. Bieniusa, "Legion: Enriching internet services with peer-to-peer interactions," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. International World Wide Web Conferences Steering Committee, 2017, pp. 283–292.

[18] "Legion," https://github.com/albertlinde/Legion, 2016.
[19] T. Bray, "The javascript object notation (json) data interchange format," Internet Requests for Comments, IETF, RFC 7158, 2014. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7158.txt
[20] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," *SIGMOD Rec.*, vol. 18, no. 2, pp. 399–407, Jun. 1989.
[21] S. Kumawat and A. Khunteta, "A survey on operational transformation algorithms: Challenges, issues and achievements," *International Journal of Computer Applications*, vol. 3, pp. 30–38, Jul. 2010.
[22] M. Shapiro, N. Perguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems*, ser. Lecture Notes in Computer Science, X. Défago, F. Petit, and V. Villain, Eds., vol. 6976. Berlin, Heidelberg: Springer Berlin Heidelberg, Oct. 2011, pp. 386–400.
[23] P. S. Almeida, A. Shoker, and C. Baquero, "Efficient state-based crdts by delta-mutation," in *Networked Systems*. Cham: Springer International Publishing, 2015, pp. 62–76.
[24] ——, "Delta state replicated data types," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 162 – 173, 2018.
[25] "Delta crdts," https://github.com/peer-base/js-delta-crdts, 2018.
[26] V. Enes, C. Baquero, P. S. Almeida, and A. Shoker, "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*, ser. PMLDC '16. NY, USA: ACM, 2016, pp. 6:1–6:3.
[27] V. Enes, P. S. Almeida, C. Baquero, and J. Leitão, "Efficient synchronization of state-based crdts," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, April 2019, pp. 148–159.
[28] M. Shapiro, *Replicated Data Types*. NY: Springer, 2017, pp. 1–5.
[29] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 354–368, 2011.
[30] I. Hickson, "The websocket api, w3c candidate recommendation," Tech. Rep., 2012. [Online]. Available: https://www.w3.org/TR/2012/CR-websockets-20120920/
[31] P. Leach, M. Mealling, and R. Salz, "A universally unique identifier (uuid) urn namespace," Internet Requests for Comments, RFC 4122, 2005. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4122.txt
[32] R. Rivest, "The md5 message-digest algorithm," Internet Requests for Comments, RFC 1321, 1992. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1321.txt
[33] "Sharedb," https://github.com/share/sharedb, 2013.
[34] M. Kleppmann and A. R. Beresford, "A conflict-free replicated json datatype," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2733–2746, 2017.
[35] "Fabric.js," https://github.com/fabricjs/fabric.js, 2011.
[36] "Azure," https://azure.microsoft.com, 2019.
[37] W. Almesberger, "Linux network traffic control – implementation overview," EPFL, Tech. Rep., 1999. [Online]. Available: https://www.almesberger.net/cv/papers/tcio8.pdf
[38] "opensignal.com," https://www.opensignal.com/reports/2019/01/usa/mobile-network-experience, 2019.
[39] Q.-V. Dang and C.-L. Ignat, "Performance of real-time collaborative editors at large scale: User perspective," in *Internet of People Workshop, 2016 IFIP Networking Conference*, ser. Proceedings of 2016 IFIP Networking Conference, Networking 2016 and Workshops. Vienna, Austria: IFIP, May 2016, pp. 548–553.
[40] "Speedtest.net," http://www.speedtest.net/reports/united-states/2018/Mobile/, 2018.
[41] J. Nielsen. (2010) Website response times. [Online]. Available: https://www.nngroup.com/articles/website-response-times/
[42] "Git," https://git-scm.com/, 2005.
[43] I. Briquemont, M. Bravo, Z. Li, and P. Van Roy, "Conflict-free partially replicated data types," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2015, pp. 282–289.
[44] S. A. Kollmann, M. Kleppmann, and A. R. Beresford, "Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 3, pp. 210 – 232, 2019.
[45] C. Bartolomeu, M. Bravo, and L. Rodrigues, "Dynamic adaptation of geo-replicated crdts," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC '16. NY, USA: ACM, 2016, pp. 514–521.
[46] J. Bauwens and E. Gonzalez Boix, "Memory efficient crdts in dynamic environments," in *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL 2019. NY, USA: ACM, 2019, pp. 48–57.
[47] K. De Porre, F. Myter, C. De Troyer, C. Scholliers, W. De Meuter, and E. Gonzalez Boix, "Putting order in strong eventual consistency," in *Distributed Applications and Interoperable Systems*. Cham: Springer International Publishing, 2019, pp. 36–56.
[48] "Apache cassandra," https://cassandra.apache.org, 2009.
[49] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.
[50] "Couchdb," https://couchdb.apache.org, 2005.
[51] "Mongodb," https://www.mongodb.com/, 2009.
[52] "Akka," https://doc.akka.io/docs/akka/current/distributed-data.html, 2018.
[53] C. Biyikoglu, "Under the hood: Redis crdts (conflict-free replicated data types)," Redis Labs, White paper, 2017. [Online]. Available: https://redislabs.com/docs/under-the-hood/
[54] "Antidote," http://syncfree.github.io/antidote, 2014.
[55] J. Eberhardt, D. Ernst, and D. Bermbach, "Smac: State management for geo-distributed containers," Technische Universitaet Berlin, Tech. Rep., 2016.
[56] R. J. T. Gonçalves, P. S. Almeida, C. Baquero, and V. Fonte, "Dotteddb: Anti-entropy without merkle trees, deletes without tombstones," in *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2017, pp. 194–203.
[57] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat, "Cimbiosys: A platform for content-based partial replication," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, 2009, pp. 261–276.
[58] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro, "Write fast, read in the past: Causal consistency for client-side applications," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15. ACM, 2015, pp. 75–87.
[59] "Pouchdb," https://pouchdb.com, 2013.
[60] "Swarm.js," https://github.com/gritzko/swarm, 2013.

Kristof Jannes is a Ph.D. candidate in the Department of Computer Science at KU Leuven in Belgium, and a member of the research group imec-DistriNet. His research activities are under the supervision of Prof. Dr. Wouter Joosen and Dr. Bert Lagaisse. He received his Master's degree in computer science from the KU Leuven in 2018. His main research interests are in the area of data synchronization, consensus and decentralization.

Bert Lagaisse is a senior industrial research manager at the imec-DistriNet research group in which he manages a portfolio of applied research projects on cloud technologies, distributed data management and security middleware in close collaboration with industrial partners. He has a strong interest in distributed systems, in enterprise middleware, cloud platforms and security services. He obtained his MSc in computer science at KU Leuven in 2003 and finished his Ph.D. in the same domain in 2009.

Wouter Joosen is full professor at the Department of Computer Science of the KU Leuven in Belgium, where he teaches courses on software architecture and component-based software engineering, distributed systems and the engineering of secure service platforms. His research interests are in aspect-oriented software development, focusing on software architecture and middleware, and in security aspects of software, including security in component frameworks and security architectures.