KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Client-centric Replication for the Decentralized Web

Kristof Jannes

Supervisors:
Prof. dr. ir. Wouter Joosen
Prof. dr. Bert Lagaisse

Dissertation presented in partial fulfillment
of the requirements for the degree of
Doctor of Engineering Science (PhD):
Computer Science

August 2023

Client-centric Replication for the Decentralized Web

ir. Kristof Jannes

Examination committee:
Prof. dr. ir. Hendrik Van Brussel, chair
Prof. dr. ir. Wouter Joosen, supervisor
Prof. dr. Bert Lagaisse, supervisor
Dr. ir. Lieven Desmet
Prof. dr. Bettina Berendt
Prof. dr. ir. Bart Preneel
Prof. dr. Tom Van Cutsem
Prof. dr. Elisa Gonzalez Boix
(Vrije Universiteit Brussel)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

August 2023

© 2023 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Kristof Jannes, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Preface

I would like to thank my promotor Wouter Joosen for providing me with the opportunity to pursue a PhD within DistriNet, and for giving me the freedom to work on the topics I want. I would like to thank my co-promotor Bert Lagaisse for his full-time support, advice, and coaching. Thanks for all your ideas, comments, and motivation to deal with yet another reject. This manuscript would not have been here without you.

I would like to thank the members of the examination committee: Lieven Desmet, Bettina Berendt, Elisa Gonzalez Boix, Tom Van Cutsem, and Bart Preneel. Thank you to read my manuscript and for your suggestions to improve it. I would like to thank Hendrik Van Brussel for chairing the examination committee.

Furthermore, I would like to thank all my fellow researchers and colleagues with whom I collaborated or had many interesting discussions over the years: Emad Heydari Beni, Martijn Sauwens, Pieter-Jan Vrielynck, Vincent Reniers, Dimitri Van Landuyt, Davy Preuveniers, Eddy Truyen, and Ansar Rafique. Thanks to all colleagues that joined for the alma lunch, went on travels, drank coffee with me, or simply created a nice working environment: Federico Quin, Laurens Sion, Koen Yskout, Victor Le Pochat, Stef Verreydt, Tim Van hamme, Matthijs Kaminski, Alexander van den Berghe, Amin Timany, Koen Jacobs, Jafar Hamin, Tobias Reinhard, Toon Dehaene, Niels Mommen, Michiel Provoost, and many more. Thank you to all administrative and technical staff at the DistriNet business office (Katrien Janssens, Annick Vandijck, An Makowski, and Annelies Wouters), the departmental secretariat, and the systems group.

Finally, I would also like to thank my parents, family, and friends for always being there in my life, for their support, and for the necessary distractions.

Thank you!

– Kristof Jannes

Abstract

Distributed systems are currently evolving from a centralized client-server architecture to decentralized, web-based architectures. Decentralized systems use replication techniques that decentralize control, but have several challenges such as resilience, interactivity, and storage overhead. In this dissertation, we address client-centric replication for the web in three distinct environments characterized by varying trust and consistency requirements: strong eventual consistency in a trusted setting, strong consistency in a Byzantine setting, and strong eventual consistency in a Byzantine setting. Each of these scenarios presents unique challenges and requirements that need to be addressed. For the first environment, strong eventual consistency in a trusted setting, we present a Conflict-free Replicated Data Type protocol that is fully state-based, yet supports fine-grained delta-merging without keeping track of individual clients. Secondly, for strong consistency in an untrusted environment, we propose a Byzantine Fault Tolerant consensus protocol with a novel way to synchronize consensus votes between the replicas, making the protocol fully leaderless. At last, we present a Conflict-free Replicated Data Type protocol that supports a Byzantine environment with strong eventual consistency, without the need to keep track of individual transactions or clients. To evaluate the proposed protocols' effectiveness, we implemented them in three separate browser-based middlewares and assessed their performance in realistic settings, including failure scenarios. This research contributes to the advancement of replication techniques in decentralized web architectures by providing robust solutions for diverse trust and consistency requirements, ultimately enhancing the resilience, interactivity, and storage efficiency of such systems.

Beknopte samenvatting

Gedistribueerde systemen evolueren momenteel van een gecentraliseerde client-server architectuur naar gedecentraliseerde, web-gebaseerde architecturen. Deze gedecentraliseerde systemen gebruiken replicatietechnieken die controle decentraliseren, maar hebben verschillende uitdagingen zoals robuustheid, interactiviteit en opslag overhead. In dit proefschrift behandelen we client-centrische replicatie voor het web in drie verschillende omgevingen die worden gekenmerkt door variërende vertrouwens- en consistentievereisten: sterke uiteindelijke consistentie in een vertrouwde omgeving, sterke consistentie in een Byzantijnse omgeving, en sterke uiteindelijke consistentie in een Byzantijnse omgeving. Elk van deze scenario's presenteert unieke uitdagingen en vereisten die moeten worden aangepakt. Voor de eerste omgeving, sterke uiteindelijke consistentie in een vertrouwde omgeving, presenteren we een Conflictvrij Gerepliceerd Datatype protocol dat volledig toestandsgebaseerd is, maar toch fijnmazige delta-samenvoeging ondersteunt zonder individuele clients bij te houden. Ten tweede, voor sterke consistentie in een onbetrouwbare omgeving, stellen we een Byzantijns Fouttolerant consensusprotocol voor met een nieuwe manier om consensusstemmen tussen de replica's te synchroniseren, waardoor het protocol volledig leiderloos wordt. Tot slot presenteren we een Conflictvrij Gerepliceerd Datatype protocol dat een Byzantijnse omgeving met sterke uiteindelijke consistentie ondersteunt, zonder de noodzaak om individuele transacties of clients bij te houden. Om de effectiviteit van de voorgestelde protocollen te evalueren, hebben we ze geïmplementeerd in drie afzonderlijke middlewares voor de browser en hebben we hun prestaties beoordeeld in realistische omstandigheden, inclusief faalscenario's. Dit onderzoek draagt bij aan de vooruitgang van replicatietechnieken in gedecentraliseerde webarchitecturen door robuuste oplossingen te bieden voor uiteenlopende vertrouwens- en consistentievereisten, waardoor uiteindelijk de robuustheid, interactiviteit en opslagefficiëntie van dergelijke systemen wordt verbeterd.

Contents at a Glance

1 Introduction *1*

2 Eventual Consistency in a Trusted Setting *17*

3 Strong Consistency in a Byzantine Setting *53*

4 Eventual Consistency in a Byzantine Setting *87*

5 Conclusion *103*

Contents

1 Introduction *1*
1.1 Client-centric replication *3*
1.2 Research goals *8*
1.3 Approach *10*
1.4 Contributions *14*
1.5 Overview *14*

2 Eventual Consistency in a Trusted Setting *17*
2.1 Introduction *19*
2.2 Motivation and Background *21*
2.2.1 Case studies *21*
2.2.2 Background *22*
2.2.3 Principles *24*
2.3 The OWebSync Data Model *25*
2.3.1 Approach *25*
2.3.2 Observed-Removed Map *25*
2.3.3 Considerations and discussion *31*
2.4 Architecture and Synchronization *32*
2.4.1 Overall architecture *32*
2.4.2 Client-tier middleware and API *33*
2.4.3 Synchronization protocol *34*
2.4.4 Performance optimization tactics *36*
2.5 Performance evaluation *37*
2.5.1 Performance of continuous online updates *38*
2.5.2 Performance in disconnected scenarios *42*
2.5.3 Total size of the data model *45*
2.5.4 Summary *46*
2.6 Related work *47*
2.7 Conclusion *50*

3 Strong Consistency in a Byzantine Setting *53*
3.1 Introduction *55*
3.2 Motivation *57*
3.3 BeauForT protocol *58*
3.3.1 System model *58*
3.3.2 Protocol specification *59*
3.3.3 Safety and liveness proofs *68*
3.4 Architecture and implementation *71*
3.4.1 Overall architecture *73*
3.4.2 Aggregate signatures using BLS *74*
3.5 Evaluation *76*
3.5.1 Validation in the loyalty points use case *76*
3.5.2 Test setup *77*
3.5.3 Optimistic scenario *77*
3.5.4 Realistic scenario *78*
3.5.5 Byzantine scenario *79*
3.5.6 Discussion and conclusions *80*
3.6 Related work *80*
3.7 Conclusion *83*

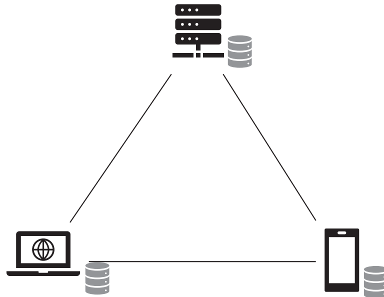
4 Eventual Consistency in a Byzantine Setting *87*
4.1 Introduction *89*
4.2 System model *91*
4.3 Secure CRDTs *92*
4.3.1 Encrypted CRDT *92*
4.3.2 Modified Merkle Patricia Trie *95*
4.3.3 Key derivation and rotation *95*
4.3.4 Global time *96*
4.3.5 Discussion *97*
4.4 Evaluation *98*
4.5 Related work *99*
4.6 Conclusion and future work *100*

5 Conclusion *103*
5.1 Summary of contributions *104*
5.2 Limitations and future directions *105*

Bibliography *111*

1

Introduction



The computing landscape has changed dramatically over the last decades. It started with a centralized model where all computations were done on a central mainframe controlled by thin clients. All data is stored on the central mainframe, and all connected clients have a user name and the central operating system enforces access control based on user names. The clients themselves are mostly just a screen, with minimal storage and computing capacity.

Later, these clients became more powerful and computation shifted to these client devices. This initiated the era of personal computing, in which each user had a personal computer. Most applications are running on that computing device, and solely operate on local data. There is no active replication of the data. If users want to exchange some data, they can use a physical device such as a floppy disk or USB drive. With the widespread availability of the internet, exchanging data via centralized servers using internet-based protocols such as FTP [PR85] and HTTP [BFN96] became possible. In the early 2000s, it became possible to exchange data via decentralized networks such as Napster, Gnutella, and BitTorrent [Coh03]. This allowed client devices to exchange data in a peer-to-peer fashion, i.e., directly between the client devices instead of via a central server [Ora01; Lua+05]. However, the data remained static in nature, as it was not possible to edit a document on one device and have the changes automatically propagate to other devices.

CVS and SVN emerged as centralized version control systems, facilitating collaboration and change tracking. Yet, they still relied on a central server, which could become a single point of failure. Eventually, Git was developed as a distributed version control system, enabling better collaboration and resilience by allowing users to maintain a complete copy of the project on their local machines. However, it has some shortcomings compared to real-time collaboration platforms, particularly in terms of latency and handling simultaneous edits.

With the rise of cloud computing [Vaq+09; Buy+09] and Software-as-a-Service offerings, the computations shifted back to central servers. Cloud computing is beneficial for users in terms of availability, durability, security, and collaboration. By having a centralized service in the cloud, which manages all data, users can seamlessly collaborate on the same data. Clients need little storage and computing capacity, as the data is stored on servers that can run complex queries and computations. However, this dependence on a centralized service has some shortcomings. If the internet connection is slow, client-side performance will also deteriorate. Even worse, when no internet is available, the application and the user's data are not available at all. Furthermore, a lot of trust is given to the service provider. If they do a bad job securing the data, the user's data can be hacked and stolen. Also, malicious employees can access the data and the company can even decide to sell the data to third parties. Finally, the user has no control over the data, as it is stored on the servers of the service provider which

can decide to delete or modify the data at any time. Lots of data, and therefore power, is given to a few large tech companies and governments, while end-users are losing sovereignty over their own data [Ber17].

To potentially address these issues, data, and computation could be moved back to client devices. However, near-real-time collaboration on the same data set should still be supported. Therefore, the data should be replicated across multiple client devices. This leads to the paradigm of client-centric replication [Dem+94]. Data is replicated among numerous client devices and servers without designating a single device as the authoritative copy. One of the major problems of replication is keeping replicas consistent. When one copy is updated, the other copies should be updated as well.

This dissertation focuses on addressing the challenges of client-centric replication on the web, especially the problems of resilience and fault-tolerance, interactive performance, and storage overhead.

This chapter first discusses the current state-of-the-art of client-centric replication, together with the major limitations and shortcomings. Secondly, the goal of this dissertation is presented. In the later sections of this chapter, we describe our approach and present the three major contributions of this work. This chapter concludes with an overview of the structure of this dissertation.

1.1 Client-centric replication

An important aspect in the area of distributed systems is data replication. Client-centric replication is an approach for managing data more efficiently, improving performance, and enhancing reliability. However, the challenges posed by mobile networks, such as latency, robustness, and bandwidth constraints, can make implementing client-centric replication more difficult. These factors can hinder the overall efficiency of client-centric systems, requiring the exploration of different replication strategies that can effectively address the specific requirements of mobile devices and their users.

Today, we are witnessing two different trends in the current approach to client-centric replication. The first trend is the field of local-first software [Kle+19], a paradigm in which updates are made directly on the local device and where the data will be replicated later to other devices. The second trend is distributed ledger technology, also known as blockchain [Nak08], in which all replicas keep track of a ledger and make sure that all copies are consistent.

Both forms of client-centric replication move data from a centralized location in the cloud to the client devices, allowing peer-to-peer communication. However, they are very different in terms of consistency level, as local-first software is

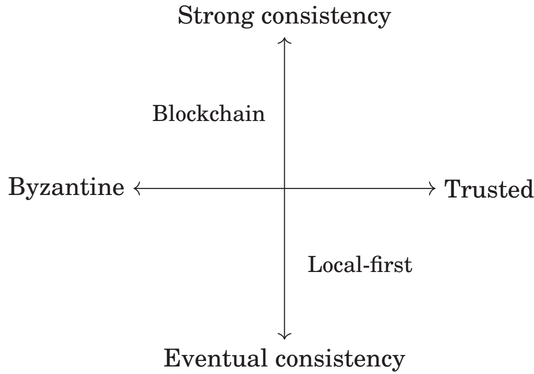


Figure 1.1: The two trends in client-centric replication situated across two dimensions: level of trust and consistency requirements.

typically eventually consistent, while blockchain is strongly consistent. There are also differences in trust models, as local-first software is typically used in a trusted environment of honest replicas, while blockchain is used in a Byzantine environment with potentially malicious replicas. These different dimensions of the level of trust and consistency requirements for client-centric replication are depicted in Figure 1.1.

Trend 1: Local-first software

Local-first software [Kle+19; JLJ19a; HK20; Haa22] is a paradigm in which clients always make updates directly on their local devices. Later on, the updates are synchronized with the cloud or other client devices. Because all updates are done locally, the application is always available, even when there is no internet connection. Generally, performance is better because local users do not experience network latency. Furthermore, the user has full sovereignty over the data, as it is stored locally. However, for developers, these kinds of applications with eventually consistent data are harder to develop, as they need to deal with synchronization and conflict resolution. Local-first software offers improved data ownership by storing data on the local client device and enables real-time collaboration. This is a major difference with file-based synchronization protocols such as Dropbox, where files are also stored locally, but fine-grained real-time collaboration is not possible. Solutions such as Google Docs do allow real-time collaboration, but the data and coordination are done on a central server, and it does not allow offline usage.

Kleppmann [Kle+19] proposed seven ideals for local-first software:

1. The primary copy of the data is stored locally on the client device, this makes local interactions with the application fast and responsive;
2. The data is replicated to the cloud or other client devices, this makes the data available on other devices;
3. No network connection is required to make changes to the data;
4. Collaboration should be seamless, with no need for explicit coordination between users;
5. If both the data and software are stored on client devices, it enables greater longevity since there is no dependence anymore on the existence of the central service provider that can decide to shut down their operations;
6. Security and privacy by design[1], there is no need for a service provider to have access to the actual data;
7. Users have full ownership and control over their data.

In essence, these ideals mean going back to the advantages of the early days of personal computing, where each user had a personal computer with applications that operate on data stored locally. However, the world has changed since then, and users now have multiple devices on which they want to access their data. As well as the fact that users want to collaborate on the same data set in real-time.

The web is a natural environment and eco-system for local-first software [JLJ19a]. The current state of browser technologies allows developers to create rich web applications that can function across different operating systems as well as on personal computers and mobile phones. Yet, the current paradigm of the web is still server-centric. The key data is stored, served, processed, and analyzed on central servers from the service provider.

Tim Berners-Lee, the founder of the web, has observed over the years that users have lost control of their personal data [Ber17]. To regain control, Tim Berners-Lee proposed that the web should evolve into a decentralized network, where data can be stored under the control of the user. Therefore, browsers need to shift from the client-server paradigm to a decentralized peer-to-peer approach. However, a true peer-to-peer approach of end-user devices is not very durable and available. Devices are often not online at the same time, do not have a large amount of storage space, and can fail more easily or more frequently compared to a server inside a data center. The Solid Platform [Man+16] proposes to use a Personal Online Datastore (pod) that can either be self-hosted or hosted with a third-party pod provider. This central pod, under the control of the user, can

[1] Assuming users are responsible to secure their own devices well.

be used to store and collaborate on data from the user. However, as most users will opt to use a cloud service to run their pod, many of the ideals of local-first software will not be reached. Data is far away from the user, and not at all available when offline. Security and privacy will depend on the trust in the pod provider, instead of the service provider for classical cloud applications. This can already be an improvement for smaller applications, but it also creates more interesting targets for attackers.

Trend 2: Blockchain

Blockchain is a distributed ledger technology that is used to store and edit data in a decentralized way. Nakamoto introduced the concept of blockchain in 2008 [Nak08] with the creation of Bitcoin. Bitcoin's primary focus was to create a decentralized digital currency that is not controlled by any central authority. Instead, all replicas agree on the current state of who owns how many tokens. Later, other blockchains were created, such as Ethereum [But+13], which support generic computations on the data via so-called smart contracts. Similar to local-first software, all replicas have a full copy of the data and can propose transactions to modify the current state. Different from local-first software, blockchains do not aim for eventual consistency but instead require strong consistency. Furthermore, in general, other replicas are not trusted, and this strong consistency should hold, even when other replicas are actively trying to break it. Such replicas are called Byzantine replicas. These blockchains such as Bitcoin and Ethereum require large amounts of processing power, have high transaction costs, and result in long latency times before a transaction is executed and confirmed reliably.

Challenges and state-of-the-art

Client-centric replication trends are supported by two underlying technologies: Conflict-free Replicated Data Types (CRDTs) and Byzantine Fault Tolerant (BFT) consensus in blockchain. In this section, we will discuss these technologies and their current limitations in the state-of-the-art.

Conflict-free Replicated Data Types

Conflict-free Replicated Data Types (CRDTs) emerged in 2011 [Sha+11b] and are an essential technology to support local-first client-centric replication. CRDTs are data structures that can be replicated across multiple devices, and that can be merged together to automatically resolve conflicts. These data structures guarantee Strong Eventual Consistency (SEC) without the need for explicit coordination between replicas or rollbacks. An object is eventually consistent if it has *eventual delivery* (an update delivered at some correct replica is eventually

delivered to all correct replicas), *convergence* (correct replicas that have delivered the same updates eventually reach equivalent state), and *termination* (all method executions terminate). For *strong* eventual consistency, a stronger convergence property is required: *strong convergence* (correct replicas that have delivered the same updates have equivalent state) [Sha+11a].

There are different types of CRDTs: operation-based, state-based, and delta-state-based. Operation-based CRDTs [Sha+11a] are the most commonly used in local-first software [KB18; Nic+15]. They work by making sure that every concurrent operation is commutative. This way, once all operations have reached all clients, in whatever order, the data structure will be in the same state on each client. However, to make sure that all these operations reach all clients, a reliable message broadcast channel is required, which also often needs to guarantee causally ordered delivery. One such way to do this is by using vector clocks [Fid88; Mat88; BP16], which will grow in size with every new replica that is modifying the data. In a dynamic environment such as the web, this vector clock will quickly grow in size and deteriorate performance. Techniques exist to reduce the communication overhead of large vectors [SK92] but typically require extra bookkeeping and storage at each replica. Furthermore, the addition and removal of clients, which happens often on the web, does not work well with a fixed-size data structure. State-based CRDTs [Sha+11a] add extra metadata on top of the actual data to be able to merge the data structures. Updates are replicated by sending the entire state and merging the two versions together. For this reason, state-based CRDTs are less suitable for client-centric replication, but they are used today between servers. Yet, they have the advantage that they are much more robust, impose little requirements on the message channel, and do not need to track clients. The third type, delta-state-based CRDTs [ASB18; ASB15; LLP16; Lin+17], are a combination of the previous two. They are essentially state-based and can always fall back to sending the full state, but in practice, they can calculate a small delta update that can be sent instead. However, often this still requires client-specific metadata such as vector clocks and inherits the same problem as operation-based CRDTs which grow in size with the number of replicas over time.

Byzantine Fault Tolerant consensus

The problem of achieving strong consistency in a distributed system with Byzantine replicas is called the Byzantine Generals Problem [LSP82] and was described already in 1982 by Lamport et al. The Byzantine Generals Problem is a fundamental issue in distributed systems that represents the challenge of reaching consensus among multiple participants (generals) in the presence of faulty or malicious actors (Byzantine actors). The problem arises when a group of generals, each commanding their own army, must agree on a coordinated strategy

(e.g., attack or retreat) to achieve a common goal, e.g., attacking a city. They can send messengers back and forth between each other to communicate the time of the attack, but this messenger has no other possibility than to go through enemy territory. He is always at risk of getting caught, killed, or replaced by the enemy, thus creating confusion and hindering consensus. If this happens, the battle strategy is sabotaged and the attack will fail. Furthermore, some generals might even be traitors who send conflicting information to other generals. The objective is to design a system where honest generals can reach agreement on a plan of action, even when faced with the presence of message loss and Byzantine actors whose actions are unpredictable and may actively attempt to undermine the consensus process. Castro and Liskov [CL99] proposed a practical solution in 1999 called Practical Byzantine Fault Tolerance (PBFT). PBFT is a leader-based consensus protocol typically used between a small number of servers (< 10). Scaling to more replicas is hard, as the protocol relies both on a leader, as well as on all-to-all communication between all replicas over a low-latency network connection. The growing interest in blockchain has led to new research in BFT consensus protocols, aiming for more scalability, throughput, and resilience. Examples are BFT-SMART [BSA14] or HotStuff [Yin+19]. However, these protocols still do not match the client-centric idea where replicas are client devices and not servers. The network link between the replicas is a mobile or WiFi connection, and clients do not always have enough resources to act as a leader for the consensus protocol. Other BFT protocols, such as Tendermint [BKM18], relax the network requirement by leveraging a multi-hop gossip [Dem+87] network between the replicas. This greatly reduces the networking cost on a single replica, as only a few network connections need to be maintained. However, this protocol is still leader-based. In a normal, server-centric environment, the failure of a leader is not very severe, as a new leader can be elected quickly. This is especially the case for Tendermint, which will rotate the leader often as part of the normal execution. Yet, in a client-centric environment, leader failures are much more common, especially when the network is unreliable. Leaders that fail often, and the remaining replicas taking some time to elect a new leader, will form a performance bottleneck in the system. If leaders fail often, more time will be spent on leader election than actually reaching consensus.

1.2 Research goals

This thesis investigates whether we can use this new paradigm of client-centric replication for collaborative web applications. In such collaborative web applications, multiple users, with multiple clients, are working concurrently on a shared data set. Current local-first software faces metadata explosion challenges when replicating and merging data peer-to-peer, while blockchain replicas these days are run on large server devices because of both computational as well as

storage requirements. Both solutions are not really suitable for client devices with unstable network conditions today.

Interactive and resilient replication

Client-centric replication should provide both interactive and resilient replication. *Interactive* replication means that the user should not notice any delay in the replication process. This is crucial for collaborative applications, as users expect immediate replication of changes. To quantify this noticeable delay, we look into the research of Nielsen on usability engineering [Nie93]. Nielsen defines a threshold of 100 milliseconds for the user to not notice any delay for local interactions. This threshold is easily met by using a local-first approach, which replicates the data to the client devices. As all data is locally stored, queries can be performed very quickly. For remote interactions, users generally expect it to take 1 to 2 seconds. They become annoyed after 5 seconds, and will often leave the application after 10 seconds. These objectives are already more difficult, especially if strong consistency is required, this will require global coordination between all replicas, limiting the overall scalability. The systems' scalability in this dissertation is directly impacted by the 5-second upper bound before users become annoyed. Furthermore, the aim is not to reach this goal for the average user, instead, most users should experience this acceptable performance [DeC+07]. For this reason, we mostly look at the 99th percentile instead of the averages.

Only having interactive performance during normal operation without failures is not enough. As we focus on client-centric replication, the network is not very reliable, and short-term interruptions will often happen. The replication protocol should, therefore, also be *resilient*. By resilient we mean that interactive performance will be restored very quickly after the failure is resolved. Note that in later chapters, we use the words robust and resilient interchangeably. We again look into Nielsen's work and take 5 to 10 seconds as an absolute maximum to restore interactive performance.

Limited storage overhead

The nature of the web, and client-centric replication in general, means that there is no fixed set of replicas that do not change often. In traditional server-centric replication, each server in a chosen set is assigned an identifier. These servers can replicate data between each other using their static identifiers. After some time, a new replica might be added, or an existing replica can be removed, but this is in general a controlled process with a human in the loop. On the web, we have two levels of identifiers. First, there are users, these are humans who should have access to the data and want to replicate it among themselves. Second, every user has several devices on which they want to replicate their data. To

further complicate the situation on the web, a single device can have multiple replicas because multiple browsers are used in parallel, an incognito tab is used at some point, or simply by clearing the browser cache. This last example might seem similar to removing a replica, however, unlike the server-centric case where a human is actively executing a command to remove the replica, and therefore notifies the other replicas, there is no such notification on the web. Thus, other replicas cannot differentiate between a removed replica and one that is merely offline for an extended period. Replication protocols that use client-identifiers, for example by using vector clocks, will therefore suffer from the so-called problem of metadata explosion [Llo+13; GPS16]. These protocols must track the growing number of client identifiers to ensure correctness, resulting in metadata size inflation over time. Our goal for the protocols in this thesis is to not rely on these identifiers, which allows us to keep the interactive and robust performance over time as the size of the metadata will be constant over time.

Data consistency

Data should be replicated in a provably consistent manner. The kind of consistency can be varied, depending on the application itself: eventual consistency [Dem+94], strong eventual consistency [Sha+11a], and strong consistency. With (strong) eventual consistency, replicas can temporarily diverge for a while, but they will eventually converge to the same value. With eventual consistency, correct replicas that have delivered the same updates eventually reach an equivalent state. With strong eventual consistency, correct replicas that have delivered the same updates have an equivalent state. The difference between eventual consistency and strong eventual consistency is subtle but important. An eventually consistent system diverges and later rolls back to converge again. This rollback requires some form of coordination between the replicas, making it less scalable for true peer-to-peer client replication. Strong eventual consistency on the other hand guarantees that when two replicas received the same set of updates, they will also be in the same state, without the need to coordinate and resolve conflicts explicitly. Strong consistency is much stronger and guarantees that all replicas will see updates in the same order. However, this requires at least global coordination between a majority of the replicas.

1.3 Approach

To achieve the goal of client-centric replication for the web in different environments, we concentrate on three distinct settings based on the level of trust and required consistency level: (E1) strong eventual consistency in a trusted setting, (E2) strong consistency in a Byzantine setting, and (E3) strong

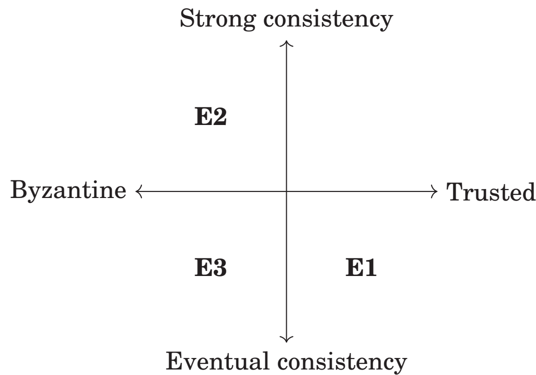


Figure 1.2: Overview of our three solution environments and contributions with respect to the level of trust and the required consistency level.

eventual consistency in a Byzantine setting. Figure 1.2 shows an overview of these environments and how they relate with each other based on trust and consistency level. One quadrant is empty: strong consistency in a trusted setting. This represents the textbook case for consistency protocols such as two-phase commit. This dissertation starts with the easiest case of the three environments: strong eventual consistency in a trusted setting (E1). Afterward, we move to the other side of the spectrum: strong consistency in a Byzantine setting (E2). This is the most difficult case and here we will push the limits for client-centric replication in terms of performance and scalability. Finally, we move to the last environment: strong eventual consistency in a Byzantine setting (E3). By relaxing the consistency level to eventual consistency, we get a more natural fit for client-centric replication on the web, with many mobile devices. Which also allows for disconnected operation or ad-hoc collaboration.

For each of these environments, we design a novel replication protocol focusing on interactive and resilient replication without too much storage overhead or performance deterioration over time. We implement these protocols in three distinct browser-based middleware systems and evaluate our protocols in realistic settings with failures. In our comparative evaluations, we demonstrate resilience, scalability, and storage overhead improvements compared to existing protocols.

E1: Strong eventual consistency in a trusted setting. (Figure 1.3)

The first environment focuses on a trusted setting, where all replicas are trusted and have full access to the replicated data. An example use case is a collaborative graphical document application. Multiple users can edit the document simultaneously, including when they are offline. Multiple concurrent

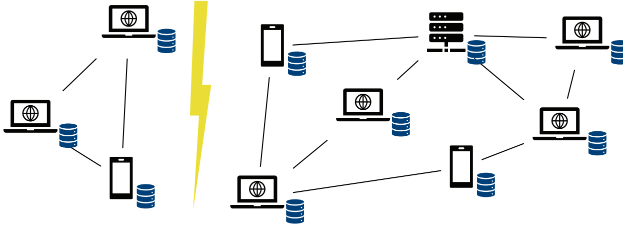


Figure 1.3: (E1) Peer-to-peer network of trusted devices. Each device has a local copy of the data and can continue making changes even with network partitions.

edits to the same document should be possible, and conflicts should be resolved in a fine-grained manner. This means that concurrent edits to two different parts of the document should be possible and that both edits should be reflected in the final document eventually. Conflicts to the same property should be resolved automatically, to not burden the user. The replicas both replicate their changes to a central server but also want ad-hoc peer-to-peer replication. This requirement is crucial when users are physically together, such as in the same meeting room, to reduce the latency impact of utilizing a central server. It becomes even more important when that central server is not reachable, e.g., when the users are working together on an airplane. In this case, the users can still replicate their changes with each other and see each other's changes, even though both are offline from the perspective of the server.

E2: Strong consistency in a Byzantine setting. (Figure 1.4)

The second environment focuses on an untrusted setting, where the replicas are not trusted and can be malicious. As an example use case, we consider an integrated loyalty program between small, local businesses or merchants at a farmer's market. Integrated loyalty programs prove more effective than traditional single-merchant loyalty programs [FT16]. However, as there is an

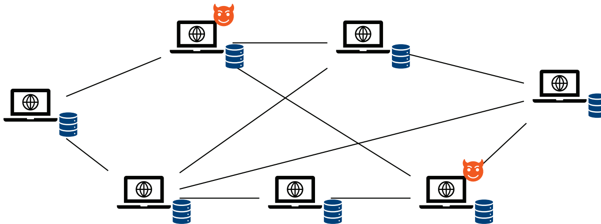


Figure 1.4: (E2) Peer-to-peer network of untrusted clients, reaching strong consistency via Byzantine Fault Tolerant consensus.

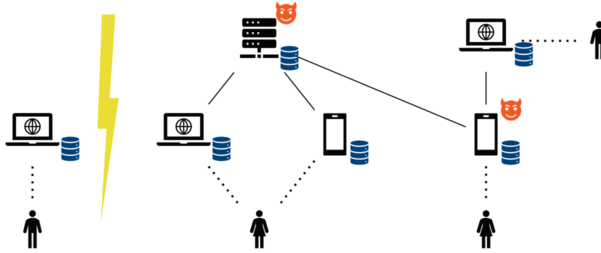


Figure 1.5: (E3) Hybrid architecture of a peer-to-peer network with a centralized server. Not every replica may have access to the actual plain data, and some devices can even be malicious.

inherent distrust between the merchants, normally a trusted third party has to be involved to keep track of the loyalty points. This brings an extra burden and costs to the merchants. We instead propose a decentralized peer-to-peer network between the merchants to keep track of the loyalty points between them. Contrary to the first environment, eventual consistency no longer suffices for this use case. Strong consistency is required to solve the double-spending problem, where a customer tries to spend the same loyalty points twice at different stores. This environment does not support offline use, as a supermajority, typically two-thirds of the replicas, must always be online for consensus to be reached. However, our focus is on a lightweight and robust protocol, which can quickly be set up on a mobile device such as a laptop or tablet with a wireless internet connection.

E3: Strong eventual consistency in a Byzantine setting. (Figure 1.5)

The third environment returns back to the eventually consistent model of E1, but now in an untrusted setting. Similar use cases as E1 apply, however, the replicas are now not necessarily trusted. Ideally, data is only replicated between trusted devices and collaborators, however, this is not always possible. For example, a user might make edits to a shared document in the afternoon, afterwards, they shut down their laptop. Another user then opens the document later in the evening, but since the first user is no longer online, the new edits cannot be replicated. Although the client-centric approach is beneficial for performance, security, and availability when users are close to each other, if users are never online at the same time, no data can be replicated. For this reason, having a central server, which is online most of the time, can act as a central replication point that all other replicas can use to replicate their changes to each other without having to be online at the same time. Nonetheless, there is no reason to trust this server; data should not be readable by the server, and the server should not have the ability to modify the data.

1.4 Contributions

This dissertation provides three key contributions to client-centric replication, spanning two dimensions: *trusted* and *untrusted* environments, as well as *strong eventual consistency* and *strong consistency*:

1. We present OWebSync, which operates in a trusted environment and uses eventual consistency (E1). It introduces a novel Conflict-free Replicated Data Type protocol, fully state-based and supporting fine-grained delta-merging without tracking individual clients.
2. We present BeauForT, which operates in a Byzantine environment with strong consistency (E2). We introduce a novel approach to synchronizing consensus votes between replicas, resulting in a fully leaderless protocol.
3. We propose a protocol for replication in an untrusted environment that offers eventual consistency (E3), without the need to keep track of individual transactions or clients. It is also the first eventually consistent protocol to support concurrent data updates and access control policy changes.

1.5 Overview

The remainder of this dissertation is structured as follows.

Chapter 2 presents and evaluates the first contribution, OWebSync, which focuses on a trusted environment and offers strong eventual consistency. This work was published in *IEEE Transactions on Parallel and Distributed Systems* [JLJ21]. The original publication focused on the evaluation in a client-server architecture. A follow-up demo at the 2021 International Conference on Service-Oriented Computing [JLJ22a] showed that the protocol can also be used in a peer-to-peer architecture.

Chapter 3 presents and evaluates the second contribution, BeauForT, which focuses on a Byzantine environment and offers strong consistency. This work was also published in *IEEE Transactions on Parallel and Distributed Systems* [Jan+23a].

Chapter 4 presents and evaluates the third contribution, which focuses on an untrusted environment and offers strong eventual consistency. This work was published in the *3rd International Workshop on Distributed Infrastructure for the Common Good* [JLJ22b].

Finally, *Chapter 5* concludes this dissertation by discussing the limitations of the contributions and outlining directions for future research.

Other publications from the author:

Kristof Jannes, Bert Lagaisse, and Wouter Joosen. “The Web Browser as Distributed Application Server: Towards Decentralized Web Applications in the Edge”. In: *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*. EdgeSys ’19. Dresden, Germany: Association for Computing Machinery, 2019, pp. 7–11. DOI: 10.1145/3301418.3313938

Kristof Jannes, Bert Lagaisse, and Wouter Joosen. “You Don’t Need a Ledger: Lightweight Decentralized Consensus Between Mobile Web Clients”. In: *Proceedings of the 3rd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. SERIAL ’19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 3–8. DOI: 10.1145/3366611.3368143

Martijn Sauwens, Kristof Jannes, Bert Lagaisse, and Wouter Joosen. “SCEW: Programmable BFT-Consensus with Smart Contracts for Client-Centric P2P Web Applications”. In: *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC ’21. Online, United Kingdom: Association for Computing Machinery, 2021. DOI: 10.1145/3447865.3457965

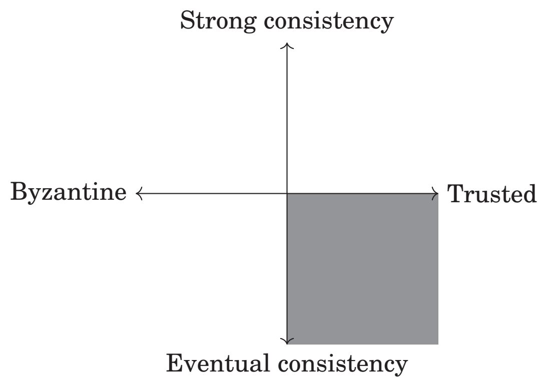
Martijn Sauwens, Emad Heydari Beni, Kristof Jannes, Bert Lagaisse, and Wouter Joosen. “ThunQ: A Distributed and Deep Authorization Middleware for Early and Lazy Policy Enforcement in Microservice Applications”. In: *Service-Oriented Computing*. Springer International Publishing, 2021, pp. 204–220. DOI: 10.1007/978-3-030-91431-8_13

Pieter-Jan Vrielynck, Emad Heydari Beni, Kristof Jannes, Bert Lagaisse, and Wouter Joosen. “DeFIRED: Decentralized Authorization with Receiver-Revocable and Refutable Delegations”. In: *Proceedings of the 15th European Workshop on Systems Security*. EuroSec ’22. Rennes, France: Association for Computing Machinery, 2022, pp. 57–63. DOI: 10.1145/3517208.3523759

Kristof Jannes, Vincent Reniers, Wouter Lenaerts, Bert Lagaisse, and Wouter Joosen. “DEDACS: Decentralized and dynamic access control for smart contracts in a policy-based manner”. In: *Proceedings of the 38th Annual ACM Symposium on Applied Computing*. SAC ’23. Tallinn, Estonia: Association for Computing Machinery, 2023. DOI: 10.1145/3555776.3577676

2

Eventual Consistency in a Trusted Setting



In this chapter we present 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 at 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.

This chapter is strongly based on our published journal article in *IEEE Transactions on Parallel and Distributed Systems* in 2021 [JLJ21]. The original publication and this chapter focus on client-server interactions. However, after this publication, we have extended this middleware to support also peer-to-peer interactions in addition to the client-server approach. We replaced the WebSocket connections between the server and the clients with WebRTC connections between the clients. No changes are made to the synchronization mechanism mentioned in this chapter. We demonstrated that this peer-to-peer approach works on a global scale with live participants on the demonstrations track of the *International Conference on Service-Oriented Computing* in 2021, which was organized online due to COVID [JLJ22a]. Upon the publication of this work, we have also open-sourced the JavaScript-based middleware to encourage further progress in the domain. Our evaluation methodology, use case, and evaluation framework for end-to-end performance testing have also been adopted by others [Wei+22].

This chapter generally illustrates that a purely state-based replication approach can work interactively for client-centric and peer-to-peer replication. The same underlying state-based gossip protocol will also be used in the frameworks proposed in Chapter 3 and Chapter 4. In Chapter 4 we modified this protocol by replacing the normal Merkle-tree with a Modified Merkle-Patricia-Tree based on random identifiers rather than the actual structure of the data. This ensures that the tree remains well-balanced, even when the actual data structure is not. In future work, this approach can also be applied to the solutions presented in this chapter to improve replication times for unbalanced tree-structured documents.

© 2021 IEEE. Reprinted, with permission, from Kristof Jannes, Bert Lagaisse, and Wouter Joosen. "OWebSync: Seamless Synchronization of Distributed Web Clients". In: *IEEE Transactions on Parallel and Distributed Systems* 32.9 (2021), pp. 2338–2351. DOI: 10.1109/TPDS.2021.3066276.

2.1 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. Although modern HTML5 applications can work offline, data is typically only available on the server, not on the client-side, and an internet connection is required to modify this data.

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 through replication.

This chapter 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 [Nie93] 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 the 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 send the updates as operations to all replicas. Operational Transformation, as used in Google Docs[1], 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 to transform the operations for other clients to deal with concurrent changes. Commutative Replicated Data Types [Pre+09; Sha+11b], as used in SwiftCloud [Zaw+13; Pre+14], Yjs[2] [Nic+15; Nic+16] and Automerge[3] [KB17; KB18; HK20], 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 [Sha+11b] are resilient against message loss but have often been considered 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[4]. Merkle Search Trees [AT19] are proposed as a solution to the high bandwidth usage. It uses Merkle-trees [Mer88] to replicate a basic key-value store like in Dynamo [DeC+07]. The solution works in large systems with low rates of updates for asynchronous background synchronization between backend servers; it is not suited for interactive groupware. Delta-state-based Conflict-free Replicated Data Types [LLP16], as used in Legion[5] [Lin+17], 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 writer 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 chapter, 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 [Bra14], 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

[1] https://support.google.com/docs/answer/2494822
[2] https://github.com/y-js/yjs
[3] https://github.com/automerge/automerge
[4] https://docs.basho.com/riak/kv
[5] https://github.com/albertlinde/Legion

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 chapter is structured as follows. Section 2.2 provides two motivating case studies and provides background on synchronization mechanisms such as CRDTs. Section 2.3 describes the underlying data model based on CRDTs and Merkle-trees. Section 2.4 presents the deployment and synchronization architecture together with two performance optimization tactics. Section 2.5 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 2.6 and then we conclude.

2.2 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.

2.2.1 Case studies

The motivation and requirements emerged from two case studies from our applied research projects, that have also been used for the evaluation of the middleware. The first case study is an online software service from eWorkforce, a company that provides technicians to install network devices for different telecom operators at their customers' premises. The second company, eDesigners, offers a web-based design environment for graphical templates that are applied to mass customer communication.

eWorkforce. eWorkforce has two kinds of employees that use the online software service: the help desk operators at the office and the technicians on the road. The help desk operators accept customer calls, plan technical intervention jobs, and assign them to a technician. The technicians can check their work plan on a mobile device and go from customer to customer. They want to see the details of their next job wherever they are and 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 afterward.

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.

eDesigners. 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, and 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 to not interrupt the user.

This case study requires that the application is always available, and 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.

2.2.2 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 (OT) and Conflict-free Replicated Data Types (CRDTs).

Operational Transformation. OT [EG89] 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 very resilient against message loss and long-lasting offline situations, as this leads to low performance [KK10].

Conflict-free Replicated Data Types. CRDTs [Sha+11a; Sha+11b] 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 [Sha+11a] 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 to 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 [Sha+11a] 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[6] [ASB15; ASB18] 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 on 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.

[6] https://github.com/peer-base/js-delta-crdts

A variant of δ-CRDTs, called Δ-CRDTs [LLP16], 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 [Ene+16; Ene+19]. 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. This improves the network usage compared to normal δ-CRDTs, however, 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.

2.2.3 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.

LWWRegister. A Last-Write-Wins Register [Sha+11b] 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.

ORSet. An Observed-Removed Set [Sha+11b; Sha17] 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 of the same item will result in that 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.

Merkle-trees. Merkle-trees [Mer88] 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 that are already equal will have equal hashes at their top nodes, so they do not need further verification.

2.3 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.

2.3.1 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 consists 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. [Sha+11b]. The ORMap is a recursive data structure that represents a map containing a mapping from strings to other ORMaps or LWWRegisters.

2.3.2 Observed-Removed Map

The ORMap is implemented starting from a state-based Observed-Removed Set [Sha17]. 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

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

```
    KV                                          ▷ Key-value store
    p_0..p_n                                    ▷ Array representation of a path in the tree
 1: state:
 2:     O ← ∅                                   ▷ Observed set with tuples (key, tag, hash)
 3:     R ← ∅                                   ▷ Removed set with tags
 4:     PATH                                    ▷ The path of this ORMap
 5: query: GET (p_0..p_n)
 6:     if ∃o ∈ O : o.key = p_0 then
 7:         c ← KV.GET(PATH + o.key)
 8:         return c.GET(p_1..p_n)
 9:     return ⊥
10: update: SET (p_0..p_n, value)
11:     if ∃o ∈ O : o.key = p_0 then
12:         c ← KV.GET(PATH + o.key)
13:         c.SET(p_1..p_n, value)
14:         o.hash ← c.hash
15:     else
16:         if LEN(p_0..p_n) = 1 ∧ IS_PRIMITIVE(value) then
17:             c ← NEW_LWWREGISTER(PATH + p_0)
18:         else
19:             c ← NEW_ORMAP(PATH + p_0)
20:         c.SET(p_1..p_n, value)
21:         O ← O ∪ {(p_0, UNIQUE(), c.hash)}
22: update: REMOVE (p_0..p_n)
23:     if ∃o ∈ O : o.key = p_0 then
24:         if LEN(p_0..p_n) = 1 then
25:             O ← O \ {o}
26:             R ← R ∪ {o.tag}
27:         else
28:             c ← KV.GET(PATH + o.key)
29:             c.REMOVE(p_1..p_n)
30:             o.hash ← c.hash
31: update: REMOVE_WITH_TAG (p_0..p_n, tag)
32:     if LEN(p_0..p_n) = 0 then
33:         if ∃o ∈ O : o.tag = tag then
34:             O ← O \ {o}
35:             R ← R ∪ {o.tag}
    (Continues in Algorithm 2)
```

Algorithm 2 Simplified implementation of the query-, update- and merge operations of an ORMap with a Merkle-tree for synchronization. *(continued)*

```
36:     else if ∃o ∈ O : o.key = p_0 then
37:         c ← KV.GET(PATH + o.key)
38:         c.REMOVE_WITH_TAG(p_1..p_n, tag)
39:         o.hash ← c.hash
40: merge: MERGE (p_0..p_n, remote)
41:     N ← ∅                                  ▷ paths that need synchronization
42:     if LEN(p_0..p_n) = 0 then
43:         R ← R ∪ remote.R
44:         O ← {o ∈ O : o.tag ∉ R}
45:         for all ro ∈ remote.O : ro.tag ∉ R do
46:             if ∃o ∈ O : o.key = ro.key then
47:                 if o.tag ≠ ro.tag then
48:                     if ro.tag > o.tag then
49:                         o.tag ← ro.tag
50:                     else
51:                         N ← N ∪ {PATH}
52:                 if o.hash ≠ ro.hash then
53:                     N ← N ∪ {PATH + o.key}
54:             else
55:                 N ← N ∪ {PATH + ro.key}
56:     else
57:         if ∃o ∈ O : o.key = p_0 then
58:             c ← KV.GET(PATH + o.key)
59:             N ← N ∪ c.MERGE(p_1..p_n, remote)
60:             o.hash ← c.hash
61:         else
62:             if LEN(p_0..p_n) = 1
63:                 ∧ TYPEOF(remote) = LWWRegister then
64:                 c ← NEW_LWWREGISTER(PATH + p_0)
65:             else
66:                 c ← NEW_ORMAP(PATH + p_0)
67:             N ← N ∪ c.MERGE(p_1..p_n, remote)
68:             O ← O ∪ {(p_0, c.tag, c.hash)}
69:     return N
```

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.

Algorithm 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 `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` operations are also similar but require updating the hash to keep the Merkle-tree up-to-date. The internal `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 the next step, the synchronization protocol will use those returned 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 2.4. 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 form 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 [Sha+11b; Sha+11a]. The partial order of the modified ORMap

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

Figure 2.1: JSON data structure of the eDesigners case study.

defined here is similar to the ORSet [Sha17], 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 Algorithm 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.

In reality, the merge is split into two phases. However, the first phase is only used to determine which parts of the child CRDTs have to be merged further and which parts are already equal between the two replicas. The actual merge is done in the second phase when the leaf CRDTs are received. Only then will all the hashes be updated all the way to the root CRDT. While we do not provide a formal proof that the CRDT properties are kept intact, each time we update the state and metadata of the CRDTs, we are following the rules of existing CRDTs (ORSet and LWWRegister), which are formally proven.

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. Figure 2.1 present a JSON data structure of a drawing with one rectangle object. Figure 2.2 represents

```
{
    "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"
    }
}
```


Figure 2.2: Internal structure of two key-values that represent `object36` and the property `top` of the JSON data structure in Figure 2.1. We use a JSON notation here, however, in practice, these two key-values are stored in a binary format in a key-value store.

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.

2.3.3 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 nested maps 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, losing 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 accumulates 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 vector grows in size with the number of clients that make changes to this ORMap. Since we target a dynamic environment such as the web, we opted for tombstones, which can be garbage collected after a sufficiently long time. Web clients come and go, without

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 conflict will be resolved by keeping the ORMap and discarding the LWWRegister.

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.

Ordered lists are 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 not needed in our use cases. 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 [Sha+11b; Roh+11] and could be added in future work. Adding new kinds of CRDTs to the data model is straightforward. An existing CvRDT can be used as is, except for an extra hash to be part of the Merkle-tree. For a CRDT that represents a leaf value (e.g., a Multi-Value Register [Sha+11b]), 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.

2.4 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 supporting 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.

2.4.1 Overall architecture

The middleware architecture is depicted in Figure 2.3 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 lightweight process, which listens to incoming web requests. The server is only responsible for data synchronization, it does not run application logic. Both the client and the server have a key-value store to persist data, and they communicate using only web-based HTTP traffic and WebSockets [Hic12]. All communication messages are sent and received inside the client and server subsystems using asynchronous workers. The tags in the ORMap are

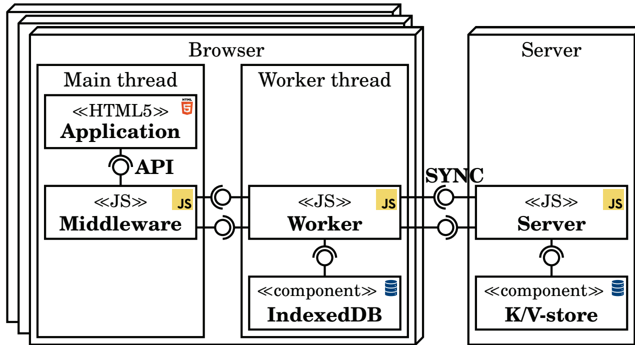


Figure 2.3: Overall architecture of the OWebSync middleware.

UUIDs [LSM05] and we use the MD5[7] [Riv92] 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.

2.4.2 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)
```

[7] We do not need a cryptographically secure hash algorithm, as every replica is trusted in this chapter. Furthermore, MD5 is the fastest hashing algorithm which is supported by browsers natively.

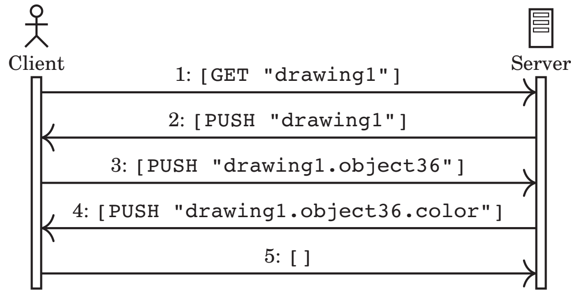


Figure 2.4: Synchronization protocol when another client updated the color. A `GET` message only sends the path and hash value, and a `PUSH` message also sends the respective CRDT. E.g., for message 3, this is the first CRDT in Figure 2.2.

The object `"drawing1"` is fetched from disk 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. Modifications to this object are not synchronized, nor are updates from other replicas reflected in this object. The last line (`set`) will save the modified object and replicate it to other replicas asynchronously. 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")
```

To reduce possible conflicts and outdated representations, a developer should use `set` every time the user makes a change, and should use `listen` to update the in-memory representation each time updates from other replicas are received.

2.4.3 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.

Algorithm 3 Specification of the synchronization protocol, using the ORMap specified in Section 2.3. Some details are abstracted for conciseness.

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

The protocol, depicted in Algorithm 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 sent 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 Algorithm 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 exist, 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 the 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.

Figure 2.4 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.

2.4.4 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.

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.

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 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 one batch after a full pass of a whole level of the tree has occurred. On 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.

2.5 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 chapter 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[8] and Automerge[9]

[8] https://github.com/y-js/yjs
[9] https://github.com/automerge/automerge

which use operation-based CRDTs, and ShareDB[10] which makes use of OT. We used Legion [Lin+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 [KB17]. Legion is the implementation of Δ-CRDTs of van der Linde [LLP16; Lin+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[11] 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[12] 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. So, this experiment consists of 10 VMs in total, one server VM, one monitoring VM, and 8 client VMs that each can represent up to 3 clients isolated with Docker. They are deployed in the same data center, but Azure does not offer insight into whether two VMs are located on the same physical machine or not. VMs in Azure have their clocks synchronized with the host machine, which is synchronized with the internal Microsoft time servers. The Linux `tc` tool [Alm99] 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 [Ope19].

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.

2.5.1 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

[10] https://github.com/share/sharedb
[11] https://github.com/fabricjs/fabric.js
[12] https://azure.microsoft.com

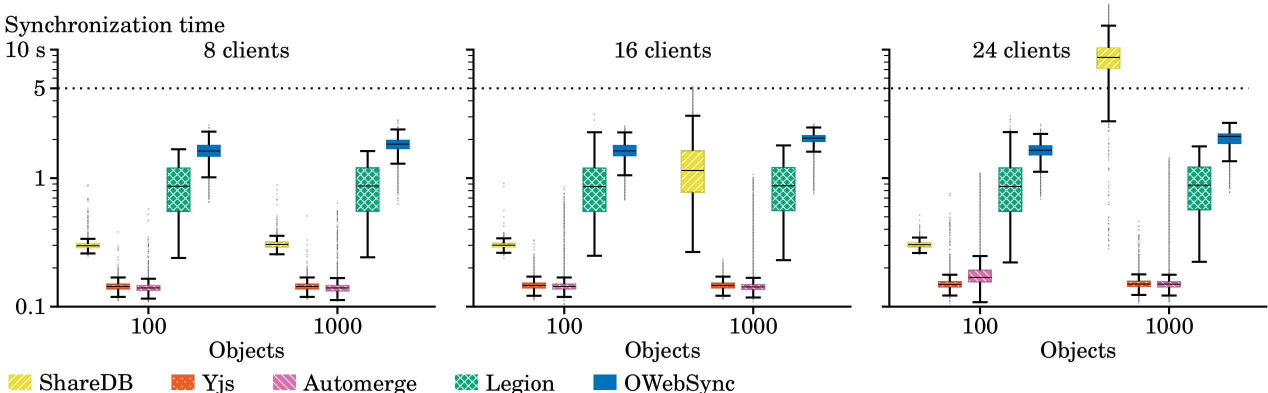


Figure 2.5: Aggregated boxplots containing the times to synchronize an update to all other clients in the online scenario. Each boxplot contains all 10 iterations for each of the 30 tests in the fully online situation. To compare technologies that have results of the same order of magnitude, as well as results in different orders of magnitude, we opted for a logarithmic Y-axis.

objects in a single shared data set. One such object was shown in Figure 2.1 in Section 2.3 and has 7 attributes. The total depth of the tree is four (root – `drawing1` – `object36` – `top`). The root has one child, while `drawing1` has either 100 or 1000 children. And these children have 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[13]. But in practice, latency starts to increase significantly when the number of users exceeds 10 [DI16]. 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 Figure 2.5.

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 seconds 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,

[13] https://support.google.com/docs/answer/2494822

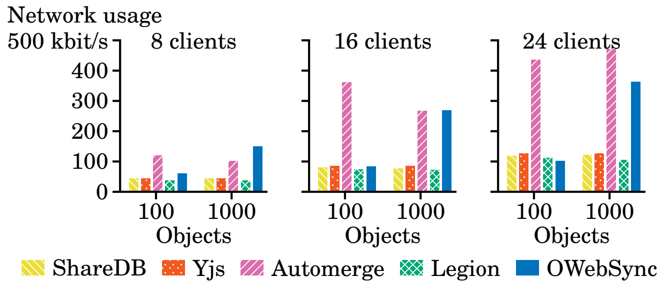


Figure 2.6: Network usage per client for each test in the online scenario.

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 (Figure 2.6). 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 by 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[14]. The server consumes about 8.7 Mbit/s, which is acceptable for a typical data center. The data structure has an important effect on the network usage. One might create a tree-structure with few nodes which have many children. This will make the Merkle-tree less useful since the metadata of all the children needs to be exchanged to be able to determine which children are updated. This can be seen in Figure 2.6 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 2.1. The average client-side CPU usage for OWebSync is 9%,

[14] https://www.speedtest.net/reports/united-states/2018/Mobile/

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.

Interpretation and discussion. For interactive web applications and groupware, usability guidelines [Nie93; Nie10] 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* [DeC+07]). 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. [DI16] for Google Docs, which also uses OT. The other technologies stay well below 5 seconds in the online scenario and can be called interactive.

2.5.2 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.

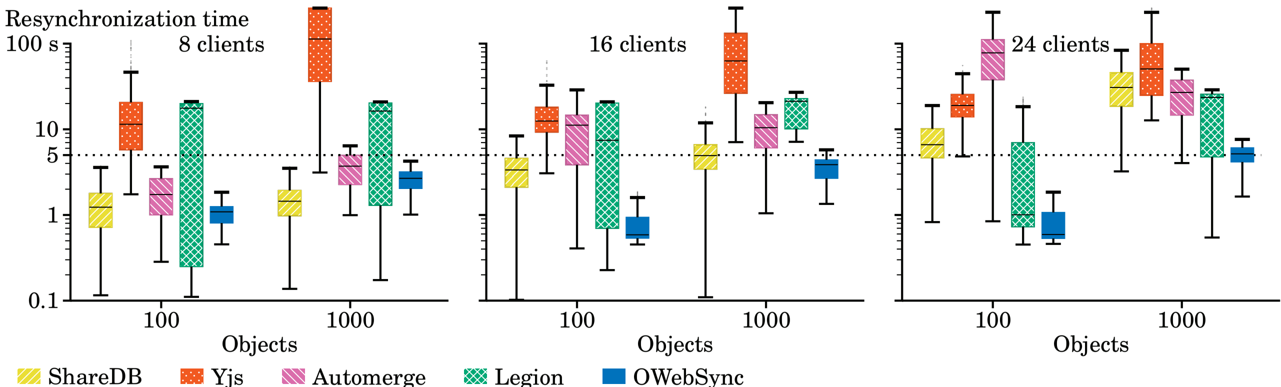


Figure 2.7: Boxplots of the time it takes for an update during the failure scenario to be received by all clients. The time before a client notices the network connection is reestablished is not taken into account. Note that the median here means that only 50% of all missed updates are synchronized to all clients. Only at the upper whisker, most of the missed updates are synchronized.

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 Figure 2.7, 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 Figure 2.8 show the resynchronization times on the y-axis, without the offline time during the network disruption, for each update done at a given moment during the test timeline. 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.

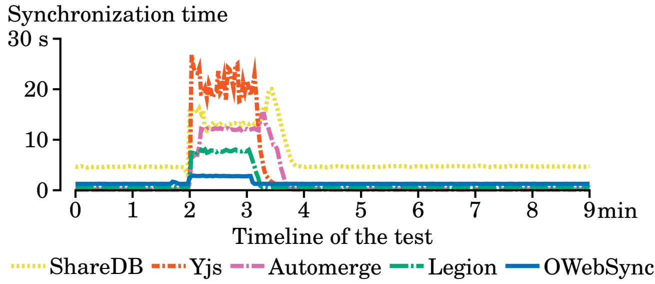


Figure 2.8: Mean time to synchronize updates in case of a network disruption between minutes 2 and 3 for the test with 24 clients and 1000 objects.

2.5.3 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. Figure 2.9 shows the total data size on the server over time while several users are joining and leaving. The size of the data

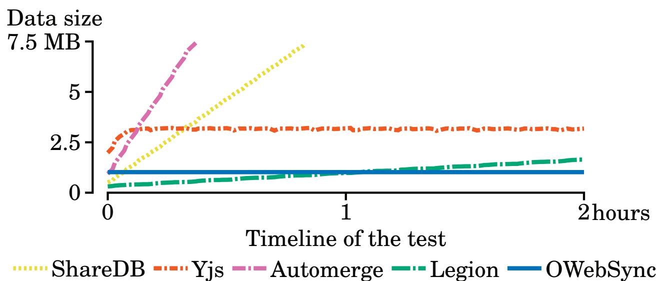


Figure 2.9: Evolution of the total data size on the server.

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[15]. This operation is not safe and clients that were offline for a longer time might end up in an inconsistent state or lose data.

Notice that the replicas in this experiment are only moving objects around, i.e., they modify existing data. If they would also remove objects, then OWebSync would have a small increase in data size over time, as tombstones need to be kept. However, even for removing large subtrees, only a single tombstone will be stored.

The first two benchmarks are performed on a clean data set, i.e., 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.

2.5.4 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 2.1 summarizes the results in seconds of

[15] https://github.com/y-js/yjs

Table 2.1: 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

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.

2.6 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 2.2. Other text-based versioning systems such as Git[16] 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 [Bri+15] allows to replicate only part of a CRDT. This helps with bandwidth and memory consumption, as well as security and privacy [KKB19]. OWebSync allows replicating any arbitrary sub-tree of the whole CRDT tree. Hybrid approaches combining operation-based and state-based CRDTs are also possible as demonstrated by Bendy [BBR16]. 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 [BG19]. But this only treats transient network disruptions where clients will come back online

[16] https://git-scm.com/

eventually, which is not necessarily the case for web clients. Pure operation-based CRDTs [BAS17] only have to send the actual operations, rather than datatype-specific messages generated internally. This works well for commutative CRDTs, but non-commutative ones require the use of their Tagged Causal Stable Broadcast middleware, consisting of a partially ordered log.

There exist other Replicated Date Types as well. Strong Eventually Consistent Replicated Objects (SECROs) [De +19] are similar to operation-based CRDTs but do not impose restrictions on the commutativity of operations. However, by doing so, they need a global total order and cannot quickly recover from network disruptions. Upon reconnection, it may be the algorithm has to compute all possible orderings to choose one, leading to high latencies and poor scalability. Explicitly Consistent Replicated Objects (ECROs) [De +21] avoid this problem of computing the orderings at run-time but have a static analysis phase before. However, they only guarantee Explicit Consistency [Bal+18] instead of SEC. Cloud Types [Bur+12] are similar to CRDTs and can be composed, but only offers eventual consistency. Mergeable Replicated Data Types [Kak+19] uses invertible relational specifications defined by the programmer to derive a three-way merge function. They are only defined for data types built as views of relations on sets.

Distributed data systems and NoSQL systems. Based on the Amazon Dynamo paper [DeC+07], 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[17] [LM10] 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[18] and MongoDB[19] 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[20], Akka[21] and Redis [Biy17]. Besides those commercial products, several research projects have emerged. Merkle Search Trees (MSF) [AT19] 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-

[17] https://cassandra.apache.org
[18] https://couchdb.apache.org
[19] https://www.mongodb.com/
[20] https://docs.riak.com/riak/kv/
[21] https://doc.akka.io/docs/akka/current/distributed-data.html

tree. MSF is targeted to asynchronous background synchronization between backend servers, and not for interactive groupware with replication to the clients. Antidote[22] is a research project to develop a geo-replicated database over worldwide 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 [EEB16] uses an operation-based CRDT storage system for state management tasks for distributed container deployments. DottedDB [Gon+17] 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 [Ram+09] 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 [Zaw+13; Pre+14; Zaw+15] 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 is 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 these frameworks for native clients, there are several JavaScript frameworks made for synchronization between distributed web clients. Legion[23] [Lin+17] 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 synchronization. Automerge[24] [KB18] is a JavaScript library for data synchronization adopting the operation-based JSON data type of Kleppmann [KB17]. It uses vector clocks which grow in size with the number of clients. PouchDB[25] 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[26] is a client-server

[22] https://syncfree.github.io/antidote
[23] https://github.com/albertlinde/Legion
[24] https://github.com/automerge/automerge
[25] https://pouchdb.com
[26] https://github.com/share/sharedb

framework to synchronize JSON documents and adopts OT as a 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[27] [Nic+15; Nic+16] is a JavaScript framework for synchronizing structured data and supports maps, arrays, XML, and text documents. All data types also use operation-based CRDTs for synchronization. Swarm.js[28] is a JavaScript client library for the Swarm database, 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.

2.7 Conclusion

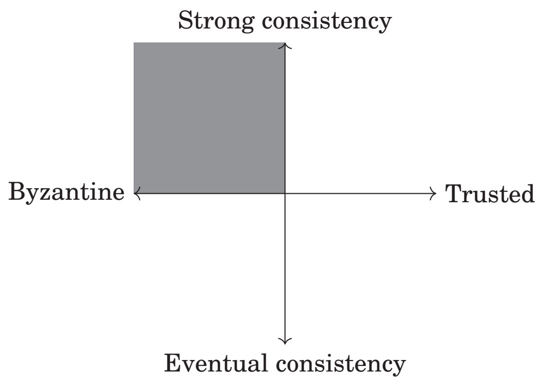
This chapter presented a web middleware that supports seamless synchronization of both online and offline clients that are concurrently editing a shared data set. Our OWebSync middleware implements a generic data model, based on JSON, that combines state-based CRDTs with Merkle-trees. This allows us 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.

[27] https://github.com/y-js/yjs
[28] https://github.com/gritzko/swarm

3

Strong Consistency in a Byzantine Setting



In this chapter, we present BeauForT, a purely browser-based platform for decentralized BFT consensus in client-centric, community-driven applications. Existing consensus protocols using either all-to-all communication or leader-based gossip suffer from performance degradation in unstable network conditions. We propose a novel, optimistic, leaderless, gossip-based consensus protocol, tolerating Byzantine replicas, combined with a robust and efficient state-based synchronization protocol. This protocol makes BeauForT well suited for the decentralized client-centric web and its dynamic nature with many network disruptions or node failures.

This chapter is strongly based on our published journal article in *IEEE Transactions on Parallel and Distributed Systems* in 2023 [Jan+23a]. A parallel work [Cas+21b] also studied whether a leaderless gossip protocol can be used for consensus. In a follow-up work, we have extended the consensus framework with a generic state-machine based language to implement smart-contracts on top of an earlier version of BeauForT [Sau+21b].

© 2023 IEEE. Reprinted, with permission, from Kristof Jannes, Emad Heydari Beni, Bert Lagaisse, and Wouter Joosen. "BeauForT: Robust Byzantine Fault Tolerance for Client-centric Mobile Web Applications". In: *IEEE Transactions on Parallel and Distributed Systems* 34.4 (2023), pp. 1241–1252. DOI: 10.1109/TPDS.2023.3241963.

3.1 Introduction

Browsers and client-side web technologies offer increasing capabilities to enable fully client-side web applications that can operate independently and in a stand-alone fashion, in contrast to the server-centric model [Gar+15; JLJ19a]. Mobile applications are also more and more purely web-based clients, where the execution environment is just a browser-based process for a mobile web application. Web 3.0 can be defined as the decentralized web where users are in control of their data, and that replaces centralized intermediaries with decentralized networks and platforms. Community-driven, decentralized networks can open the road to many use cases for the sharing economy [Mad+19] or shared loyalty programs for local communities [JLJ19b]. Such client-centric collaborations can, for example, enable a small network of merchants in a local shopping street, or at a farmer's market to set up a shared loyalty program between the merchants in an ad-hoc fashion. These small-scale, specialized collaborative networks can empower motivated citizens to bring value to their local community, without involving an incumbent big-tech company that can change the rules unilaterally at any moment.

However, current state-of-the-art peer-to-peer data synchronization frameworks for the browser such as Legion [Lin+17], Automerge [KB17; KB18; Kle+22], and OWebSync [JLJ21] focus on full replication and eventual consistency between trusted clients. Each replica can modify all data, and all modifications are automatically replicated to all replicas. These protocols lack Byzantine Fault Tolerance (BFT). Yet, they are easy to set up and applications from *trusted* parties can leverage these to synchronize and modify a shared data set between them.

Decentralized interactions between *distrusting* parties can be enabled by using a classical BFT consensus protocol such as PBFT [CL99], BFT-SMART [BSA14], or HotStuff [Yin+19]. These classical BFT protocols are very fast and have a high throughput, but typically assume server-to-server communication with low-latency network connections, and assume every node is connected to all other nodes. Other classical BFT consensus protocols, such as Tendermint [BKM18], relax the requirement that every node is connected to every other node. Nakamoto consensus [Nak08], used in several blockchains such as Bitcoin, relaxes this requirement and only requires a loosely coupled network. However, blockchains based on Nakamoto consensus are too slow for many use cases. They need minutes, or even an hour, to confirm a transaction with high probability. Moreover, they consume a large amount of energy and need a lot of processing power. Ethereum [But+13] started as a blockchain using Nakamoto consensus but recently moved to Proof-of-Stake with Gasper [But+20] as consensus protocol. Its finality is still expected to be around 15 minutes. At last, Avalanche consensus [Roc+19] tries to solve the scalability problem by using the concept of

meta-stability. Only a small subset of replicas needs to be sampled in each round to reach consensus. However, a replica still needs a connection to every other replica, as the replicas that they need to sample change continuously.

Ultimately, a decentralized mobile application should be able to run in a robust and resilient way over a network of online client devices such as smartphones. We target an environment with 10-100 lightweight and mobile web clients. Such devices have a permanent yet unstable internet connection over a data subscription and are operational and reactive most of the time. I.e., we assume those mobile devices always have a 3G or 4G connection, but this kind of connection is less stable than a wired connection and short disruptions are commonplace. Many existing protocols such as PBFT [CL99], RBFT [AMQ13], BFT-SMART [BSA14], DBFT [Cra+18], HotStuff [Yin+19], or SBFT [Gue+19] use all-to-all communication, which is simply not possible in a web-based environment. A browser can keep a connection open to 10-20 other browsers, but after that performance deteriorates quickly. Alternatively, there exist gossip-based protocols, such as Tendermint [BKM18], that do not require a connection to every other node. However, Tendermint is leader-based, which in practice means that when this leader fails, consensus will be delayed until the next leader is elected (this can take 10 seconds). Moreover, these existing BFT protocols are designed for server-like infrastructure that has lots of processing power, storage space, and a stable, low-latency network connection. The motivated citizens in our envisioned use cases do not have this kind of knowledge, budget, and infrastructure available to set up a private network of servers, that are running a BFT protocol between them. These citizens rather want to use their existing hardware such as a low-end computer, or a mobile device.

In this chapter, we present BeauForT, a novel peer-to-peer data synchronization framework for decentralized web applications between mistrusting parties. BeauForT combines the efficient operation and lightweight setup of a peer-to-peer data synchronization framework with the resilience and fault tolerance of a BFT consensus protocol. The novel BFT protocol, optimized for unstable network conditions with higher latencies, does not require that all replicas are directly connected to each other. It also does not rely on a leader, removing the need for a costly leader election procedure when this leader is malicious or loses its network connection temporarily. The latter scenario is common in our target environment. Each browser replica only maintains the current authenticated state and does not need to keep track of an operation log or transaction history, keeping the storage footprint small. To further reduce the storage and bandwidth requirements, we use an aggregate signature scheme called BLS [BLS01]. This also reduces the computational requirements, as you can verify multiple signatures at once. The authenticated state and consensus votes are replicated over multiple hops using a gossip protocol.

BeauForT combines the following contributions in a browser-based middleware:

1. Lightweight, leaderless, client-centric Byzantine fault tolerant consensus, over values constrained by an application-level callback.
2. Resilient and robust, state-based synchronization of both the data and the votes for the consensus protocol using state-based CRDTs and Merkle-trees.
3. Delayed verification and aggregation of signatures using the BLS scheme.

Our evaluation, using our application use case of a shared loyalty program between small-scale merchants, shows that BeauForT is a practical solution for these kinds of community-driven use cases. BeauForT achieves transaction finality in the order of seconds, even in networks with 100 browser clients. Compared to other state-of-art BFT consensus protocols, our protocol is more robust against unstable network conditions.

This chapter is structured as follows. Section 3.2 presents a motivational use case. Section 3.3 presents BeauForT's lightweight BFT consensus protocol and the state-based replication strategy. The detailed web-based middleware architecture of BeauForT is elaborated in Section 3.4. Our evaluation in Section 3.5 focuses on many aspects of performance in both the optimistic scenario as well as more realistic and even Byzantine scenarios. Section 3.6 elaborates on important related work. We conclude in Section 3.7.

3.2 Motivation

We first describe an initial use case that would benefit from the lightweight, robust consensus offered by BeauForT. The use case involves business transactions happening in real life and needs interactive performance and robustness, rather than high throughput or scalability. We then formulate our vision on decentralized web applications.

Loyalty programs. Integrated loyalty programs can be more effective than traditional loyalty programs that are limited to a single company [FT16]. Think about airlines that award *miles* which can be redeemed with several partners. Such collaborations usually introduce an extra trusted intermediary and add more layers of management and operational logistics. This trusted party can charge high transaction costs to be part of the integrated network. For small merchants on a farmer's market or in a local shopping street, this operational overhead is too much of a burden. A decentralized peer-to-peer network can enable fast and secure creation, redemption, and exchange of loyalty points across different merchants. For the purpose of this chapter, we assume a fixed membership, i.e., an initial group of merchants come together and decide to run this protocol. They have a list of all public keys of all the merchants. If later a

new merchant wants to be part of the network, or an existing member wants to leave, they will have to stop the protocol (i.e., in the evening after the stores are closed) and update the list of public keys.

Vision. We envision that communities will be able to use BeauForT as a platform to explore new applications and use cases that were previously not feasible. While our initial proof-of-concept implementation is targeting the browser, the techniques explained in this chapter can be easily ported toward native mobile and lightweight desktop applications. BeauForT does not need any complex infrastructure, and it currently provides a simple JavaScript-based API, which allows many developers to start developing decentralized applications. Those decentralized applications can be made open source, which allows many people to verify and vouch for them. Local communities who want to set up a decentralized application between the local participants can use such an application and do not need to concern themselves with a complex infrastructure setup to run the application. Nor do they need to rely on a general-purpose third-party network, such as a public blockchain.

3.3 BeauForT protocol

This section explains the state-based consensus protocol used in BeauForT. First, it describes the adversary model and its properties. Then it explains the protocol specification. At last, we prove the safety and liveness properties of the protocol.

3.3.1 System model

We assume a partially synchronous network [DLS88]. Messages can be delayed, dropped, or delivered out of order. An adversary might corrupt up to f replicas of the $n \geq 3f+1$ total replicas. They can deviate from the protocol in any arbitrary way. Such replicas are called Byzantine, while the replicas that are strictly following the protocol are called honest. At least $2f+1$ honest replicas should be able to make a connection to each other. In practice, they are transitively connected to each other, but only directly connected to a few replicas. The topology can change over time. If no progress is being made on a new proposal, replicas will close some existing connections and connect to a few different replicas. Each replica will gossip its neighbors to every replica it connects to. We assume attackers are computationally bounded and it is infeasible to forge the used asymmetric signatures or find collisions for the used cryptographic hash functions.

We address in this chapter a replicated key-value store for which replicas coordinate agreement using a Byzantine Fault Tolerant consensus protocol, such that the following classical properties hold [CGR11]:

- *Termination*: Every correct replica eventually decides some value.
- *Validity*: If all replicas are correct and propose the same value v, then no correct replica decides a value different from v; furthermore, if all replicas are correct and some replica decides v, then v was proposed by some replica.
- *Agreement*: No two correct replicas decide differently.
- *Integrity*: No correct replica decides twice.

All writes to a key-value pair are atomic, meaning that only a single state transition can happen at any time. Extra application-level conditions can be applied to limit who can write to it, and which values are acceptable given the previous value. BeauForT does not use a leader to coordinate the protocol, removing a common single-point-of-failure compared to many existing BFT protocols. In such leader-based protocols, the failure of a leader leads to a long delay before consensus can be reached. This is even the case for rotating leader protocols such as HotStuff [Dan+22]. The set of replicas is fixed, and changes to the replica set have to be made outside the protocol, e.g., by halting the protocol, updating the set of replicas on all replicas, and starting the protocol again. Consensus is reached for each key-value pair separately, which means that each key has its own instance of the BeauForT protocol.

3.3.2 Protocol specification

The specification of the protocol is shown in Algorithm 4 and 5. The state consists of three parts. The first part is the current value (line 2) and a quorum certificate (line 3). The quorum certificate contains signatures of a supermajority of $n-f$ replicas and proves the validity of the value. The second part is a map, which maps rounds to a collection of votes for the next value (line 5). In each round, there can be multiple proposed values. The third part consists of a new proposed value (line 6) and a partial quorum certificate for that value (line 7). Consensus is reached in two steps, first, a supermajority needs to be reached in the last round of the *votes*, then a supermajority needs to be reached for the next quorum certificate. The first step will establish a resilient quorum, while the second step will guarantee that sufficiently many replicas know that such a quorum has been achieved. The flow of the protocol is shown in Figure 3.1.

Proposing new values. To write a new value, a replica has to propose a new value to the other replicas. This process is the PREPARE phase in Algorithm 4. The proposing replica adds the new value and its vote to round 0 of *votes* (line 10). As the protocol is leaderless, any replica can be a proposing replica and multiple replicas can propose a new value simultaneously. Replicas are only allowed to vote once in each round for each view, so if the replica already voted for another value, it will have to wait until consensus is reached for the current view and

Algorithm 4 Basic protocol for replica *id*.

```
 1: n, f          ▷ Total number of replicas, Maximum number of Byzantine replicas
 2: value ← ⊥                                          ▷ Current accepted value
 3: qc ← ⊥                                      ▷ Quorum certificate for value
 4: for v ← 1,2,3,... do                                                ▷ view
 5:     votes ← ∅                                   ▷ round ↦ votesInRound
 6:     value′ ← ⊥                                              ▷ Next value
 7:     qc′ ← ∅                                     ▷ Next quorum certificate
    ▷ PREPARE phase
 8:     as a proposing replica:
 9:         wait for value val from client
10:         votes[0] ← {VOTE(v,0,val,PRE-COMMIT)}
11:     as a non-proposing replica:
12:         wait for any value in votes
13:     for r ← 0,1,2,3,... do                                         ▷ round
    ▷ PRE-COMMIT phase
14:         if ¬HASVOTED(votes[r]) then
15:             val ← WINNINGVALUE(votes[0])
16:             vote ← VOTE(v,r,val,PRE-COMMIT)
17:             votes[r] ← votes[r] ∪ {vote}
18:         wait for (n − f) votes in votes[r]
19:         val ← WINNINGVALUE(votes[r])
20:         valVotes ← VOTESFORVALUE(votes[r],val)
21:         if LEN(valVotes) ≥ (n − f) then
22:             vote ← VOTE(v,r,val,COMMIT)
23:             value′ ← val
24:             qc′ ← qc′ ∪ {vote}
25:         else
26:             val ← WINNINGVALUE(votes[0])
27:             vote ← VOTE(v,r + 1,val,PRE-COMMIT)
28:             votes[r + 1] ← {vote} ∪ votes[r + 1]
29:             continue
    ▷ COMMIT phase
30:         wait for (n − f) votes in qc′:
31:             if LEN(votes) − 1 > r then
32:                 value′ ← ⊥
33:                 qc′ ← ∅
34:                 continue
35:         value ← value′
36:         qc ← qc′
```

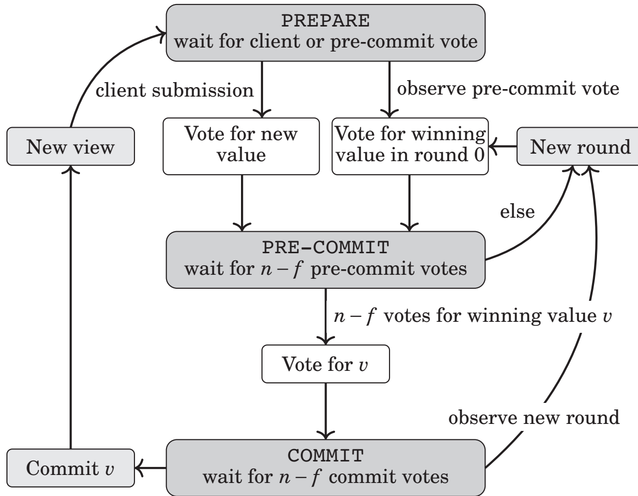


Figure 3.1: State transition diagram of the BeauForT consensus protocol.

Algorithm 5 Basic protocol for replica id. (continued)

37: **function** WINNINGVALUE($votesInRound$)
38: **return** $argmax_{value}$ LEN($\{v \in votesInRound : v.value = value\}$)
39: **function** VOTESFORVALUE($votesInRound$, $value$)
40: **return** $\{v \in votesInRound : v.val = value\}$
41: **function** HASVOTED($votesInRound$)
42: **return** $\exists\, v \in votesInRound : v.id = id$
43: **function** VOTE($view$, $round$, val, $type$)
44: **return** ($val, id,$ SIGN($view, round, val, type, id$))

propose the new value in the next view. The non-proposing replicas will receive the new proposal(s) via the gossip protocol, and also enter into the next phase.

Consensus. Consensus about which value will be accepted in a view is reached in two phases, called PRE-COMMIT and COMMIT in Algorithm 4. Honest replicas will always vote for the value with the most votes in round 0 (line 14-17). If multiple values have the same number of votes, the lexicographic order of the hash of those values is taken as a tiebreaker. If a round has reached a supermajority of votes for a single value, then no new round can be started anymore, and the replicas will start creating a new quorum certificate (line 21-24). If a supermajority of the replicas have voted in a round, but not a single value reaches a supermajority, a new round is started (line 25-29) and all replicas can vote again in this new round. The replicas are only allowed to vote on the current winner in round 0

according to their local state (line 14-17). Because each replica might have a different state on the current set of votes in round 0, there can still be multiple values in the next round without any supermajority for a single value.

Another factor is Byzantine nodes trying to halt the system by voting not according to the rules. However, the set of possible values to vote on gets smaller with every round, and eventually, the view of all the honest replicas on the votes in round 0 will become the same, and the winning value can be chosen unanimously. The reason for this is that a replica does not simply send a message with his vote to the others, but instead gossips the entire state. This includes all votes for the previous rounds. This means that when two replicas disagree with each other in a certain round, once they communicate with each other, they will learn each other's state. In the next round, they will both vote for the same value (as their local state of $votes[0]$ will be the same). Malicious replicas can try to shift the balance to violate liveness, but with each round, they have less possibility to do so. Because when they gossip $votes[i]$ they also gossip the previous rounds which should show why they voted on a certain value. If a replica detects that another replica is Byzantine, it will exclude this Byzantine replica permanently, and its votes do not count anymore.

Once a replica enters the COMMIT phase, it will wait for $n-f$ replicas to also confirm that the proposed value can be committed (line 30). A malicious replica can trick an honest replica to enter this phase without the support of enough honest replicas. For this reason, during this waiting period, if the replica observes that other replicas started a new round, it will realize its mistake and remove the partial commit certificate and go back to the PRE-COMMIT phase (line 31-34). The malicious replica can also be detected, as there will be two signatures of him signing two votes for two different values in the same round.

If $n-f$ replicas agree and add their vote to the quorum certificate for the next value, the value will be accepted and the quorum certificate will be stored to later convince other replicas that the value is indeed correct (line 36).

Correctness. The integrity and validity properties are trivially satisfied. We can now reformulate the agreement and termination properties more precisely as a safety and liveness property. We prove these properties in Section 3.3.3.

Theorem 1. *Let $\Re$ be a cluster of n replicas with f Byzantine replicas and $n \geq 3f+1$. BeauForT's correctness is defined by the following two properties:*

- *Safety: If replicas $R_1, R_2 \in \Re$ are able to construct quorum certificates qc_1 for value $value_1$ and qc_2 for value $value_2$ at view v, then $value_1 = value_2$.*
- *Liveness: If an honest replica $R \in \Re$ proposes a new value $value_1$ at view v, eventually a replica will be able to construct a quorum certificate qc for some value at view v.*

State-based replication protocol. During all phases in the algorithm, the state is continuously broadcasted to the other replicas. The full state, including all votes in the consensus protocol, is replicated by using a state-based gossip protocol. A major feature of gossip-based communication is its reliability [Cas+21b]. Each time a new state is received, the local state is merged with the remote state. This protocol synchronizes data peer-to-peer using state-based Conflict-free Replicated Data Types (CRDTs) [Sha+11a] combined with a Merkle-tree [Mer88] to efficiently replicate the updated state, similar to OWebSync [JLJ21]. All key-value pairs are put inside a Merkle-tree. Each key-value pair is a separate instance of the consensus protocol in Algorithm 4. The Merkle tree is used to efficiently replicate the state between any two replicas. A replica will first send its own root hash to another replica. If those hashes are equal, that replica knows that both replicas have the same state, and the gossip protocol ends. If however the hashes are not equal, that replica will descend in the Merkle-tree and send all hashes in the next level of the tree to the first replica. This process continues until a specific key-value pair is reached, and then the full state of the consensus protocol in Algorithm 4 is sent ($value$, qc, v, $votes$, $value'$, and qc'). The state of the protocol can be represented as a CRDT: $votes$ and qc' are Grow-only Sets [Sha+11a], and a state associated with a higher $view$ number overwrites any older state, much similar to a LWWRegister [Sha+11a]. There are two extra constraints imposed on the CRDTs due to the Byzantine nature. First, signatures have to be correct, no replica may accept any invalid signature, if a replica does send a wrong signature, it can be considered Byzantine, and the other replicas will drop their connection to it. Secondly, not all states are valid. For example, $votes$ keeps track of the different rounds, but no new round can be started unless $n-f$ votes in the previous round are present, and no consensus has been reached yet. When a replica receives an invalid state, it will be ignored, and the other replica can be considered Byzantine. If those $n-f$ votes are all for the same next value, then no new round is started. These constraints, signatures, and invalid states, are verified before the CRDTs are merged.

By using a state-based approach, rather than the operation-based approach of operation-based CRDTs [Sha+11a], blockchains [Nak08], or traditional BFT protocols, we only need to store the current state together with some metadata. There is no need to store the full log of all operations to later convince replicas that were temporarily offline of the new state. Replicas also do not need to keep track of the state of other replicas, or which messages are already received by which replica. If a new value and quorum certificate with a higher view are received, then the protocol will accept the new state, and the protocol will reset back to line 3 of Algorithm 4 with that newer view. Note that we do not explicitly show the gossiping in Algorithm 4 to keep the algorithm compact. During the whole protocol, the state is continuously gossiped between the replicas. This way, $votes$ or qc' will eventually contain enough votes to continue in the

protocol specification. The state-based replication also helps with the consensus protocol. Instead of only sending proposals and decisions to other replicas, the full state of $votes$ and qc' is sent. This approach allows replicas to hold each other accountable when they cast their vote. Their $votes$ should support why they voted for a specific value, otherwise, they will be considered Byzantine and excluded from the network.

Examples. An example of this replication process is shown in Figure 3.2. There are four non-Byzantine replicas with an empty set of $votes$ and empty qc' at t_0. The scenario starts at t_1 with replica A proposing a new value v (line 10 of Algorithm 4). The state is replicated to the other replicas randomly. In the example, the state is gossiped to replicas B and C at t_2, and those replicas merge the received state with their local state. Since B and C did not yet vote in this view and round, they will cast their vote for the current winning value (line 14-17). This process continues at t_3 when replica B sends its state to replicas A and C. At t_3, replica C observes that a supermajority of the replicas support value v, and it starts working on a new quorum certificate to determine if at least a supermajority of the replicas also knows about this (line 21-24).

A second example of this replication process is shown in Figure 3.3. Imagine now the same four non-Byzantine replicas. Replica A again proposes a new value v_1, but concurrently replica B proposes another value v_2. If we use the same gossiping path as in Figure 3.2, then at t_2 replicas B and C receive the vote from replica A. Replica B will not vote anymore, because it already voted for its own value v_2. At t_3, replica B gossips its state to replica A and C. Replica A will now have one vote v_1 (his own) and one vote for v_2 (from B). Replica C however will now have two votes for v_1 (from A and C) and one vote for v_2 (from B). Since replica C now has $n-f=3$ votes in round 0, but there are only two votes for the winning value, it will start a new round and vote for the winning value in $votes[0]$, which is v_1. B will now also vote for v_1 in $votes[1]$ and a commit certificate can be created after round 1.

A third and last example of this replication process is shown in Figure 3.4. Imagine that replica D also receives the votes from A and B between t_1 and t_2. If the vote from B comes in first, then D will also vote for v_2 and start a new round with a vote for v_2 (as this is the winning value in its opinion). So after t_3 we now have replica C in round 1 with v_1 and replica D in round 1 with v_2. The other replicas A and B are still in round 0 until they receive more votes. If, for example, replica C now gossips its state to D, all votes in round 0 will become known, and all replicas will deterministically vote for the same value v_2 in the next round (if we assume the hash of v_2 is larger then the hash of v_1).

Since replicas will vote for the first value they observe, a well-placed replica that can send its request to enough other replicas first can prevent requests from other

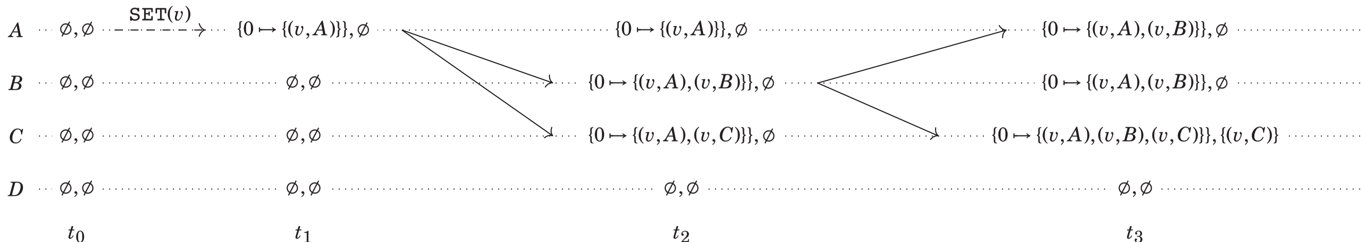


Figure 3.2: Example 1 of the state-based synchronization with 4 replicas A, B, C, D. Only the current *votes* and qc' are shown. Arrows represent a state transfer.

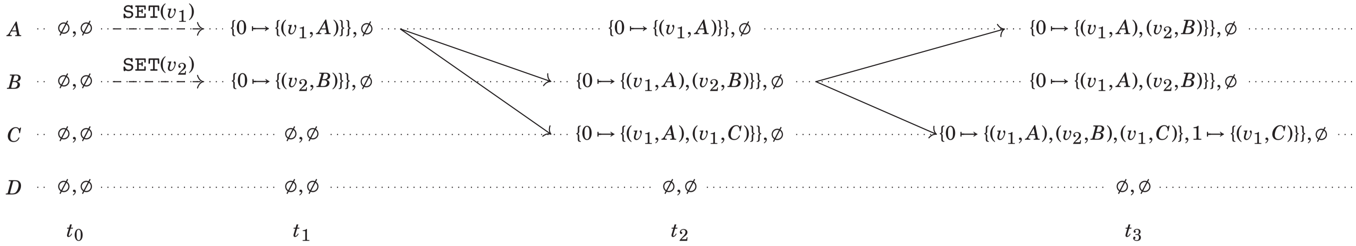


Figure 3.3: Example 2 of the state-based synchronization with 4 replicas A, B, C, D. Only the current *votes* and qc' are shown. Arrows represent a state transfer.

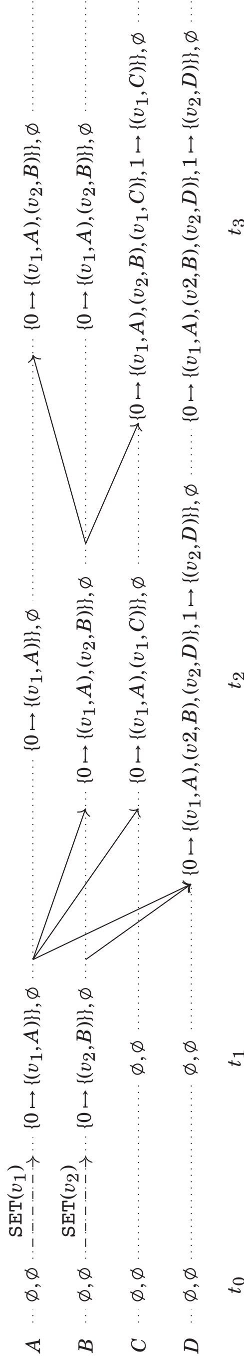


Figure 3.4: Example 3 of the state-based synchronization with 4 replicas A, B, C, D. Only the current *votes* and qc' are shown. Arrows represent a state transfer.

replicas from ever being accepted. This does satisfy the liveness constraint that was specified formally in Section 3.3.2: in which we specify that when new values are proposed, some value should be eventually accepted. The protocol does not provide deterministic fairness, i.e., no guarantees are made for a single proposed value. In practice, we have two arguments in favor of our model. First, when a replica notices that no progress is being made on a proposal, it will close some connections randomly and open new connections to other replicas. This makes it much harder for such a well-placed replica to be well-placed for a long time. Second, our use case of loyalty points across small-scale merchants prevents any problems because only the client (customer) can sign a message to spend loyalty points at a certain merchant. In this case, only a single proposal will ever be present if the client is honest, and it will always be eventually accepted by the network.

Delaying signature verification. For brevity, we did not show the actual verification of signatures in Algorithm 4. However, in the basic protocol, each time a new signature is received, it needs to be verified. This can become quite costly, and therefore BeauForT will use a fast path and delay the verification of any incoming signatures. BeauForT will just accept and replicate them, until a decision needs to be made, such as starting a new round or starting to create a new proposed quorum certificate. Only then, all signatures will be verified in one batch. If all signatures are valid, the protocol can continue as normal. If there are invalid signatures, then those will be removed and BeauForT will continue to collect more signatures and verify them on arrival. This hybrid approach enables very fast consensus when all replicas are honest, while gracefully degrading to a slower, more costly protocol that can detect which replicas are actively acting Byzantine.

3.3.3 Safety and liveness proofs

Safety

Theorem 2 (Safety)**.** *Let $\Re$ be a cluster of n replicas with f Byzantine nodes and with $n > 3f$. If replicas $R_1, R_2 \in \Re$ are able to construct quorum certificates qc_1 for value $value_1$ and qc_2 for value $value_2$ at view v, then $value_1 = value_2$.*

We will first prove this for the simplified case when both quorum certificates belong to the same round (Lemma 1), and we will then prove that once a quorum certificate can be constructed, no more rounds can be started (Lemma 2).

Lemma 1. *If replicas $R_1, R_2 \in \Re$ are able to construct quorum certificates qc_1 and qc_2 for value $value_1$ and $value_2$ respectively with $qc_{1\,view} = qc_{2\,view}$ and $qc_{1\,round} = qc_{2\,round}$, then $value_1 = value_2$.*

Proof. Assume two different replicas R_1 and R_2 have constructed a quorum certificate qc_1 and qc_2 for value $value_1$ and $value_2$ respectively with $qc_{1\ view} = qc_{2\ view}$ and $qc_{1\ round} = qc_{2\ round}$. They are constructed in the same round, so of the n possible votes, at least $n-f$ replicas have voted on $value_1$, and at least $n-f$ replicas have voted on $value_2$. Honest replicas will never vote twice in the same view and round. Therefore, at least $n-2f$ honest replicas have voted on $value_1$ and $n-2f$ *different* honest replicas have voted on $value_2$. In total, we have $(n-2f)+(n-2f)+f \equiv 2n-3f$ replicas that have voted. We defined $n \geq 3f+1$ before, which gives $f \leq \frac{1}{3}n - \frac{1}{3}$. If we replace f in the first equation, this gives $2n-3f \geq 2n-3(\frac{1}{3}n-\frac{1}{3}) = n+1$. This is a contradiction, there need to be at least $n+1$ replicas to construct two such certificates for different values, however, we only have n replicas. So the two values $value_1$ and $value_2$ have to be equal. □

Lemma 2. *If replicas $R_1, R_2 \in \Re$ are able to construct quorum certificates qc_1 and qc_2 for value $value_1$ and $value_2$ respectively with $qc_{1\ view} = qc_{2\ view}$, then $qc_{1\ round} = qc_{2\ round}$.*

Proof. Assume two different replicas R_1 and R_2 have constructed a quorum certificate qc_1 and qc_2 for value $value_1$ and $value_2$ respectively with $qc_{1\ view} = qc_{2\ view}$ and $qc_{1\ round} < qc_{2\ round}$. Since qc_1 is accepted, at least $n-f$ replicas vote on the proposed quorum certificate, and at least $n-f$ replicas voted on $value_1$ in round $qc_{1\ round}$. The fact that $n-f$ replicas voted on the proposed quorum certificate means that at least $n-2f$ honest replicas observed $n-f$ votes for $value_1$. Of those votes, at least $n-2f$ are coming from honest replicas. The only way to now construct a quorum certificate qc_2 for $value_2$ is to start a new round. To start a new round, a replica needs to not have voted for the proposed quorum certificate qc_1, and observe a different winning value $value_2$. Yet, at least $n-2f$ honest replicas observed that at least $n-2f$ honest replicas think that $value_1$ is the winning value. This leaves only $2f$ replicas who can prefer another value $value_2$. By definition, we have $n \geq 3f+1$. This means that in the worst case, $f+1$ honest replicas observe $f+1$ honest replicas thinking $value_1$ is the winning value, together with f Byzantine replicas. Value $value_2$ has only $2f$ supporting replicas, which is not enough to start a proposed quorum certificate. So, at least one replica currently supporting $value_1$ needs to switch votes in a future round. However, once a replica has voted for a proposed quorum certificate, it will not change its opinion unless it is convinced that a new valid round is started. So once $n-2f$ honest replicas are locked on a value, by voting on a proposed quorum certificate, it is impossible to start a valid new round. □

Liveness. When a new value is proposed, eventually the protocol will end and a valid quorum certificate is created for a new value. This value is not necessarily the first proposed value, and it is not even guaranteed that a specific value ever

gets committed as long as other values continue to be proposed. Safety is always chosen over liveness. When there are not enough honest replicas online to reach a supermajority, no consensus can be reached and the protocol will simply block and wait for more votes. However, all those replicas do not need to be online at the same time, since the state is replicated to all available replicas over time, and votes can be verified by all replicas in the end.

Theorem 3 (Liveness)**.** *Let $\Re$ be a cluster of n replicas with f Byzantine nodes and with $n > 3f$. If an honest replica $R \in \Re$ proposes a new value at view v, eventually a replica will be able to construct a quorum certificate qc for some value at view v.*

We will first proof two simplified lemmas (Lemma 3 and Lemma 4) which we will use in the proof for Theorem 3 which is equal to Lemma 5.

Lemma 3. *If only a single replica $R \in \Re$ proposes a new value $value_1$, eventually a replica will be able to construct a valid quorum certificate qc.*

Proof. As there is only a single proposed value, all honest replicas who observe this will cast their vote for that value. Eventually, an honest replica will observe $n-f$ votes for $value_1$ and that replica can start creating a new proposed quorum certificate qc'. Eventually, $n-f$ votes will be cast to this proposed quorum certificate qc', and a valid quorum certificate qc is constructed, and $value$ is committed. □

Lemma 4. *If x replicas $R_{1..x} \in \Re$ propose values $value_{1..x}$, and no Byzantine replicas vote twice in the same round, eventually a replica will be able to construct a valid quorum certificate qc.*

Proof. Either a single value reaches a quorum, in which case the previous lemma holds. Or a split vote occurs and a new round will be started after $n-f$ votes are observed. All replicas will base their vote for this new round on the winning value that they observed from round 0. At least $n-f$ votes are known, and only f votes are still unknown. “Known” means known to the one replica that is making some decision and going ahead in the protocol. But to make progress, at least $n-f$ replicas need to know about $n-f$ votes. These votes that are known, are not necessarily the same for all $n-f$ replicas, but eventually, all honest replicas will know about the exact same votes. As long as not all votes are known to all voting replicas, the winning value might change. In each new round, either all unknown votes stay unknown, or one becomes known. In the former case, then the currently known votes will all be the same, and a proposed quorum certificate can be started. In the latter case, one extra vote is known, which might again result in the system ending up in a split vote, and a new round will be started.

However, this last case can only happen at most f times. After $f+1$ rounds, all replicas will have voted in round 0, every replica will observe the same winning value, and a quorum certificate can be created. □

Lemma 5. *If x replicas $R_{1..x} \in \Re$ propose values $value_{1..x}$, eventually a replica will be able to construct a valid quorum certificate qc.*

Proof. If no Byzantine replicas vote twice in the same round, or only a single value is proposed, the previous two lemmas hold. If a split vote occurs, a new round will be started after $n-f$ votes are observed. f of those votes might belong to Byzantine replicas who can vote for multiple values. As a new round is only started after $n-f$ votes, a least $n-2f$ honest votes are observed. No Byzantine replica can send conflicting votes to any of those $n-2f$ honest replicas, as otherwise those replicas will detect this conflicting vote and exclude the Byzantine replica. With exclusion we only mean that their votes are not counted anymore on each honest replica that observed that a Byzantine replica voted twice. So it is even possible that some replicas exclude the Byzantine replica, while other replicas are still trusting it. However, as all votes will be gossiped, eventually all honest replicas will know about the Byzantine replica. Safety will not be violated because n (in the formula $n-f$) stays the same. But to reach this threshold, the votes from Byzantine replicas are ignored. If another Byzantine replica sends conflicting votes, then after at most f times, all Byzantine replicas are excluded and the previous lemma holds. Moreover, no Byzantine replica can continue to vote on values that are not the winning value. Each replica is only allowed to vote on the winning value or any other value that has at least support from $f+1$ replicas in the previous round. All honest replicas converge to a single value, even with Byzantine replicas supporting other values. Because the protocol only looks to round 0 to determine the winning value. In the rounds after that, the f Byzantine replicas can support a different value, but if they do, they will be excluded as $f < f+1$. This means that after at most $2f+1$ rounds, a proposed quorum certificate can be started, which will be committed. □

3.4 Architecture and implementation

This section describes the client-centric architecture, deployment, and implementation of BeauForT. This middleware architecture is key to supporting the BFT consensus and synchronization protocol described in the previous section. BeauForT is fully web-based and written in JavaScript and can execute in any recent browser without any plugins. This section first describes the overall architecture. Then it explains our use of aggregate signatures using BLS to reduce the size of the data.

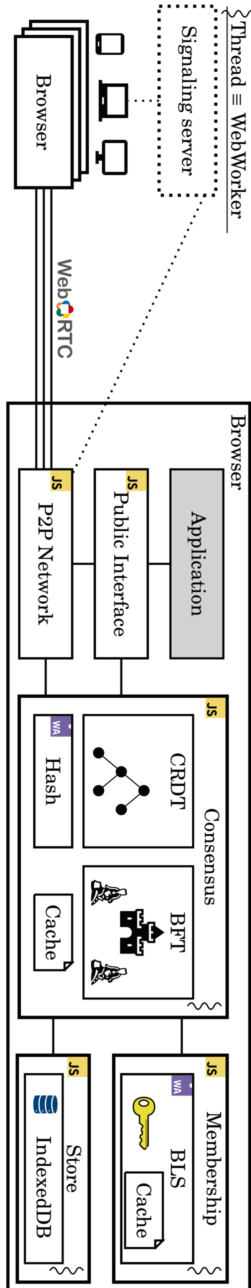


Figure 3.5: Browser-based architecture of BeauForT.

3.4.1 Overall architecture

The BeauForT middleware architecture consists of five main components (Figure 3.5): (i) a *public interface* that offers an API for developers, (ii) a *peer-to-peer network* component to communicate directly with other browsers, (iii) a *consensus* component to handle the consensus protocol described in the previous section, (iv) a *membership* component to handle all cryptographic operations, and (v) a *store* component to save all state to persistent storage. The last three components run on a different browser thread by using Web Workers.

(i) Public interface. This component provides an API to application developers to use this middleware. It provides four functions to modify the application state. `GET(key)` returns the current value at the given key. `SET(key, value)` submits a proposal to update the value at the given key. `DELETE(key)` submits a proposal to delete the value at the given key. A tombstone is kept for correct replication. `LISTEN(key, callback)` supports reactive programming by calling the callback with the new value each time a new value for the key is confirmed by the network.

Apart from those functions, the middleware also provides a constructor function to initialize the middleware by passing the following four configuration parameters: the list of all members of the network together with their public key, the private key of the replica, the URL to the signaling server to set up the peer-to-peer connections, and a callback to verify state changes and restrict valid values. This callback is called before voting for a new proposed value, with both the old and new values as arguments. It should return a `boolean` whether to allow this change or not. This callback enables the implementation of basic access control policies on the values. One example is to embed the public key of the owner into the value and require each new value to be signed by the owner. This value can only be changed by the owner and supports passing ownership by changing the embedded public key.

(ii) Peer-to-peer network. The *P2P Network* component manages the peer-to-peer network and is responsible for the replication of the state-based CRDTs. Many browser-based replicas are connected to each other using WebRTC (Web Real-Time Communications). WebRTC enables a browser to communicate peer-to-peer. However, to set up those peer-to-peer connections, WebRTC needs a signaling server to exchange several control messages. Once the connection is set up, all communication can happen peer-to-peer, without a central server. Another WebRTC peer-connection can also be used as a signaling layer, so once a replica is connected to another one, it can also connect to all of its peers, without the need for a central signaling server. In our adversary model, this server is assumed to be trusted. If this signaling server would be malicious, the safety of the system is not endangered as no actual data is sent to this central server. However, some peers

might not be able to join the network and the required supermajority might not be reached, which violates liveness. The use of multiple independent signaling servers can lower the risk of this happening. At startup, every replica will connect to some other replicas randomly. In our implementation, a connection will be made to at least seven other replicas. This number is arbitrary but performed best in our experimental evaluation. A higher number will increase resource usage, and decrease the potential to batch multiple updated states together. A lower number will increase the number of hops, and therefore increase the latency. To defend against an eclipse attack, where few Byzantine neighbors try to surround an honest replica to break liveness, a replica can periodically create new connections to other peers and drop older connections when no updates are being gossiped to them, or when proposals are not being voted on. This is similar to how Bitcoin works [Nak08].

(iii) Consensus. The *Consensus* component handles the consensus protocol described in Section 3.3. It maintains a Merkle-tree of all key-value pairs and uses the state-based CRDT framework OWebSync [JLJ21] to replicate the local state to other replicas using the *P2P Network* component. The Merkle-tree is constructed using the Blake3[1] cryptographic hash function. For performance reasons, the hash function is implemented in Rust and compiled into WebAssembly.

(iv) Membership. The *Membership* component contains all cryptographic material and is responsible for all cryptographic operations such as signing and verification of signatures. We use an aggregate signature scheme called BLS [BLS01], more specifically BLS12-381. Section 3.4.2 provides more details about the BLS implementation. It is implemented in C and compiled into WebAssembly.

(v) Store. At last, the *Store* component saves all state to the IndexedDB database. IndexedDB is a key-value datastore built inside the browser. Each value and the Merkle-tree are serialized to bytes and stored there under the respective key. This enables users to close the browser and continue afterward without losing the current state.

3.4.2 Aggregate signatures using BLS

The consensus protocol in Section 3.3 is resource-intensive with respect to aggregation and verification of digital signatures. Signatures must be continuously collected and verified. This means, in every intermediate state of a transaction, each party needs to keep track of all incoming signatures and verify them to prevent malicious scenarios. Persistence, management, and transmission of these signatures are costly, especially in a browser-based setting. Therefore,

[1]As a research prototype, we opted for a newer hashing algorithm with fewer rounds [Aum19]. For production usage, a standardized hashing algorithm such as SHA-256 or SHA3-256 is more suitable.

$\mathbb{G}_0$ and $\mathbb{G}_1$ are two multiplicitive cyclic groups of prime order q. $\mathsf{H}_0 : \{0,1\}^* \to \mathbb{G}_0$ and $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{Z}_q$ are hash functions viewed as random oracles.

1. *Parameters Generation:* $\mathsf{PGen}(\kappa)$ sets up a bilinear group $(q, \mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_t, e, g_0, g_1)$ as described by [BDN18]. e is an efficient non-degenerating bilinear map $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_t$. g_0 and g_1 are generators of the groups $\mathbb{G}_0$ and $\mathbb{G}_1$. It outputs $params \leftarrow (q, \mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_t, e, g_0, g_1)$.
2. *Key Generation:* $\mathsf{KGen}(params)$ is a probabilistic algorithm that takes as input the security $params$, generates $sk \xleftarrow{\$} \mathbb{Z}_q$, computes and sets $pk \leftarrow g_1^{sk}$, and outputs (sk, pk).
3. *Signing:* $\mathsf{Sign}(sk, m)$ is a deterministic algorithm that takes as input a secret key sk and a message m. It computes $t \leftarrow \mathsf{H}_1(pk)$, and outputs $\sigma \leftarrow \mathsf{H}_0(m)^{sk \cdot t} \in \mathbb{G}_0$.
4. *Key Aggregation:* $\mathsf{KAgg}(\{(pk_i, r_i)\}_{i=1}^n)$ is a deterministic algorithm that takes as input a set of public key pk and the multiplicity r pairs. It computes $t_i \leftarrow \mathsf{H}_1(pk_i)$, and outputs $apk \leftarrow \prod_{i=1}^n pk_i^{t_i \cdot r_i}$.
5. *(Multi-)Signature Aggregation:* $\mathsf{Agg}(\sigma_1, ..., \sigma_n)$ is a deterministic algorithm that takes as input n signatures. It outputs $\sigma \leftarrow \prod_{i=1}^n \sigma_i$.
6. *Verification:* $\mathsf{Ver}(apk, m, \sigma)$ is a deterministic algorithm that takes as input aggregated public keys $apk \in \mathbb{G}_1$, and the related message m and signature $\sigma \in \mathbb{G}_0$. It outputs $e(g_1, \sigma) \stackrel{?}{=} e(apk, \mathsf{H}_0(m))$.

Figure 3.6: Formal specification of the optimized BLS signature scheme.

our protocol requires short and compact signatures to reduce storage and network footprint. Boneh–Lynn–Shacham (BLS) [BLS01] presented a signature scheme based on bilinear pairing on elliptic curves. The size of a signature produced by BLS is compact since a signature is an element of an elliptic curve group. The aggregation algorithm [Bon+03] outputs a single aggregate signature as short and compact as the individual signatures, unlike other approaches that rely on ECDSA, DSA, or Schnorr. The subgroups have a prime order of 255 bits and BLS12-381 offers a security level close to 128 bits. Other state-of-the-art BFT systems such as SBFT [Gue+19] and HotStuff [Yin+19] also use aggregate or threshold signatures. However, they use it in a different way. They let the leader compute the aggregate signature. BeauForT uses a different approach, once a proposed quorum certificate has reached a supermajority of the votes, any replica can aggregate these into one single aggregated BLS signature. BeauForT makes a trade-off between performance, bandwidth, and storage space. Verifying a single signature is expensive, however, aggregation is cheap in performance. For this reason, BeauForT will delay the verification of the signatures until the latest possible moment (as explained in Section 3.3.2). Only then the individual signatures are aggregated and verified. If the verification fails, a binary search

can be conducted to find the invalid signatures and remove them. This leads to a higher bandwidth usage, compared to always aggregating two shares immediately. But allows for cheaper recovery when a Byzantine replica is sending invalid signatures. Once a signature is aggregated and verified, the individual shares are discarded, saving both bandwidth and storage space.

The standard scheme is vulnerable to rogue public key attacks. The state-of-the-art approach [BDN18] to mitigate such attacks is to compute $(t_1,...,t_n) \leftarrow \mathsf{H}_1(pk_1,...,pk_n)$ for each Agg invocation and compute $\sigma \leftarrow \prod_{i=1}^{n} \sigma_i^{t_i}$, where pk_i is the public key of replica i, H_1 is a hash function, and σ_i is a signature produced by replica i. Although the t_i values can be cached, the computation of σ would be costly. Moreover, Agg does not take as input the same set of public keys at different states of a transaction in our consensus protocol. Therefore, we distribute the computations by moving the calculations of the t_i and $\sigma_i^{t_i}$ values to the signing parties, and as a result, these computations are performed only once. Now, any replica can run Agg by only computing $\sigma_1...\sigma_n$. The security properties of BLS remain intact [BDN18], and we obtain more efficient aggregations at scale. We provide the mathematical background and formal specification of the optimized BLS scheme in Figure 3.6.

3.5 Evaluation

We validated the BeauForT middleware with the loyalty points use case presented in Section 3.2. The first subsection presents this validation. Next, we present three different benchmarks with different scales. The first benchmark shows the performance results in the optimistic scenario without network failures or Byzantine failures. The second benchmark evaluates the performance in a more realistic scenario with some network failures. The last benchmark evaluates the performance in the presence of a Byzantine replica.

3.5.1 Validation in the loyalty points use case

The deployment of the loyalty points use case consists of three services: a web application running in a browser for each merchant, a web server to serve the static web application files, and a signaling server to set up WebRTC peer-to-peer connections between the browsers. The web server is optional. Every merchant can also store those application files themselves and load them from their local file system. The signaling server is a trusted component. However, if trust is not present, you can set up multiple signaling servers to reduce potential misbehavior. No actual data is sent to the signaling server. It is only used to discover other peers on the network. To have a baseline, we compare BeauForT to two other existing state-of-the-art systems for BFT consensus: BFT-

SMART [BSA14; SBV18] and Tendermint [BKM18; Cas+21a]. BFT-SMART is a more traditional BFT protocol, similar to PBFT, where all replicas are connected to each other, and one leader drives the protocol. If that leader fails, a new one will have to be elected before any progress can be made. Tendermint uses gossip for communication between the replicas. There is still a leader, however, that leader changes frequently.

3.5.2 Test setup

To test the performance of BeauForT, we implemented the use case and deployed it on the Azure public cloud. We used 21 VMs (Azure F8s v2 with 8 vCPUs and 16 GB of RAM) with one VM acting as a central server running the web server and signaling server. The other VMs are running Chrome browsers inside a Docker container. Each of those VMs holds one to five browser instances for different scales of the benchmarks. To simulate a truly mobile environment, the network is delayed to an average latency of 60 milliseconds using the Linux `tc` tool, which simulates the latency of a 4G network. Every test is executed 10 times to ensure the results are reliable. In every run, the network configuration will be different, because replicas will connect to each other randomly to form the gossip network.

We are interested in the time it takes to confirm a transaction, experienced by the browser that submitted the transaction. Each transaction is a group of loyalty points being changed from owner. For example, a merchant gives some loyalty points to a customer or a customer redeems their loyalty points with a merchant. In the evaluation, the browser clients will do one transaction per second. This throughput is more than enough for the local community-scale use cases we envision. We compare the latency and network bandwidth with a different number of browsers. We show a boxplot of the latency results instead of only the average, as all users should experience fast confirmation times, and not only the average user.

3.5.3 Optimistic scenario

In the optimistic scenario, every replica is honest and no replicas fail, so the fast path can be used. One single aggregate signature is verified only before a decision, avoiding costly signature verifications after every message. As every replica is honest, this aggregate signature is correct and the new value can be accepted by all replicas.

Figure 3.7 shows the latency for the different technologies. For the use case of loyalty points, transactions must be confirmed fast, as people are waiting at checkout to receive or redeem loyalty points. BeauForT can confirm transactions within 4 seconds, even with a network of one hundred browsers. BFT-SMART

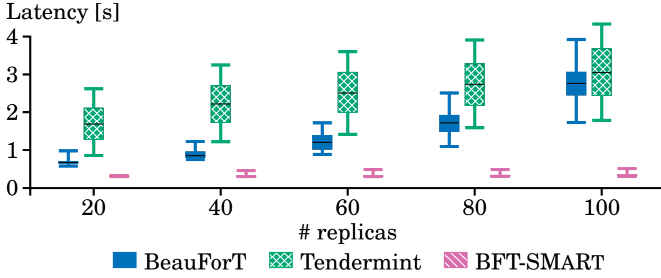


Figure 3.7: Latency in the optimistic scenario without failures.

can confirm transactions within half a second. This is because all replicas communicate directly with each other. However, having all replicas directly connected to each other is not realistic in a mobile peer-to-peer network. In contrast, BeauForT and Tendermint use gossip and need multiple hops before all replicas are reached. This also causes the increased latency. Furthermore, BFT-SMART uses HMAC to authenticate requests, which are an order of magnitude faster than the asymmetric signatures used in BeauForT and Tendermint. We can see a similar pattern in the bandwidth requirements shown in Figure 3.8. In the large-scale scenario with 100 browsers, BeauForT uses less than 3 Mbit/s, which is acceptable for a typical mobile network.

3.5.4 Realistic scenario

The same benchmark is now repeated with 25% of the replicas failing during the benchmark. A failure is simulated by dropping all network packets to and from that replica. Replicas fail one by one, with a 5-second delay between each failure. As all systems are Byzantine fault tolerant, they should be able to tolerate up to 33% of the replicas failing or acting Byzantine.

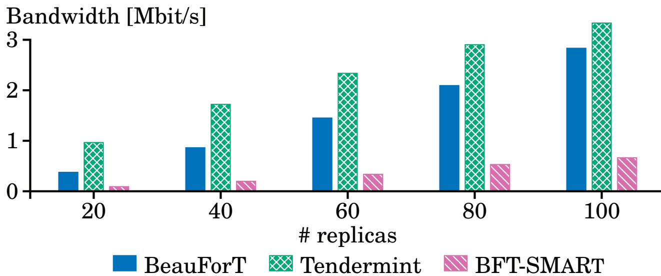


Figure 3.8: Network usage in the optimistic scenario without failures.

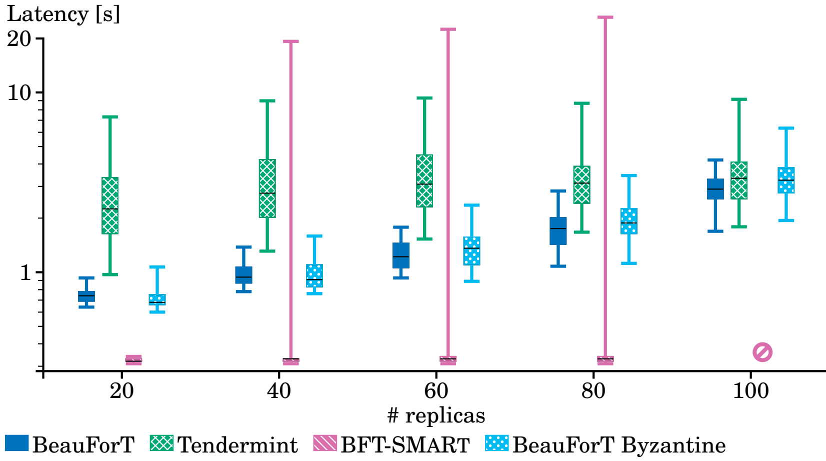


Figure 3.9: Latency in the realistic scenario with network failures. For BeauForT we included an extra scenario with a Byzantine replica trying to halt the network.

Figure 3.9 shows the latency in this scenario. BeauForT is not impacted much by the failing replicas and can still confirm transactions within 5 seconds. The impact on Tendermint is also small, but the tail latency is doubled to about 10 seconds. BFT-SMART however needs to use a costly leader election protocol when the current leader fails. This process takes some time, during which no transaction can be committed. Once a leader is chosen, the same fast performance can be achieved again. This behavior is clearly visible in Figure 3.9. The median latency of BFT-SMART is not affected by the failures. However, the tail latency increases to 27 seconds for the scenario with 80 replicas. It cannot handle the case with 100 replicas. BFT-SMART is unable to handle large network sizes when the latency between the nodes is higher than usual, e.g., in geo-distributed systems or mobile networks. This has been shown before [BNT20]. Tendermint does have a leader, but it is rotated round-robin all the time. This makes the failure of a leader less severe, as a new one will quickly be elected anyway.

3.5.5 Byzantine scenario

For BeauForT, we performed an extra benchmark with a Byzantine replica. As long as the honest replicas are still using the fast path, the Byzantine replica will send extra invalid signatures. As the signatures are only verified when a supermajority is reached, the honest replicas only realize this at the end, and they cannot find out which replica is Byzantine. Once the fast path is disabled, the signatures are verified for every message, so malicious replicas can be detected and excluded from the network. In this case, the Byzantine replica keeps the

signature intact to avoid being detected. However, it will try to slow down the consensus by not voting itself.

The latency in this Byzantine scenario is shown in Figure 3.9. BeauForT can handle Byzantine replicas very well for smaller networks, however, for networks of size 100 replicas, the tail latency becomes 7 seconds. Which might already be quite high for the use case of loyalty points. This is mostly due to the cost to verify more BLS signatures. We did not test the effect of Byzantine replicas for BFT-SMART or Tendermint. As they do not use a fast path when everyone is honest, the impact is less. However, if the current elected leader happens to be Byzantine, it can delay the consensus until some timers end and a new leader is elected [AMQ13].

3.5.6 Discussion and conclusions

We have shown that BeauForT can be used for the loyalty points use case with up to 100 different merchants, even when some of them are acting maliciously. BeauForT can achieve similar latencies as other gossip-based BFT protocols, such as Tendermint. Our evaluation also shows the trade-offs that BeauForT makes. In an optimal scenario where there is a good connection available between all replicas and no network disruptions or crashes happen, then a classical leader-based protocol such as BFT-SMART will outperform BeauForT. However, as we mention in the introduction, we envision a more ad-hoc network between low-end devices on a residential or even a mobile network, where short-term disruptions are common. Our evaluation shows that BeauForT is very robust against this kind of setting and achieves similar performance as in the optimal scenario: a transaction is always finalized within 5 seconds. A leader-based protocol such as BFT-SMART is not well suited. The temporary failure of a leader leads to long commit times and even total failure for larger network sizes. This leader also needs more resources and a direct connection to every other replica. Keeping 100 WebRTC connections open in a browser, while theoretically possible, drastically reduces performance. However, BeauForT does not impose this, since consensus can be reached gradually over time, as the full state of the proposals and votes propagates through the network. BeauForT can confirm transactions fast, in the order of seconds, without needing a complex back-end setup or wasting a lot of energy. BeauForT has a small storage footprint due to its state-based nature.

3.6 Related work

Several client-side frameworks for data synchronization between web applications exist: Legion [Lin+17], Yjs [Nic+15], Automerge [KB18], and OWebSync [JLJ21]. They make use of various kinds of Conflict-free Replicated

Data Types (CRDTs) [Sha+11a] to deal with concurrent conflicting operations and can synchronize data peer-to-peer. They are easy to set up and only require a browser and a peer-to-peer discovery service. However, they assume trusted operation as the default setting. Some work has been done in a semi-trusted setting [LLP20; Bar+21]. Recent work [Kle22; JLJ22b] also looked into making CRDTs Byzantine fault-tolerant in the eventual consistency model. BeauForT provides strong consistency.

Permissioned blockchains such as Hyperledger Fabric [And+18] have closed membership and often use a BFT consensus protocol to order transactions. For example BFT-SMART in HyperLedger Fabric [BSA14; SBV18]. The first known BFT protocol is Practical Byzantine Fault Tolerance (PBFT) [CL99]. Other protocols bring improvements to the original PBFT protocol. Zyzzyva [Kot+07] uses speculative execution which improves latency and throughput if there are no Byzantine replicas. However, its performance drops significantly if this premise does not hold. 700BFT [Aub+15] provides an abstraction for these BFT algorithms. These protocols are targeting a small number of replicas in a local network. They generally work in two phases: the first guarantees proposal uniqueness, and the second guarantees that a new leader can convince replicas to vote for a safe proposal. HotStuff [Yin+19] proposed a three-phase protocol to reduce complexity and simplify leader replacement. This makes HotStuff more scalable. All these algorithms use a leader to drive the protocol. When the leader is malicious, the performance can degrade quickly [AMQ13]. GeoBFT [Gup+20] is a topology-aware, decentralized consensus protocol, designed for geo-distributed scalability. AWARE [Ber+19] is a variant of BFT-SMART that dynamically changes the voting power of a replica depending on its latency over time, decreasing the consensus latency. BeauForT gives every replica equal voting power. In future work, BeauForT could be extended to associate a weight to each vote. While we believe this would be especially beneficial for our target environment with mobile and unreliable clients, special care will have to be given to ensure safety will stay intact. BeauForT does not use a leader and replicas communicate only to a subset of the other replicas using a gossip-like protocol.

WebBFT [BR18b] shares a similar vision of client-centric, decentralized web applications. However, they only interface to a backend BFT-SMART cluster, instead of running the BFT protocol directly between browsers. Similarly, earlier work [Mos+12] extended the Web Services Atomic Transactions specification to include BFT. However, also here the protocol is running between the backend servers, rather than between the actual web clients.

Tendermint [BKM18; Cas+21a], used in Cosmos, uses Proof-of-Stake (PoS), where voting power is based on the amount of cryptocurrency owned by each replica. Because block times are short, in the order of seconds, there is a limited number of validators Tendermint can have because finality needs to be reached for each

block. It is also not resistant to cartel forming, which allows those with a lot of cryptocurrencies to work together to control the network.

Instead of reaching consensus between all of the replicas of the network, Stellar [Maz15] and Ripple [SYB+14] reach federated Byzantine agreement between a subset of the replicas which act as representatives.

Other protocols use a randomized approach. Ouroboros [Kia+17], HoneyBadger [Mil+16], Dumbo [Guo+20] and BEAT [DRZ18] use distributed coin flipping for consensus. HoneyBadger [Mil+16] uses threshold encryption [Sho00] for censorship resilience. Algorand [Gil+17] uses Verifiable Random Functions [MVR99] to select a random committee for the next round. Avalanche [Roc18; Roc+19] uses meta-stability to reach consensus by sampling other replicas without any leader. While Avalanche is lightweight and scalable, it needs to be able to sample all other validators directly. The number of connections one can open in a browser without performance loss is limited. BeauForT supports propagation of votes over multiple hops.

Several BFT consensus protocols use a leaderless approach. Although most deterministic BFT consensus protocols designate a special leader, there exist deterministic protocols that are fully leader-free [BS10]. However, the algorithm only terminates in $f+3$ rounds in the best case, even without failures. [Ant+21] provides a leaderless algorithm that is optimal, and also provides a fast path in good conditions. It assumes replicas can directly broadcast to every other honest replica. A hybrid approach is also possible, DBFT [Cra+18] uses a so-called weak-coordinator which is not required to reach consensus, but can speed up consensus when this weak coordinator is honest. Messages are broadcasted to every other replica. Our protocol only maintains the state of the protocol, and state-changes are gossiped by dynamically computing a diff using the Merkle-tree. This naturally allows to batch multiple votes and state changes in a single network request.

There are several proposals to improve the performance and response time of BFT consensus. StreamChain [ISV18] reaches consensus over a stream of transactions instead of blocks. FabricCRDT [NMJ19] uses CRDTs to support concurrent transactions to occur in the same block, using the built-in conflict resolution of CRDTs to resolve the conflict automatically. Other approaches also borrow from CRDTs: PnyxDB [BNT20] supports commuting transactions to be applied out-of-order. A novel design for gossip in Fabric [Ber+20] improves the block propagation latency and bandwidth. Other approaches dynamically adapt the number of faults the system can withstand in reaction to threat level changes [Sil+21]. While these improvements make BFT faster, none of them try to reduce the infrastructure requirements to be able to easily set up an untrusted peer-to-peer network.

Open or permissionless blockchains such as Bitcoin [Nak08] allow everyone to participate and use Proof-of-Work (PoW) to reach agreement over the ledger. However, PoW has several flaws [BR18a]. PoW uses a lot of processing power and energy [OM14] and performs poorly in terms of latency. It assumes a synchronous network to guarantee safety and achieve finality. When this assumption is violated, temporary forks can happen in the blockchain as liveness is chosen over safety. Therefore, PoW blockchains do not offer consensus finality, instead one needs to wait for several consecutive blocks to be probabilistically certain that a transaction cannot be reverted. Simplified Payment Verification mode [Nak08] for clients can reduce the resource usage at the cost of increased centralization.

ByzCoin [Kok+16] uses PoW for a separate identity chain to guard against Sybil attacks but uses a BFT protocol to order transactions. ByzCoin makes use of collective signatures (CoSi) [Syt+16] and a balanced tree for the communication flow. CoSi makes use of aggregate signatures by constructing a Schnorr multi-signature. However, CoSi needs multiple communication round-trips to generate the multi-signature and assumes a synchronous network.

Lightning Network or state channels for Bitcoin [Lin+19] or Ethereum [Mil+19; McC+20] are *off-chain* protocols that run on top of a blockchain. A new state channel between known participants is created by interacting with the blockchain. After its creation, participants can use this channel to execute state transitions by collectively signing the new state. These transactions do not involve the blockchain and have fast confirmation times and no transaction costs. However, state channels assume all participants to be always online and honest. If this is violated, the underlying blockchain needs to be used to resolve the conflict, or a trusted third party can be used [McC+19]. BeauForT uses a similar state-transitioning protocol where only the latest collectively agreed state needs to be stored. However, BeauForT can tolerate both failing and malicious replicas, without resorting to a blockchain or a trusted third party.

Another approach is to use a trusted hardware component [Ver+13; Kap+12; DCK16; Zha+17; BDK17; Liu+18]. These are faster and less computationally intensive but require specialized hardware to be present. Moreover, trusted execution environments have been broken in the past [Lip+18; Koc+19].

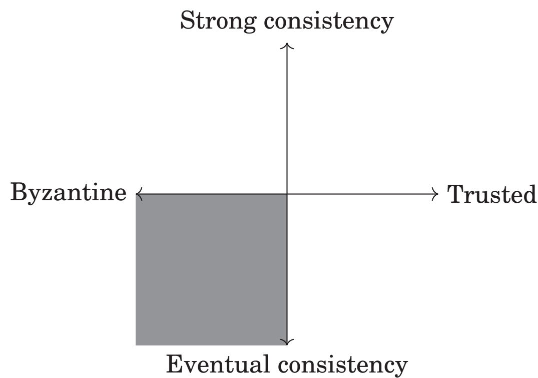
3.7 Conclusion

In this chapter, we presented BeauForT. A browser-based middleware for decentralized, community-driven web applications. BeauForT uses a client-centric, leaderless BFT consensus protocol, combined with a robust and efficient state-based synchronization protocol. BeauForT uses an optimized BLS scheme for efficient computation and storage of signatures. It supports a client-centric,

browser-based, state-based, permissioned datastore with a low infrastructure and storage footprint for small-scale, citizen-driven networks. Compared to other state-of-the-art protocols, BeauForT offers consistent and robust confirmation times to achieve finality of transactions in the order of seconds, even in failure settings and Byzantine environments. In optimal environments, with no crashes or Byzantine failure, a leader-based protocol confirms transactions faster than BeauForT. In contrast to traditional blockchains, BeauForT does not store a transaction log or blockchain, keeping the overall storage footprint small.

4

Eventual Consistency in a Byzantine Setting



Traditional Conflict-free Replicated Data Types (CRDTs) assume that all replicas are trusted, which is not necessarily the case in a peer-to-peer system. In this chapter, we present a protocol for secure state-based CRDTs which provide fine-grained confidentiality and integrity by using encryption per field in every (sub)-document. Our protocol guarantees Strong Eventual Consistency despite any Byzantine replicas. It provides a fine-grained, dynamic membership and key management system, without violating Strong Eventual Consistency or losing concurrent updates. Our evaluation shows that the protocol is suitable for use in interactive, collaborative applications.

This chapter is strongly based on our published workshop paper in *Proceedings of the 3rd International Workshop on Distributed Infrastructure for the Common Good* in 2022 [JLJ22b]. This contribution is a step forward towards decentralized, peer-to-peer deployments of collaborative web applications. Chapter 2 ignores the fact that not every peer or server will be trusted. Chapter 3 pushes the limit of browser-based applications by running a decentralized BFT consensus protocol between them but keeping client devices continuously online to ensure continuous operation might be difficult in practice. This chapter revisits the eventual consistency of Chapter 2, without making assumptions about the honesty and trustworthiness of the different peers.

4.1 Introduction

In the last decade, personal data has been stored in the cloud, rather than on a local computer. From many perspectives, this is beneficial for end-users. Data is accessible everywhere and collaboration with anyone in the world is made easy. Users also do not need to worry about data loss due to malfunction, or security breaches. However, the reality today often does not match this ideal. Few large tech companies and governments have access to vast amounts of data. They can potentially misuse it and invade the privacy of their customers or citizens to gain more money or harm political dissidents. Moving to another vendor is often very hard, if not impossible. The data is also not secure, as we hear about new security breaches almost every month, and most breaches probably even go undetected.

One solution is to move to a decentralized and client-centric approach [Kle+19; JLJ19a]. The primary copy of the data is stored under the control of the user on their local device. Data can then be replicated peer-to-peer to all other user devices and collaborators. However, a true peer-to-peer approach of end-user devices is not very durable and available. Devices are often not online at the same time, do not have a large amount of storage space, and can fail more easily or more frequently compared to a server inside a data center.

Having some kind of centralized server can be beneficial to aid the client-centric vision. The server is most of the time online, and all clients can use this server to replicate their data to each other. Even when they are never online simultaneously. Ideally, this server does not belong to a big-tech company but is under the control of the end user.

One such approach is the Solid Platform [Man+16]. With Solid, every person manages their own Personal Online Datastore (pod), either self-hosted or hosted with a third party pod-provider. Each application will store all user data inside the user's pod, and the user is in control to decide who has access to it. This also makes it easy to switch to a different application. However, the majority of the users will not choose to host their pod themselves. Instead, they will rely on a third-party company or the government to provide them with a pod. This might lead to an even bigger problem of surveillance capitalism, where few companies provide pods to their customers and gain immediately access to even more data. These providers with all data of a large number of users will also be an interesting target for hackers.

The solution we propose is a hybrid approach of a peer-to-peer network of mostly client devices and some centralized servers to improve availability and durability. An example is shown in Figure 4.1. While we trust the centralized server to keep data available, we do not want them to read or modify the actual data. A similar durability can be reached by creating a larger peer-to-peer network with friends

or family and replicating all data between all these devices. However, it should be avoided that peers are able to look into all personal data of other peers. A secure replication protocol without having access to the plain data is required.

In such systems, eventual consistency is the most pragmatic and only viable option. As devices are often offline, reaching a global consensus to have strong consistency would be nearly impossible or beyond a user-friendly time window. With strong consistency, making updates on an offline device would be impossible, and latency will be bad as clients are often only connected via WiFi or a mobile network. By opting for eventual consistency, we need a way to make sure all replicas converge to the same state after they have received all operations. One option is to use Conflict-free Replicated Data Types (CRDTs) [Sha+11a]. CRDTs are data structures that guarantee eventual consistency without explicit coordination. However, classical CRDTs do not encrypt their data and are not resilient against an attacker trying to prevent convergence.

In this chapter, we present a secure state-based CRDT protocol that extends classical state-based CRDTs with:

- Fine-grained encryption per field in every (sub)-document, to preserve confidentiality and integrity of all user data,
- Byzantine Fault Tolerance, to guarantee Strong Eventual Consistency even with Byzantine parties,
- Dynamic membership and fine-grained key management, without breaking Strong Eventual Consistency, leaking extra data, or losing updates.

Compared to other state-of-the-art approaches for secure CRDTs [Bar+21; Kle22], we provide the first framework to allow both concurrent data updates, as well as

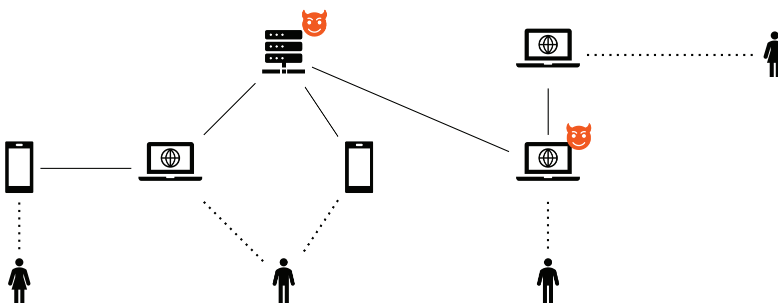


Figure 4.1: Hybrid architecture of a peer-to-peer network with a centralized server. Some users have one device, others have multiple. Some devices have access to the server, others connect peer-to-peer. Some devices can be malicious.

concurrent updates to the access control policy. This means that a user can share a document, or revoke access to a document, without losing concurrent updates to that document.

Being able to change the encryption key to give or revoke access, or to rotate the encryption key when it might be compromised is especially important for collaborative applications. For a single user, that user can easily coordinate a key rotation by bringing all his devices together, halting the system, and updating the key. However, for collaborative applications with several users, this process should be done online, without halting the system or explicit coordination between all collaborators. The protocol presented in this chapter supports this.

This chapter is structured as follows. Section 4.2 describes the system- and adversary model. We explain our protocol for secure CRDTs in Section 4.3. We evaluate our protocol in Section 4.4. Section 4.5 presents related work. We conclude in Section 4.6.

4.2 System model

In this chapter, we consider a peer-to-peer network of replicas connected by an asynchronous network (Figure 4.1). Replicas do not have a direct connection to every other replica, and they do not necessarily know the full set of replicas. Messages can be delayed, dropped, or delivered out of order, but eventually, some messages will be received. Honest replicas will follow the protocol exactly, Byzantine replicas can behave arbitrarily. There is no limit on the number of Byzantine replicas. Data is structured as a JSON document (tree) and every node has an owner, who is responsible for deciding who has access to it. Every user has an asymmetric key-pair, and other users are able to retrieve the public key of other users in a secure way, outside our protocol. We assume attackers are computationally bounded and it is infeasible to reverse the used symmetric encryption without the secret, forge the used asymmetric signatures or find collisions for the used cryptographic hash functions.

Given this system model, our protocol provides the following properties in the face of an active adversary:

- *Confidentiality*: Only users who have access can read the content.
- *Integrity*: Only users who have access can edit the content.
- *Attributability*: Each edit is attributable to the user who made the modification.
- *Availability*: As long as at least two honest replicas are available, they can work together and replicate correctly between each other.

- *Eventual delivery*: An update delivered at some correct replica is eventually delivered to all correct replicas.
- *Strong convergence*: Correct replicas that have delivered the same updates have equivalent states.
- *Termination*: All method executions terminate.

The last three properties together deliver Strong Eventual Consistency [Sha+11a]. All the properties are kept intact, even when the adversary has been given access to the actual content. The adversary is then able to arbitrarily change the content, in a way that might not make sense for the application or end-user. However, all replicas will still converge to the same end-state, and the bad updates will be attributable to the Byzantine user. The owner can then decide to revoke access.

4.3 Secure CRDTs

This section explains the protocol for our secure CRDTs. We use the term key to refer to a key from a key-value pair. When we are referring to cryptographic keys, we will always specify this as a *secret key* (k) for symmetric encryption and as a *private key* (sk) or *public key* (pk) for asymmetric encryption or signatures.

4.3.1 Encrypted CRDT

We now present two encrypted CRDT protocols. These two data structures are enough to encode a JSON tree with only maps and values into a CRDT. Arrays are not yet supported. Figure 4.2a shows an example of a JSON document and Figure 4.2b shows how it will be represented internally by the protocol.

State-based CRDTs have a merge-function, which takes as input two states of the same CRDT and produces a new state. Mathematically, these states form a join semi-lattice, and the resulting state of the merge function is the smallest state that is larger or equal to the two input states according to the partial order of the lattice. To replicate this data structure, a replica needs to send its state to another replica. This receiving replica can use the merge function with its local state and the received state to end up with the merged state.

Each CRDT is associated with an asymmetric key-pair. The public key is included in the CRDT and also functions as unique ID to reference the CRDT. The private key is only shared with users who have read-write access to the data.

LWWRegister. A LWWRegister [Sha+11a] is a data structure that holds one single value. When updating the value, the new value is associated with the current timestamp. Conflicts are resolved by selecting the value associated

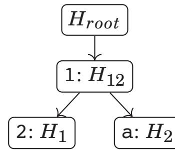
```
{
  "name": "John Doe"
}
```

(a) JSON data

```
ORMap:
  id: pk_1 (0x12)
  observed:
   - key: Enc_k1("name")
     timestamp: t_1
     σ_sk1
     pk_a
     σ_a
  removed: ∅
  σ_sk'1
  pk_a
  σ_a
```

```
LWWRegister:
  id: pk_2 (0x1a)
  value: Enc_k2("John Doe")
  timestamp: t_2
  σ_sk2
  pk_a
  σ_a
```

(b) CRDTs



(c) Trie

$sk_1 \leftarrow \mathtt{RND}()$
$pk_1 \leftarrow sk_1 \times G$
$k_1 \leftarrow \mathtt{H}(sk_1)$
$sk_2 \leftarrow \mathtt{HKDF}_{sk_1}(\texttt{"name"}, t_1)$
$pk_2 \leftarrow sk_2 \times G$
$k_2 \leftarrow \mathtt{H}(sk_2)$

(d) Key derivations

Figure 4.2: Example of how a JSON data structure can be translated into a secure CRDT data structure, consisting of two CRDTs. These CRDTs are put inside the Modified Merkle-Patricia Trie. At the bottom, we show how keys can be derived starting from one root key: sk_1.

with the highest timestamp. If the timestamps are equal, the value with the lexicographically largest hash value will be selected. Since the actual value is not used to perform a merge, the value can be encrypted using any symmetric encryption protocol, and the resulting data structure is still a CRDT.

ORMap. An ORMap [Sha+11a] is a data structure that holds a mapping of keys to values. In practice, it consists of two sets: the observed-set and the removed-set. When a new key-value pair is added to the ORMap, it is associated with a unique ID and added to the observed-set. When removing the key-value pair, it is added to the removed set. The key-value pairs included in the ORMap are all pairs included in the observed-set, which are not present in the removed-set. Thanks to the unique ID, it is possible to remove an item and add it again later. In our protocol, the values are references to other CRDTs: either a LWWRegister or another ORMap. The keys, however, need to be encrypted to maintain confidentiality. The unique ID also has to be protected against Byzantine replicas [Kle22]. If the replica itself is responsible for generating a new random ID, a Byzantine replica can easily generate duplicate IDs. Therefore, we will generate IDs deterministically based on the update. The ID of a new key-value pair is derived from the secret key linked to the ORMap and the key from the key-value pair. Since it must be possible to remove and add a key-value pair again, we also add a timestamp to each key-value pair. This timestamp is also used as input to derive the ID. There is no need to store this ID, every replica that has access to the secret key can compute the ID itself. Since the IDs and keys are therefore only available to replicas that have access to the secret key, replicas without access do not know when two items have the same ID or key, and they will therefore not be able to propagate the merge to the child CRDTs. Instead, two copies of these similar key-value pairs will be stored in the ORMap, and any other replica which does have access to the secret key can perform the merge later. This derived ID is also the value of the key-value pair: i.e., it is the ID of the child CRDT. Since this ID is only available to replicas with access to the secret key, the structure of the data is also hidden from replicas that do not have access. This means that a replica that has no access at all, will only be able to see how many individual ORMaps and LWWRegisters there are, without knowing how they belong together.

Signatures. Only users who have been given access to the secret key should be able to modify data. For this reason, every update has to be signed by the private key of the CRDT. For a LWWRegister one signature is sufficient. An ORMap will have one signature per key-value pair in the observed- and removed-set. Since the public key is also included in the CRDT, anyone can verify that an update came from a party with access to the private key. These signatures also ensure it is safe to use a public key as ID for the CRDT in a context with Byzantine actors.

You cannot reuse the same ID if you do not have access, and if you do have access, using the same ID will lead to a merge of those two CRDTs. This is equivalent to a write to the first CRDT, which you are allowed to do as you do have access to the private key.

Each update is also signed by the private key of the user who makes the update. This way, each update is attributable to the user who made the edit. The first signature (σ_{sk} in Figure 4.2b) with the private key of the CRDT proves that you have the right to modify it, the second signature (σ_a in Figure 4.2b) with your own private key proves who you are. If attributability is not required, it is possible to leave out the second signature with no other changes to the protocol.

4.3.2 Modified Merkle Patricia Trie

All individual CRDTs are stored inside a Modified Merkle Patricia Trie [Woo14] (Figure 4.2c). A Patricia Trie is a tree-shaped data structure in which items associated with a key with a common prefix, will share the same path in the tree for that prefix. A Merkle-tree [Mer88] is a tree-shaped data structure of hashes, in which the hash of a parent node is based on the hash of the hashes of the child nodes. This way, large data structures can quickly be compared or verified based on the hash in the root node of the tree. A Modified Merkle Patricia Trie combines both a Patricia Tree and a Merkle-tree. Each node in the trie also carries a hash value. This data structure is also used by Ethereum to store the state of the Ethereum blockchain [Woo14]. In Figure 4.2c, the trie consists of two items with id `0x12` and `0x1a` (the ids of the CRDTs in Figure 4.2b). As they share a common prefix, they are under the same internal node `0x1`.

The key to insert a CRDT into the trie is the ID of the CRDT. Since the ID is also a public key, they are random, and therefore the trie will be relatively well-balanced. By using the Merkle-tree, two replicas can efficiently exchange updates between each other. The replicas can compare the root hash of the trie. If the hashes match, the two tries are exactly the same, and no replication is required. If the hashes do not match, the replicas will descend in the tree and send the hashes of the next level in the tree. This process continues until it reaches the leaves of the tree. At this time, the updated CRDTs can be sent and merged. This process is similar to the replication process in OWebSync [JLJ21].

4.3.3 Key derivation and rotation

In the previous two sub-sections, we created a trie of individual CRDTs which contain signatures and are partially encrypted by the respective private and secret keys of the CRDT. The secret key can be derived from the private key by using a key derivation function, for example, HKDF [KE10]. This leads to

one encryption key to manage per CRDT. However, as already indicated in the paragraph on ORMaps, the key material for children is derived from the parent key. Instead of directly deriving the ID for a key-value pair in an ORMap, we derive a private key. We can then use this private key to derive the secret key and public key. This public key is also the ID. A user who has access to the full document tree only needs access to the private key of the root and can derive all other keys from this single key. This makes sharing a document and key management easy. An example of this derivation process is shown in Figure 4.2d.

When access is revoked from a user, the encryption key will have to be updated. Otherwise, that user still has access to the secret key, and would still be able to read and write. A new private key is derived from the parent private key, the key (from the path in the tree), and the current timestamp. Because the timestamp will be different, a brand new private key is generated and the CRDT can be re-encrypted with the corresponding secret key. Since the private key is changed, the public key will also be different and the CRDT will be stored under a different ID in the trie. Only users that have access to the private key of the parent are able to do a key rotation. The new key will be derived from this parent key and a signature with the parent key is required. Key rotations for the root are not possible, as there is no parent key. The owner of the document should therefore only delegate access to sub-trees instead of the root directly.

Because these CRDTs end up in different places of the trie, they can co-exist for a while. Replicas that are not yet informed about the key rotation can still perform updates on the old version, while other replicas can do updates on the new version. Any replica that has access to both the old and the new version knows those two CRDTs are in fact the same CRDT and can perform a merge operation as usual. Replicas that do not have access to both private keys are unaware they are the same CRDT and will treat them as two separate CRDTs.

4.3.4 Global time

Common wisdom in the field of distributed systems is that you cannot have a global time in a distributed system. Although this is true, a coarse-grained global timestamp is still possible. The Ethereum blockchain, for example, includes a timestamp in every block header. In the Geth implementation, a timestamp of a new block has to be larger than the timestamp of the previous block and less than 15 seconds in the future of the current time of a replica. Similar timestamps and rules are present in other blockchains.

We use similar rules for the timestamps used in our protocol. A timestamp may only be at most one minute in the future, otherwise, the replica will not accept it and stop communication with the other replica. Replicas need to keep their clocks

correct. These days, internet-connected devices automatically synchronize their time with an internet time server and are generally correct within one minute.

A Byzantine replica can reuse timestamps without problem since the lexicographic order of the hash value will then be used as a tie-breaker. Such a replica can also get an edge over other replicas by always using a timestamp one minute in the future. Because its timestamps are generally larger, when an update is done simultaneously, its update is more likely to win in the last-writer-wins conflict resolution. This is however only possible for Byzantine replicas that have access to the private key, i.e., replicas with write access. Replicas without access can never change anything. Hence, such replicas do not get to choose a timestamp. This edge that a Byzantine replica has is only present for short intervals. On larger intervals, the correct user intention will be kept. For example, when user A makes a change in the morning, and another user B changes the same data in the afternoon, the change of user B will be chosen. User A can of course keep increasing the timestamp of his update, but this is equivalent to a new write by a replica that has write access, so this is allowed.

4.3.5 Discussion

This section presented a novel protocol for secure and confidential CRDTs. Since replication is state-based, there are no client-specific identifiers kept for the replication. Replicas do not need to know every other replica. Only the replica modifying the access control policy has to know the public key of all users with read and write access. This makes the protocol extremely robust against network failures and long-term disconnects [JLJ21]. Centralized servers which are only there to improve the availability and durability of the replication between clients, do not need any private key material to function.

The current protocol will keep both old and new versions of a CRDT after a key rotation forever. This is not required, once the new version is created, the old one can be removed. With concurrent edits, it is possible that the old version will resurface again, but after each merge with the new version, it is removed again. After some time, all replicas will know about the key rotation and all updates will be applied to the new version and the old version will never resurface. If the replica that has been revoked access by the key rotation makes an update, it will not be merged in the new version, but simply be discarded. This is possible due to the coarse-grained timestamps. So, there is a small interval of less than a minute in which its updates will still be accepted. For most application cases that already opted for eventual consistency, this is acceptable.

To be able to determine whether an update from an old version of the CRDT should be merged with the new version, a list of all users having access to it is

required. This is a list of public keys and only needs to be kept at the point in the JSON tree where you give access to other users. This can be encrypted as well, as only replicas with access to the actual data will have to use the list to potentially merge data updates across key updates. Replicas that decide to rotate a key can also use this list to determine who should have access to the new key. Replicas with no access to the data do not use this list and instead rely on the key-pair of each CRDT to determine whether access was correctly granted.

The only cryptographic protocols used are plain symmetric encryption (e.g., AES), public-key cryptography (e.g., RSA or ECDSA), and hashing (e.g., SHA256). Furthermore, we use a key derivation algorithm based on these protocols (HKDF). As these are older, well-tested protocols, we can be more certain of their correctness and safety. There are also more well-tested and maintained libraries available, making it possible to implement our protocol in multiple languages. Also, the availability of hardware support for some of these will be good for the performance on client devices.

4.4 Evaluation

We implemented the protocol in a JavaScript-based web application, without browser plugins. For our experiments, we launched up to 30 virtual machines in the Azure public cloud (F2s_v2 with 2 vCPU and 4 GB RAM) in the same data center. To emulate geographically distributed users, we use the Linux `tc` tool to increase the network latency between each VM to an average of 100 ms with 50 ms jitter. Each VM contains one Chromium browser. Every client makes one write every second. We are interested in the interactive latency, i.e., after one client makes an update, how long does it take for other clients to receive it. To compare the overhead of our encrypted and Byzantine fault-tolerant approach to a regular approach without security, we also performed the same experiments with the open-source version of OWebSync. OWebSync [JLJ21; JLJ22a] is a state-based CRDT framework, in which all clients are trusted.

The performance results are shown in Table 4.1. We compare the protocol from this chapter (secure CRDTs) with OWebSync (baseline) for three different numbers of active replicas. With 30 different replicas, each making one request per second, the average latency is 1.6 seconds before an update is visible to other replicas. With smaller network sizes, the latency is lower. OWebSync has a much lower latency, of 0.5 seconds, even for larger network sizes as no cryptographic operations are required there. Overall, the latency is low enough to be considered interactive when multiple users are collaboratively working on the same document. The storage overhead of the protocol ranges from 16 to 19 times, compared to the size of the raw data. For OWebSync this overhead is only 4 times. The overhead comes from the extra metadata required for state-

Table 4.1: Performance characteristics of the proposed protocol, compared to a baseline protocol without any security.

# replicas		10	20	30
Latency [s]	secure CRDTs	0.62	0.76	1.59
	baseline	0.56	0.50	0.54
Storage overhead	secure CRDTs	×16	×16	×19
	baseline	×4	×4	×4
Bandwidth [kbps]	secure CRDTs	229	806	830
	baseline	231	1382	4160
CPU usage [%]	secure CRDTs	19	56	72
	baseline	13	42	83

based CRDTs, but also from the signatures and encrypted data. This leads to a bandwidth usage of 830 kbps for 30 replicas, which is readily available on any mobile network. Interestingly, the network usage for our baseline, OWebSync is a factor 5 higher, even though the actual storage size is much lower. The explanation for this is two-fold. First, OWebSync uses a Merkle-tree which is based on the actual tree-structure of the data, while our protocol uses a much better balanced Merkle-Patricia Trie. This allows replicas to propagate updates more fine-grained, i.e., when only a leaf in the JSON data changes, we do not need to replicate the intermediate nodes of the JSON tree. Second, as the latency of OWebSync is much better, it does more traversals of the Merkle-tree, while our protocol does less as more updates can be batched in the same tree traversal given the higher latency. This second point also explains the discrepancy in CPU usage for the network with 30 replicas, as we would expect that our protocol always has a higher CPU usage compared to a solution without any signatures and cryptography.

To conclude, we have shown that our protocol for secure CRDTs, which tolerates Byzantine replicas, and which supports very fine-grained access control by encrypting every field in every (sub)-document with a different key, can be used for interactive, collaborative document editing. The price to pay is a significant increase in the size of the data (up to 19 times).

4.5 Related work

This section covers related work that also tries to reach eventual consistency in an adversarial context.

Snapdoc [KKB19] is a collaborative peer-to-peer text editing protocol. New replicas can be added to the network by only sending a snapshot of the data, including a cryptographic proof of the integrity. They can keep the edit history private for new replicas, but new replicas can still attribute all changes, as well as verify the integrity. This is made possible by using RSA accumulators and Merkle-proofs. However, the new replica can only accept operations that are created after the snapshot. When an operation, not included in the snapshot, was created before or concurrent to a snapshot, the new replica will have to request a new snapshot and do the verification process again.

In [LLP20], Linde, Leitão, and Preguiça present a system that protects against rational misbehaving clients in causal consistency. However, servers are considered trusted, and the focus is on detecting the Byzantine client, rather than avoiding divergence at all.

In [Bar+21], Barbosa, B. Ferreira, J. Marques, Portela, and Preguiça extend standard CRDTs with cryptographic protocols. The paper focuses on a client-server context, where servers are unable to see the actual data. The same approach can most likely also be used in a peer-to-peer setting. However, the provided algorithms only work as long as the same cryptographic key is used. Switching to a new key will require coordination between the replicas. Furthermore, the approach focuses on confidentiality and does not tolerate an active Byzantine replica.

In [Kle22], Kleppmann shows how operation-based CRDTs can be adapted to tolerate Byzantine replicas. The paper lists four techniques that are together sufficient to make most operation-based CRDT tolerate Byzantine replicas. The techniques are: constructing a hash-graph of all updates, with links to predecessor; ensuring eventual delivery, which could be done by using the hash graph; constructing unique IDs, which cannot be controlled by an attacker; and ensuring that replicas only look at the predecessors of an update to check the validity of it. However, the paper only focuses on maintaining eventual consistency, and not on confidentiality.

Secure Scuttlebutt [Tar+19] is a peer-to-peer event-sharing protocol, using individual append-only logs. However, strong eventual consistency can only be reached on replicas that subscribe to the same logs. Furthermore, the append-only logs will grow without bounds.

4.6 Conclusion and future work

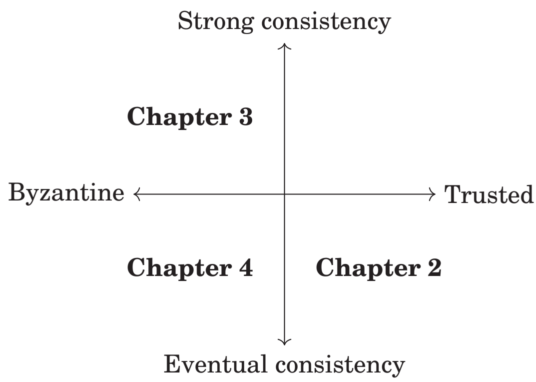
In this chapter, we presented a protocol for secure state-based CRDTs. We have shown that Strong Eventual Consistency can be reached even in settings with

Byzantine replicas. We have also shown that a key rotation does not have to break Strong Eventual Consistency and that you can do this concurrently, while other users are still making updates with the old keys. The key idea to support this is to store all CRDTs inside a Merkle-Patricia Trie, and only allow replicas that have access to both the old and the new secret key to merge two different versions of the same CRDT.

In future work, we will extend this protocol with online pruning to remove old versions which are not necessary anymore from the trie. Arrays are also not yet supported. The current protocol uses basic, state-of-practice cryptography. More research is required to evaluate whether newer cryptography protocols such as attribute-based encryption [SW05] can offer any benefits.

5

Conclusion



This chapter concludes this dissertation. We begin by summarizing the contributions presented in the previous chapters. Then we discuss the limitations of our proposed solutions and present opportunities for future work.

5.1 Summary of contributions

This dissertation has addressed several challenges in the scope of client-centric replication. In this section, we provide an overview of our contributions toward solving these challenges. We presented three major contributions in the form of three client-centric replication protocols for different environments with different trust and consistency levels. All three protocols have been implemented in a browser-based middleware. In summary, this dissertation offers the following contributions:

1. **A state-based Conflict-free Replicated Data Type protocol that supports fine-grained delta-merging and conflict resolution.** To counteract the problems with operation-based approaches and using client identifiers, we opted to use state-based CRDTs which are much more robust and resilient to network failures and which can achieve constant performance over time. State-based CRDTs were not previously used on the client, as the size of the state is typically much larger than the size of the operations, making it unsuitable for low-bandwidth connections. Existing delta-state-based approaches tried to solve this problem but at the same time re-introduced the problems of operation-based approaches by using vector clocks and client identifiers. We proposed a new state-based approach that can dynamically determine which fine-grained parts of the data need to be sent to merge correctly by using a Merkle-tree on top of the data structure. This approach eliminates the need for client identifiers or vector clocks. Hence it can achieve constant performance, without deterioration over time.
2. **A lightweight leaderless Byzantine Fault Tolerant consensus protocol.** Traditional BFT consensus protocols based on a leader are not suitable for client-centric replication, as they require a stable leader with enough resources to be performant. We proposed a leaderless protocol in which both the state as well as the consensus votes are replicated over a state-based gossip protocol. This replication method is highly robust, as consensus votes can be replicated through multiple hops and routes to all other replicas. This also naturally allows batching of many different consensus votes in a single network request. In contrast to existing permissioned BFT frameworks such as Hyperledger Fabric, this setup is lightweight in terms of infrastructure requirements.

3. **A Byzantine Fault Tolerant Conflict-free Replicated Data Type protocol.** State-of-the-art CRDT protocols for collaborative applications mostly assume a trusted environment. This is also the case for the first contribution of this dissertation (Chapter 2). We argue that a peer-to-peer client-centric application can benefit from adding untrusted servers or other untrusted clients in terms of availability, performance and robustness. However, as replicas are no longer trusted, existing protocols cannot be used, because strong eventual consistency is no longer guaranteed. We proposed a novel CRDT protocol that guarantees strong eventual consistency, even in the presence of Byzantine faults. This includes also fine-grained encryption per field in every sub-document to preserve the confidentiality and integrity of all data. Furthermore, membership changes, i.e., who can read and write which parts of the document, are possible without breaking strong eventual consistency, leaking extra data, or losing concurrent updates.

These three contributions address the challenges and goals presented in the introduction, offering solutions across different environments.

We achieved *resilient* replication by opting for a state-based approach instead of working directly with the updates. This eliminates the need to determine which updates must still be sent to certain replicas and which can be discarded. We achieved *interactive* replication by using Merkle-trees to quickly calculate a delta-state to be sent over to the other replicas. This way, we still only need to send a small update, similar to operation-based approaches. Recovery is also efficient, as the same replication process of traversing the Merkle-tree once can be used to fully replicate all updates. Our second contribution, BeauForT (Chapter 3), provides strong consistency. This means that at least a supermajority of the replicas has to be online to achieve interactive performance. However, short-term failures are no longer a problem, as the protocol can quickly recover from them with the same replication protocol as in the normal scenario.

Our protocols have *limited storage overhead* and do not suffer from metadata explosion. A state-based approach has the benefit that we no longer need client identifiers, vector clocks, or totally ordered logs to assist with the replication, as is the case for operation-based or delta-state-based approaches. The current state and some metadata are enough to reliably replicate the data. By keeping the metadata size constant over time, the proposed protocols maintain consistent performance, without degradation due to increased metadata or log size.

5.2 Limitations and future directions

In this dissertation, we presented several contributions in the scope of client-centric replication. We showed that our solutions solve the typical problems of

local-first software and blockchains on a scale with tens of client replicas. They provide acceptable performance in optimal conditions. However, they outperform the state-of-the-art of other client-centric collaborative software in more realistic conditions, where short-term network and device failures are common. Our solutions are much more robust and resilient in such unstable environments. This section discusses the current limitations and future research directions to overcome these.

Unclear conflict resolution

Chapter 2 and Chapter 4 offer something that resembles last-writer-wins conflict resolution towards application developers. In Chapter 3, no conflict resolution is present, instead strong consistency is achieved for every update to make sure only one state-transition is happening. This is conflict-avoidance, rather then conflict-resolution.

The last-writer-wins conflict resolution makes it easy for the application developer, as the application code will never have to deal with conflicts itself. However, there are choices to be made by the developer that can affect the actual conflict resolution. Since we are dealing with a tree-based structure, developers can choose to construct a very fine-grained tree, with small leaf values, or they can go more coarse-grained, with large leaf values. The last-writer-wins conflict resolution is only applied to the leaf values. A less fine-grained tree with more data stored as strings in the leafs means that there is a higher chance that a concurrent modification can lead to one of the updates being lost. The intermediate levels in the tree maintain an add-wins conflict resolution in which a concurrent add and remove will result in the item being present. However, this resolution is only the case for add and removes on the same level in the tree, if the concurrent remove was for a higher level in the tree, then the result will be that the whole subtree is removed, even though there are more recent changes in lower levels of the tree. This violates the last-writer-wins strategy that we imposed earlier. From a user's point of view, this can be confusing. Most of the time, they will experience last-writer-wins conflict resolution, with some exceptions based on how the application developer structured the tree. This trade-off is both for usability for the application developer, as well as for performance reasons. The fact that a whole sub-tree is removed, including future updates deeper in the tree, means that we only have to store a single tombstone even though the sub-tree itself could be very large.

Potential solutions. Instead of using a LWWRegister CRDT, we can extend the protocols to also include Multi-Value Registers [Sha+11b] (MVRegister). This CRDT represents a single value, just like a LWWRegister. However, when two conflicting operations happen, both values are kept in the MVRegister and it is

up to the application developer to select the correct value to keep and which one to discard (or to set a new value based on an application-level merge of the two values). The developer can also opt to delegate this choice to the user. While this allows for more transparent and application-specific conflict resolution, the application developer will now have to deal with conflict resolution.

Limited scalability in terms of participants

All three protocols discussed in this dissertation are limited to a small number of replicas, namely tens of replicas to at most 100 replicas. This is due to the fact that all data is replicated on all replicas. Traditional server-centric cloud applications would replicate data over a small number of servers, typically 3 or 5. These servers are normally located within one or multiple data centers where the replicas can communicate with each other over a high-bandwidth network connection. Many clients can connect to these servers to read and write data. Keeping all data replicated on all mobile client devices forms an inherent scalability problem in the system. Scaling to more replicas requires increased computational power, network bandwidth, or lower latency.

Potential solutions. We can expect that these requirements will become available in the long term future, as computing devices become more and more powerful. The last requirement, more network bandwidth with lower latency, however, might be reached much sooner with the introduction of 5G networks or even 6G networks. Having a low-latency, high-bandwidth network available would greatly improve the performance of our proposed solutions. This is especially the case for BeauForT (Chapter 3), in which global consensus is required for every write operation. This means that a supermajority of all replicas needs to be aware of the decision of a supermajority of the replicas which requires a lot of communication. The performance of the system is therefore directly dependent on the number of replicas and the latency of the network between the replicas. For the eventual consistent protocols (Chapter 2 and Chapter 4) the requirement is much lower as updates can be directly applied locally. However, for collaborative applications, the latency of the network is still a major factor in the performance of the system. With the introduction of 5G and 6G networks, we can expect that the scalability of our proposed solutions will increase as they are now mostly limited by the network latency. Since all of our solutions use a gossip protocol to replicate the data over multiple hops, the latency of the network is a major factor in the performance of our systems.

Besides these advancements in available resources, namely network latency, network bandwidth, and computational power, another way to scale our solutions is to use a hybrid approach. In a hybrid approach, we move away from the client-centric model in which the clients themselves directly act as a replica. Instead,

we can use a few large-scale servers or edge devices, ideally closely located to the clients. Each device can then service many low-end clients in a local area. This way the data only has to be replicated to tens of devices, which our solutions are currently capable of, while many more clients can access the data. Since our protocols can scale to much more replicas than the typical 3 to 5 replicas of a centralized cloud solution, data can be located closer to the clients, even when clients are distributed globally. This hybrid approach brings lower latency and higher availability to the clients compared to a traditional cloud-based approach, while still being able to scale to a large number of clients compared to the native client-centric approach.

Another possible way to improve the scalability is to tailor the protocol to the specific needs of the application. For example, in Chapter 3, strong consensus is required for every change. However, we could allow the application developer to choose which properties are required for each operation. In the loyalty use case, we could relax the consistency requirements when handing out loyalty points to customers. They do not have to be confirmed strongly consistent quickly, as is required at checkout to redeem them. Instead, they could be eventually consistent at first and become strongly consistent confirmed later in batch.

Limited scalability in terms of data size

A second limitation of our solutions is the scalability in terms of data size. All experiments were conducted with small data sets, expressed in kilobytes and megabytes. The data size is of course limited to the available storage space on the client devices and the network bandwidth between the replicas. However, the current limitations are mostly due to the use of a browser as the underlying platform. All discussed protocols in this dissertation are implemented in a JavaScript-based framework, which is executed in a browser. Storing large amounts of data in the browser is not yet feasible today. Although browsers support standardized APIs for data storage, such as IndexedDB, its performance is still not sufficient for the workload required by our protocols. Our protocols store tree-shaped data in a flat structure of many key-value pairs. Updating a single leaf in the tree requires updating all nodes on the path from the leaf to the root, amplifying the number of writes on the underlying database.

Potential solutions. Moving away from a purely browser-based solution can improve performance when handling larger amounts of data. It can also be expected that browser vendors will improve the performance of their storage APIs in the future, or that new storage APIs are standardized and adopted. Again, a hybrid approach can be used here as well to scale our solutions. In a hybrid approach, servers, and edge devices can be used to store all of the data, while the actual mobile client device only stores a small part of it. This is very similar to

what, for example, Microsoft OneDrive does today: some files are stored on the device and are therefore available offline, and other files are stored on the server and are only available online. But once accessed, the file is downloaded to the device and is available offline in the future. The problem then becomes deciding which part of the data to store locally. Note that this part of the data is still an authoritative copy, not a cached version. Clients can make edits to it offline and these edits have to be replicated to the other peers.

The future of client-centric replication

In this dissertation, we presented three different protocols for client-centric replication. All three protocols have limited scalability in terms of participants and data size. Their ideal use case is collaborative applications with a small number of participants where both real-time interactivity is required, as well as resilience to short-term network and device failures, and possibly long-term offline usage. However, expecting that clients are always online is unrealistic. While current networking abilities such as 4G and 5G make it possible for a mobile device to be online most of the time, there are still many situations where clients are not online such as tunnels, airplanes, trains, rural areas, etc. Furthermore, a client might be online but battery life is limited, which means it is not desirable to keep a mobile device actively responding to requests from other replicas. A mix of mobile computing devices, local edge devices such as home routers, fog devices for example hosted by the ISP, and cloud servers is probably the better solution for the future. This becomes a hierarchical peer-to-peer system, in which some peers can act as super-peers, connected to many other peers, and other peers such as mobile devices are connected to a single super-peer and other local mobile clients. In this case, all advantages of local-first software are still present as the local clients still act as fully authoritative replicas. However, the peers closer to the cloud can be used to synchronize the data between the different clients more efficiently and more reliably. This is especially the case when the clients are not online at the same time. There is, however, no need for the peers to have any inside knowledge of the data, only the clients need to be aware of the content of the data. These super-peers are simply used as a synchronization mechanism between the clients. This is the approach taken in Chapter 4 in which we presented a CRDT protocol that can work with untrusted replicas that do not have access to the actual data.

In conclusion, this dissertation explored client-centric replication not as the end goal, but as a promising approach to be used in conjunction with cloud-based server-centric approaches and the various edge- and fog-based approaches in between. A hybrid approach, especially for collaborative applications, can bring performance, resilience, and confidentiality due to the local-first approach, as well as availability and scalability by using cloud servers and edge devices.

Bibliography

[Alm99] Werner Almesberger. *Linux network traffic control – implementation overview*. Tech. rep. EPFL, 1999. URL: https://www.almesberger.net/cv/papers/tcio8.pdf → 38.

[AMQ13] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. "RBFT: Redundant byzantine fault tolerance". In: *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*. ICDCS '13. IEEE, 2013, pp. 297–306. DOI: 10.1109/ICDCS.2013.53 → 56, 80, 81.

[And+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018. DOI: 10.1145/3190508.3190538 → 81.

[Ant+21] Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. "Leaderless Consensus". In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 2021. DOI: 10.1109/ICDCS51616.2021.00045 → 82.

[ASB15] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. "Efficient State-Based CRDTs by Delta-Mutation". In: *Networked Systems*. Springer International Publishing, 2015, pp. 62–76. DOI: 10.1007/978-3-319-26850-7_5 → 7, 23.

[ASB18] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. "Delta state replicated data types". In: *Journal of Parallel and Distributed*

Computing 111 (2018), pp. 162–173. DOI: 10.1016/j.jpdc.2017.08.003 → 7, 23.

[AT19] Alex Auvolat and François Taïani. "Merkle Search Trees: Efficient State-Based CRDTs in Open Networks". In: *38th IEEE International Symposium on Reliable Distributed Systems*. SRDS '19. Lyon, France: IEEE, 2019. DOI: 10.1109/SRDS47363.2019.00032. URL: https://hal.inria.fr/hal-02303490 → 20, 48.

[Aub+15] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. "The Next 700 BFT Protocols". In: *ACM Trans. Comput. Syst.* 32.4 (2015). DOI: 10.1145/2658994 → 81.

[Aum19] Jean-Philippe Aumasson. *Too Much Crypto*. Cryptology ePrint Archive, Paper 2019/1492. 2019. URL: https://eprint.iacr.org/2019/1492 → 74.

[Bal+18] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. "IPA: Invariant-Preserving Applications for Weakly Consistent Replicated Databases". In: *Proc. VLDB Endow.* 12.4 (Dec. 2018), pp. 404–418. DOI: 10.14778/3297753.3297760 → 48.

[Bar+21] Manuel Barbosa, Bernardo Ferreira, João Marques, Bernardo Portela, and Nuno Preguiça. "Secure Conflict-Free Replicated Data Types". In: *International Conference on Distributed Computing and Networking 2021*. ICDCN '21. Nara, Japan: Association for Computing Machinery, 2021, pp. 6–15. DOI: 10.1145/3427796.3427831 → 81, 90, 100.

[BAS17] Carlos Baquero, Paulo Sergio Almeida, and Ali Shoker. *Pure Operation-Based Replicated Data Types*. 2017. arXiv: 1710.04469 → 48.

[BBR16] Carlos Bartolomeu, Manuel Bravo, and Luis Rodrigues. "Dynamic Adaptation of Geo-replicated CRDTs". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. Pisa, Italy: Association for Computing Machinery, 2016, pp. 514–521. DOI: 10.1145/2851613.2851641 → 47.

[BDK17] Johannes Behl, Tobias Distler, and Rudiger Kapitza. "Hybrids on Steroids: SGX-Based High Performance BFT". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: Association for Computing Machinery, 2017, pp. 222–237. DOI: 10.1145/3064176.3064213 → 83.

[BDN18] Dan Boneh, Manu Drijvers, and Gregory Neven. "Compact multi-signatures for smaller blockchains". In: *Advances in Cryptology*. ASIACRYPT 2018. Springer International Publishing, 2018, pp. 435–464. DOI: 10.1007/978-3-030-03329-3_15 → 75, 76.

[Ber+19] Christian Berger, Hans P. Reiser, João Sousa, and Alysson Bessani. "Resilient Wide-Area Byzantine Consensus Using Adaptive Weighted Replication". In: *38th Symposium on Reliable Distributed Systems*. SRDS '19. Lyon, France: IEEE, 2019. DOI: 10.1109/SRDS47363.2019.00029 → 81.

[Ber+20] Nicolae Berendea, Hugues Mercier, Emanuel Onica, and Etienne Riviere. "Fair and Efficient Gossip in Hyperledger Fabric". In: *IEEE 40th International Conference on Distributed Computing Systems*. ICDCS '20. Singapore, Singapore: IEEE, 2020. DOI: 10.1109/ICDCS47774.2020.00027 → 82.

[Ber17] Tim Berners-Lee. *Three challenges for the Web, according to its inventor*. 2017. URL: https://webfoundation.org/2017/03/web-turns-28-letter/ → 3, 5.

[BFN96] Tim Berners-Lee, Roy Fielding, and Hendrik Nielsen. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. May 1996. DOI: 10.17487/RFC1945 → 2.

[BG19] Jim Bauwens and Elisa Gonzalez Boix. "Memory Efficient CRDTs in Dynamic Environments". In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 48–57. DOI: 10.1145/3358504.3361231 → 47.

[Biy17] Cihan Biyikoglu. *Under the Hood: Redis CRDTs (Conflict-free Replicated Data Types)*. White paper. Redis Labs, 2017. URL: https://redislabs.com/docs/under-the-hood/ → 48.

[BKM18] Ethan Buchman, Jae Kwon, and Zarko Milosevic. *The latest gossip on BFT consensus*. 2018. arXiv: 1807.04938 → 8, 55, 56, 77, 81.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. "Short signatures from the Weil pairing". In: *Advances in Cryptology*. ASIACRYPT 2001. Springer Berlin Heidelberg, 2001, pp. 514–532. DOI: 10.1007/3-540-45682-1_30 → 56, 74, 75.

[BNT20] Loick Bonniot, Christoph Neumann, and François Taiani. "PnyxDB: a Lightweight Leaderless Democratic Byzantine Fault Tolerant Replicated Datastore". In: *The 39th IEEE International Symposium on Reliable Distributed Systems*. SRDS '20. Shanghai, China: IEEE, 2020, pp. 155–164. DOI: 10.1109/SRDS51746.2020.00023 → 79, 82.

[Bon+03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. “Aggregate and verifiably encrypted signatures from bilinear maps”. In: *Advances in Cryptology*. EUROCRYPT 2003. Springer Berlin Heidelberg, 2003, pp. 416–432. DOI: 10.1007/3-540-39200-9_26 → 75.

[BP16] Carlos Baquero and Nuno Preguiça. “Why Logical Clocks Are Easy”. In: *Commun. ACM* 59.4 (Mar. 2016), pp. 43–47. ISSN: 0001-0782. DOI: 10.1145/2890782 → 7.

[BR18a] Christian Berger and Hans P. Reiser. “Scaling Byzantine Consensus: A Broad Analysis”. In: *Proceedings of the 2Nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. SERIAL ’18. Rennes, France: Association for Computing Machinery, 2018, pp. 13–18. DOI: 10.1145/3284764.3284767 → 83.

[BR18b] Christian Berger and Hans P. Reiser. “WebBFT: Byzantine Fault Tolerance for Resilient Interactive Web Applications”. In: *Distributed Applications and Interoperable Systems*. Springer International Publishing, 2018, pp. 1–17. DOI: 10.1007/978-3-319-93767-0_1 → 81.

[Bra14] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7158. Mar. 2014. DOI: 10.17487/RFC7158 → 20.

[Bri+15] Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. “Conflict-free partially replicated data types”. In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. Vancouver, BC, Canada: IEEE, 2015, pp. 282–289. DOI: 10.1109/CloudCom.2015.81 → 47.

[BS10] Fatemeh Borran and André Schiper. “A Leader-Free Byzantine Consensus Algorithm”. In: *Distributed Computing and Networking*. ICDCN 2010. Springer Berlin Heidelberg, 2010, pp. 67–78. DOI: 10.1007/978-3-642-11322-2_11 → 82.

[BSA14] Alysson Bessani, Joao Sousa, and Eduardo E. P. Alchieri. “State Machine Replication for the Masses with BFT-SMART”. In: *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN ’14. Atlanta, GA, USA: IEEE, 2014. DOI: 10.1109/DSN.2014.43 → 8, 55, 56, 77, 81.

[Bur+12] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. “Cloud Types for Eventual Consistency”. In: *ECOOP 2012 – Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 283–307. DOI: 10.1007/978-3-642-31057-7_14 → 48.

[But+13] Vitalik Buterin et al. *A next-generation smart contract and decentralized application platform*. White paper. ethereum.org, 2013 → 6, 55.

[But+20] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. *Combining GHOST and casper*. 2020. arXiv: 2003.03052 → 55.

[Buy+09] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility". In: *Future Generation computer systems* 25.6 (2009), pp. 599–616. DOI: 10.1016/j.future.2008.12.001 → 2.

[Cas+21a] Daniel Cason, Enrique Fynn, Nenad Milosevic, Zarko Milosevic, Ethan Buchman, and Fernando Pedone. "The design, architecture and performance of the Tendermint Blockchain Network". In: *40th International Symposium on Reliable Distributed Systems*. SRDS '21. Chicago, IL, USA: IEEE, 2021, pp. 23–33. DOI: 10.1109/SRDS53918.2021.00012 → 77, 81.

[Cas+21b] Daniel Cason, Nenad Milosevic, Zarko Milosevic, and Fernando Pedone. "Gossip Consensus". In: *Proceedings of the 22nd International Middleware Conference*. Middleware '21. Québec city, Canada: Association for Computing Machinery, 2021, pp. 198–209. DOI: 10.1145/3464298.3493395 → 54, 63.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luis E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Berlin, Heidelberg, 2011. DOI: 10.1007/978-3-642-15260-3 → 58.

[CL99] Miguel Castro and Barbara Liskov. "Practical Byzantine fault tolerance". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 173–186. DOI: 10.5555/296806.296824. URL: https://www.usenix.org/legacy/publications/library/proceedings/osdi99/full_papers/castro/castro.ps → 8, 55, 56, 81.

[Coh03] Bram Cohen. "Incentives build robustness in BitTorrent". In: *Workshop on Economics of Peer-to-Peer systems*. Vol. 6. Berkeley, CA, USA. 2003, pp. 68–72 → 2.

[Cra+18] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. "DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains". In: *2018 IEEE 17th International Symposium*

on Network Computing and Applications (NCA). Cambridge, MA, USA, 2018, pp. 1–8. DOI: 10.1109/NCA.2018.8548057 → 56, 82.

[Dan+22] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. "Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22. Rennes, France: Association for Computing Machinery, 2022, pp. 34–50. DOI: 10.1145/3492321.3519594 → 59.

[DCK16] Tobias Distler, Christian Cachin, and Rudiger Kapitza. "Resource-Efficient Byzantine Fault Tolerance". In: *IEEE Transactions on Computers* 65.9 (2016), pp. 2807–2819. DOI: 10.1109/TC.2015.2495213 → 83.

[De +19] Kevin De Porre, Florian Myter, Christophe De Troyer, Christophe Scholliers, Wolfgang De Meuter, and Elisa Gonzalez Boix. "Putting Order in Strong Eventual Consistency". In: *Distributed Applications and Interoperable Systems*. Springer International Publishing, 2019, pp. 36–56. DOI: 10.1007/978-3-030-22496-7_3 → 48.

[De +21] Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. "ECROs: Building Global Scale Systems from Sequential Code". In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485484 → 48.

[DeC+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. Vol. 41(6). SOSP '07. Stevenson, Washington, USA: Association for Computing Machinery, 2007, pp. 205–220. DOI: 10.1145/1294261.1294281 → 9, 20, 42, 48.

[Dem+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. "Epidemic Algorithms for Replicated Database Maintenance". In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*. PODC '87. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1987, pp. 1–12. DOI: 10.1145/41840.41841 → 8.

[Dem+94] Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, and Brent Welch. "The Bayou Architecture: Support for Data Sharing Among Mobile Users". In: *1994 First Workshop on Mobile Computing Systems and Applications*. IEEE, 1994, pp. 2–7. DOI: 10.1109/WMCSA.1994.37 → 3, 10.

[DI16] Quang-Vinh Dang and Claudia-Lavinia Ignat. “Performance of real-time collaborative editors at large scale: User perspective”. In: *Internet of People Workshop, 2016 IFIP Networking Conference*. Proceedings of 2016 IFIP Networking Conference, Networking 2016 and Workshops. Vienna, Austria: IEEE, May 2016, pp. 548–553. DOI: 10.1109/IFIPNetworking.2016.7497258 → 40, 42.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. “Consensus in the Presence of Partial Synchrony”. In: *J. ACM* 35.2 (1988), pp. 288–323. DOI: 10.1145/42282.42283 → 58.

[DRZ18] Sisi Duan, Michael K. Reiter, and Haibin Zhang. “BEAT: Asynchronous BFT Made Practical”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 2028–2041. DOI: 10.1145/3243734.3243812 → 82.

[EEB16] Jacob Eberhardt, Dominik Ernst, and David Bermbach. *SMAC: State Management for Geo-Distributed Containers*. Tech. rep. Technische Universitaet Berlin, 2016 → 49.

[EG89] C. A. Ellis and S. J. Gibbs. “Concurrency Control in Groupware Systems”. In: *SIGMOD Rec.* 18.2 (June 1989), pp. 399–407. DOI: 10.1145/66926.66963 → 22.

[Ene+16] Vitor Enes, Carlos Baquero, Paulo Sérgio Almeida, and Ali 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*. PMLDC ’16. Rome, Italy: Association for Computing Machinery, 2016. DOI: 10.1145/2957319.2957374 → 24.

[Ene+19] Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero, and João Leitão. “Efficient Synchronization of State-based CRDTs”. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. Macao, China, Apr. 2019, pp. 148–159. DOI: 10.1109/ICDE.2019.00022 → 24.

[Fid88] Colin J Fidge. “Timestamps in message-passing systems that preserve the partial ordering”. In: *Proceedings of the 11th Australian Computer Science Conference*. ACSC ’88. 1988, pp. 56–66 → 7.

[FT16] Steve Fromhart and Lincy Therattil. *Making blockchain real for customer loyalty rewards programs*. Tech. rep. Deloitte, 2016. URL: https://www2.deloitte.com/content/dam/Deloitte/us/Documents/financial-services/us-fsi-making-blockchain-real-for-loyalty-rewards-programs.pdf → 12, 57.

[Gar+15] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. "Edge-Centric Computing: Vision and Challenges". In: *SIGCOMM Comput. Commun. Rev.* 45.5 (2015), pp. 37–42. DOI: 10.1145/2831347.2831354 → 55.

[Gil+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. "Algorand: Scaling Byzantine Agreements for Cryptocurrencies". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 51–68. DOI: 10.1145/3132747.3132757 → 82.

[Gon+17] 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)*. Hong Kong, China: IEEE, 2017, pp. 194–203. DOI: 10.1109/SRDS.2017.28 → 49.

[GPS16] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. "Trade-offs in Replicated Systems". In: *IEEE Data Engineering Bulletin* 39.1 (2016), pp. 14–26. URL: http://infoscience.epfl.ch/record/223701 → 10.

[Gue+19] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. "SBFT: a scalable and decentralized trust infrastructure". In: *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks*. DSN '19. Portland, OR, USA: IEEE, 2019, pp. 568–580. DOI: 10.1109/DSN.2019.00063 → 56, 75.

[Guo+20] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. "Dumbo: Faster Asynchronous BFT Protocols". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 803–818. DOI: 10.1145/3372297.3417262 → 82.

[Gup+20] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. "ResilientDB: Global Scale Resilient Blockchain Fabric". In: *Proc. VLDB Endow.* 13.6 (2020), pp. 868–883. DOI: 10.14778/3380750.3380757 → 81.

[Haa22] Julian Haas. "Programming Support for Local-First Software: Enabling the Design of Privacy-Preserving Distributed Software without Relying on the Cloud". In: *Companion Proceedings of*

the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. SPLASH Companion 2022. Auckland, New Zealand: Association for Computing Machinery, 2022, pp. 21–24. DOI: 10.1145/3563768.3565546 → 4.

[Hic12] Ian Hickson. *The WebSocket API, W3C Candidate Recommendation*. Tech. rep. 2012. URL: https://www.w3.org/TR/2012/CR-websockets-20120920/ → 32.

[HK20] Peter van Hardenberg and Martin Kleppmann. "PushPin: Towards Production-Quality Peer-to-Peer Collaboration". In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '20. Heraklion, Greece: Association for Computing Machinery, 2020. DOI: 10.1145/3380787.3393683 → 4, 20.

[ISV18] Zsolt István, Alessandro Sorniotti, and Marko Vukolić. "Stream-Chain: Do Blockchains Need Blocks?" In: *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. SERIAL '18. Rennes, France: Association for Computing Machinery, 2018, pp. 1–6. DOI: 10.1145/3284764.3284765 → 82.

[Jan+23a] Kristof Jannes, Emad Heydari Beni, Bert Lagaisse, and Wouter Joosen. "BeauForT: Robust Byzantine Fault Tolerance for Client-centric Mobile Web Applications". In: *IEEE Transactions on Parallel and Distributed Systems* 34.4 (2023), pp. 1241–1252. DOI: 10.1109/TPDS.2023.3241963 → 14, 54, 131.

[Jan+23b] Kristof Jannes, Vincent Reniers, Wouter Lenaerts, Bert Lagaisse, and Wouter Joosen. "DEDACS: Decentralized and dynamic access control for smart contracts in a policy-based manner". In: *Proceedings of the 38th Annual ACM Symposium on Applied Computing*. SAC '23. Tallinn, Estonia: Association for Computing Machinery, 2023. DOI: 10.1145/3555776.3577676 → 15, 132.

[Jan22] Kristof Jannes. "Secure and Resilient Data Replication for the Client-centric Decentralized Web". In: *Proceedings of the 23nd International Middleware Conference Doctoral Symposium*. Middleware '22 Doctoral Symposium. Quebec, Quebec City, Canada: Association for Computing Machinery, 2022, pp. 1–4. DOI: 10.1145/3569950.3569961 → 132.

[JLJ19a] Kristof Jannes, Bert Lagaisse, and Wouter Joosen. "The Web Browser as Distributed Application Server: Towards Decentralized Web Applications in the Edge". In: *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*.

EdgeSys '19. Dresden, Germany: Association for Computing Machinery, 2019, pp. 7–11. DOI: 10.1145/3301418.3313938 → 4, 5, 15, 55, 89, 131.

[JLJ19b] Kristof Jannes, Bert Lagaisse, and Wouter Joosen. "You Don't Need a Ledger: Lightweight Decentralized Consensus Between Mobile Web Clients". In: *Proceedings of the 3rd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. SERIAL '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 3–8. DOI: 10.1145/3366611.3368143 → 15, 55, 131.

[JLJ21] Kristof Jannes, Bert Lagaisse, and Wouter Joosen. "OWebSync: Seamless Synchronization of Distributed Web Clients". In: *IEEE Transactions on Parallel and Distributed Systems* 32.9 (2021), pp. 2338–2351. DOI: 10.1109/TPDS.2021.3066276 → 14, 18, 55, 63, 74, 80, 95, 97, 98, 131.

[JLJ22a] Kristof Jannes, Bert Lagaisse, and Wouter Joosen. "Seamless Synchronization for Collaborative Web Services". In: *Service-Oriented Computing – ICSOC 2021 Workshops*. Springer International Publishing, 2022, pp. 311–314. DOI: 10.1007/978-3-031-14135-5_27 → 14, 18, 98, 132.

[JLJ22b] Kristof Jannes, Bert Lagaisse, and Wouter Joosen. "Secure Replication for Client-centric Data Stores". In: *Proceedings of the 3nd International Workshop on Distributed Infrastructure for Common Good*. DICG '22. Quebec, Quebec City, Canada: Association for Computing Machinery, 2022, pp. 31–36. DOI: 10.1145/3565383.3566111 → 14, 81, 88, 132.

[Kak+19] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. "Mergeable Replicated Data Types". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360580 → 48.

[Kap+12] Rudiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schroder-Preikschat, and Klaus Stengel. "CheapBFT: Resource-Efficient Byzantine Fault Tolerance". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: Association for Computing Machinery, 2012, pp. 295–308. DOI: 10.1145/2168836.2168866 → 83.

[KB17] Martin Kleppmann and Alastair R. Beresford. "A Conflict-Free Replicated JSON Datatype". In: *IEEE Transactions on Parallel & Distributed Systems* 28.10 (2017), pp. 2733–2746. DOI: 10.1109/TPDS.2017.2697382 → 20, 38, 49, 55.

[KB18] Martin Kleppman and Alastair R Beresford. "Automerge: Real-time data sync between edge devices". In: *1st UK Mobile, Wearable and Ubiquitous Systems Research Symposium*. MobiUK '18. 2018. URL: http://martin.kleppmann.com/papers/automerge-mobiuk18.pdf → 7, 20, 49, 55, 80.

[KE10] Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. 2010. DOI: 10.17487/RFC5869 → 95.

[Kia+17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol". In: *Advances in Cryptology – CRYPTO 2017*. Springer International Publishing, 2017, pp. 357–388. DOI: 10.1007/978-3-319-63688-7_12 → 82.

[KK10] Santosh Kumawat and Ajay Khunteta. "A survey on operational transformation algorithms: Challenges, issues and achievements". In: *International Journal of Computer Applications* 3 (July 2010), pp. 30–38. DOI: 10.5120/787-1115 → 22.

[KKB19] Stephan A. Kollmann, Martin Kleppmann, and Alastair R. Beresford. "Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing". In: *Proceedings on Privacy Enhancing Technologies* 2019.3 (2019), pp. 210–232. DOI: 10.2478/popets-2019-0044 → 47, 100.

[Kle+19] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. "Local-First Software: You Own Your Data, in Spite of the Cloud". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 154–178. DOI: 10.1145/3359591.3359737 → 3, 4, 89.

[Kle+22] Martin Kleppmann, Dominic P. Mulligan, Victor F. Gomes, and Alastair R. Beresford. "A Highly-Available Move Operation for Replicated Trees". In: *IEEE Transactions on Parallel & Distributed Systems* 33.7 (2022). DOI: 10.1109/TPDS.2021.3118603 → 55.

[Kle22] Martin Kleppmann. "Making CRDTs Byzantine Fault Tolerant". In: *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '22. Rennes, France: Association for Computing Machinery, 2022, pp. 8–15. DOI: 10.1145/3517209.3524042 → 81, 90, 94, 100.

[Koc+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. San Francisco, CA, USA: IEEE, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002 → 83.

[Kok+16] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. "Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing". In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC '16. Austin, TX, USA: USENIX Association, 2016, pp. 279–296. DOI: 10.5555/3241094.3241117 → 83.

[Kot+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. "Zyzzyva: Speculative Byzantine Fault Tolerance". In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: Association for Computing Machinery, 2007, pp. 45–58. DOI: 10.1145/1294261.1294267 → 81.

[Lin+17] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. "Legion: Enriching Internet Services with Peer-to-Peer Interactions". In: *Proceedings of the 26th International Conference on World Wide Web*. WWW '17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 283–292. DOI: 10.1145/3038912.3052673 → 7, 20, 38, 49, 55, 80.

[Lin+19] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gun Sirer, and Peter Pietzuch. "Teechain: A Secure Payment Network with Asynchronous Blockchain Access". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 63–79. DOI: 10.1145/3341301.3359627 → 83.

[Lip+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium*. USENIX Security '18. USENIX Association, 2018, pp. 973–990 → 83.

[Liu+18] Jian Liu, Wenting Li, Ghassan O Karame, and N Asokan. "Scalable byzantine consensus via hardware-assisted secret sharing". In: *IEEE Transactions on Computers* 68.1 (2018), pp. 139–151. DOI: 10.1109/TC.2018.2860009 → 83.

[Llo+13] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. "Stronger Semantics for Low-Latency Geo-Replicated Storage". In: *10th USENIX Symposium on Networked Systems Design and Implementation*. NSDI '13. USENIX Association, 2013, pp. 313–328. DOI: 10.5555/2482626.2482657 → 10.

[LLP16] Albert van der Linde, João Leitão, and Nuno Preguiça. "Δ-CRDTs: Making δ-CRDTs Delta-based". In: *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data*. PaPoC '16. London, United Kingdom: Association for Computing Machinery, 2016. DOI: 10.1145/2911151.2911163 → 7, 20, 24, 38.

[LLP20] Albert van der Linde, João Leitão, and Nuno Preguiça. "Practical Client-Side Replication: Weak Consistency Semantics for Insecure Settings". In: *Proc. VLDB Endow.* 13.12 (2020), pp. 2590–2605. DOI: 10.14778/3407790.3407847 → 81, 100.

[LM10] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40. DOI: 10.1145/1773912.1773922 → 48.

[LSM05] Paul J. Leach, Rich Salz, and Michael H. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. July 2005. DOI: 10.17487/RFC4122 → 33.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401. DOI: 10.1145/357172.357176 → 7.

[Lua+05] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. "A survey and comparison of peer-to-peer overlay network schemes". In: *IEEE Communications Surveys & Tutorials* 7.2 (2005), pp. 72–93. DOI: 10.1109/COMST.2005.1610546 → 2.

[Mad+19] Akash Madhusudan, Iraklis Symeonidis, Mustafa A. Mustafa, Ren Zhang, and Bart Preneel. "SC2Share: Smart Contract for Secure Car Sharing". In: *Proceedings of the 5th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP*. INSTICC. SciTePress, 2019. DOI: 10.5220/0007703601630171 → 55.

[Man+16] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. "A Demonstration of the Solid Platform for Social Web Applications". In: *Proceedings of the 25th International Conference Companion on World Wide Web*. WWW '16

Companion. Montréal, Québec, Canada: WWW, 2016, pp. 223–226. DOI: 10.1145/2872518.2890529 → 5, 89.

[Mat88] Friedemann Mattern. "Virtual time and global states of distributed systems". In: *Proceedings of the International Workshop on Parallel and DistributedAlgorithms*. Elsevier, 1988 → 7.

[Maz15] David Mazieres. *The stellar consensus protocol: A federated model for internet-level consensus*. Tech. rep. Stellar Development Foundation, 2015 → 82.

[McC+19] Patrick McCorry, Surya Bakshi, Iddo Bentov, Sarah Meiklejohn, and Andrew Miller. "Pisa: Arbitration Outsourcing for State Channels". In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. AFT '19. Zurich, Switzerland: Association for Computing Machinery, 2019, pp. 16–30. DOI: 10.1145/3318041.3355461 → 83.

[McC+20] Patrick McCorry, Chris Buckland, Surya Bakshi, Karl Wust, and Andrew Miller. "You Sank My Battleship! A Case Study to Evaluate State Channels as a Scaling Solution for Cryptocurrencies". In: *Financial Cryptography and Data Security*. Springer International Publishing, 2020, pp. 35–49. DOI: 10.1007/978-3-030-43725-1_4 → 83.

[Mer88] Ralf Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology*. CRYPTO '87. Springer Berlin Heidelberg, 1988, pp. 369–378. DOI: 10.1007/3-540-48184-2_32 → 20, 25, 63, 95.

[Mil+16] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. "The Honey Badger of BFT Protocols". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 31–42. DOI: 10.1145/2976749.2978399 → 82.

[Mil+19] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. "Sprites and State Channels: Payment Networks that Go Faster Than Lightning". In: *Financial Cryptography and Data Security*. Springer International Publishing, 2019, pp. 508–526. DOI: 10.1007/978-3-030-32101-7_30 → 83.

[Mos+12] L. E. Moser, H. Zhang, W. Zhao, P. Melliar-Smith, and H. Chai. "Trustworthy Coordination of Web Services Atomic Transactions". In: *IEEE Transactions on Parallel and Distributed Systems* 23.08 (2012), pp. 1551–1565. DOI: 10.1109/TPDS.2011.292 → 81.

[MVR99] Silvio Micali, Salil Vadhan, and Michael Rabin. "Verifiable random functions". In: *40th Annual Symposium on Foundations of Computer Science*. FOCS '99. IEEE, 1999, pp. 120–130. DOI: 10.1109/SFFCS.1999.814584 → 82.

[Nak08] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008 → 3, 6, 55, 63, 74, 83.

[Nic+15] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. "Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types". In: *Engineering the Web in the Big Data Era*. Springer International Publishing, 2015, pp. 675–678. DOI: 10.1007/978-3-319-19890-3_55 → 7, 20, 50, 80.

[Nic+16] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. "Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types". In: *Proceedings of the 19th International Conference on Supporting Group Work*. GROUP '16. Sanibel Island, Florida, USA: Association for Computing Machinery, 2016, pp. 39–49. DOI: 10.1145/2957276.2957310 → 20, 50.

[Nie10] Jakob Nielsen. *Website Response Times*. 2010. URL: https://www.nngroup.com/articles/website-response-times/ → 42.

[Nie93] Jakob Nielsen. *Usability Engineering*. Nielsen Norman Group, 1993. ISBN: 978-0125184069. URL: https://www.nngroup.com/books/usability-engineering/ → 9, 19, 42.

[NMJ19] Pezhman Nasirifard, Ruben Mayer, and Hans-Arno Jacobsen. "FabricCRDT: A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains". In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 110–122. DOI: 10.1145/3361525.3361540 → 82.

[OM14] Karl J O'Dwyer and David Malone. "Bitcoin mining and its energy footprint". In: *Proceedings of the 2014 IET Irish Signals and Systems Conference*. ISSC 2014/CIICT 2014. IET, 2014, pp. 280–285. DOI: 10.1049/cp.2014.0699 → 83.

[Ope19] OpenSignal. *Mobile Network Experience Report*. https://www.opensignal.com/reports/2019/01/usa/mobile-network-experience. 2019 → 38.

[Ora01] Andy Oram. *Peer-to-Peer: Harnessing the power of disruptive technologies*. O'Reilly Media, Inc., 2001 → 2.

[PR85] Jon Postel and Joyce Reynolds. *File Transfer Protocol*. RFC 959. Oct. 1985. DOI: 10.17487/RFC0959 → 2.

[Pre+09] 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*. Montreal, QC, Canada, June 2009, pp. 395–403. DOI: 10.1109/ICDCS.2009.20 → 20.

[Pre+14] Nuno Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc 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. DOI: 10.1109/SRDSW.2014.33 → 20, 49.

[Ram+09] Venugopalan Ramasubramanian, Thomas L Rodeheffer, Douglas B Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C Marshall, and Amin Vahdat. “Cimbiosys: A platform for content-based partial replication”. In: *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. 2009, pp. 261–276 → 49.

[Riv92] Ronald L. Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321. Apr. 1992. DOI: 10.17487/RFC1321 → 33.

[Roc+19] Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gun Sirer. *Scalable and Probabilistic Leaderless BFT Consensus through Metastability*. 2019. arXiv: 1906.08936 → 55, 82.

[Roc18] Team Rocket. *Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies*. White paper. avalabs.org, 2018 → 82.

[Roh+11] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. “Replicated abstract data types: Building blocks for collaborative applications”. In: *Journal of Parallel and Distributed Computing* 71.3 (2011), pp. 354–368. DOI: 10.1016/j.jpdc.2010.12.006 → 32.

[Sau+21a] Martijn Sauwens, Emad Heydari Beni, Kristof Jannes, Bert Lagaisse, and Wouter Joosen. “ThunQ: A Distributed and Deep Authorization Middleware for Early and Lazy Policy Enforcement in Microservice Applications”. In: *Service-Oriented Computing*. Springer International Publishing, 2021, pp. 204–220. DOI: 10.1007/978-3-030-91431-8_13 → 15, 132.

[Sau+21b] Martijn Sauwens, Kristof Jannes, Bert Lagaisse, and Wouter Joosen. “SCEW: Programmable BFT-Consensus with Smart Contracts for Client-Centric P2P Web Applications”. In: *Proceedings of the 8th Workshop on Principles and Practice of Consistency for*

Distributed Data. PaPoC ’21. Online, United Kingdom: Association for Computing Machinery, 2021. DOI: 10.1145/3447865.3457965 → 15, 54, 131.

[SBV18] Joao Sousa, Alysson Bessani, and Marko Vukolic. “A byzantine fault-tolerant ordering service for the Hyperledger Fabric blockchain platform”. In: *48th annual IEEE/IFIP international conference on dependable systems and networks*. DSN ’18. Luxembourg, Luxembourg: IEEE, 2018, pp. 51–58. DOI: 10.1109/DSN.2018.00018 → 77, 81.

[Sha+11a] Marc Shapiro, Nuno Perguiça, Carlos Baquero, and Marek Zawirski. “Conflict-Free Replicated Data Types”. In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS ’11. Springer Berlin Heidelberg, 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29 → 7, 10, 22, 23, 28, 63, 81, 90, 92, 94.

[Sha+11b] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: https://hal.inria.fr/inria-00555588 → 6, 20, 22, 24, 25, 28, 32, 106.

[Sha17] Marc Shapiro. “Replicated Data Types”. In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Ozsu. Springer New York, 2017, pp. 1–5. DOI: 10.1007/978-1-4899-7993-3_80813-1 → 24, 25, 29.

[Sho00] Victor Shoup. “Practical threshold signatures”. In: *Advances in Cryptology*. EUROCRYPT 2000. Springer Berlin Heidelberg, 2000, pp. 207–220. DOI: 10.1007/3-540-45539-6_15 → 82.

[Sil+21] Douglas Simões Silva, Rafal Graczyk, Jérémie Decouchant, Marcus Volp, and Paulo Esteves-Verissimo. “Threat Adaptive Byzantine Fault Tolerant State-Machine Replication”. In: *40th International Symposium on Reliable Distributed Systems*. SRDS ’21. IEEE, 2021, pp. 78–87. DOI: 10.1109/SRDS53918.2021.00017 → 82.

[SK92] Mukesh Singhal and Ajay Kshemkalyani. “An efficient implementation of vector clocks”. In: *Information Processing Letters* 43.1 (1992), pp. 47–52. DOI: 10.1016/0020-0190(92)90028-T → 7.

[SW05] Amit Sahai and Brent Waters. “Fuzzy Identity-Based Encryption”. In: *Advances in Cryptology – EUROCRYPT 2005*. Springer Berlin Heidelberg, 2005, pp. 457–473. DOI: 10.1007/11426639_27 → 101.

[SYB+14] David Schwartz, Noah Youngs, Arthur Britto, et al. *The ripple protocol consensus algorithm*. Tech. rep. Ripple Labs Inc, 2014 → 82.

[Syt+16] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. “Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning”. In: *2016 IEEE Symposium on Security and Privacy*. S&P ’16. San Jose, CA, USA: IEEE, 2016, pp. 526–545. DOI: 10.1109/SP.2016.38 → 83.

[Tar+19] Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. “Secure Scuttlebutt: An Identity-Centric Protocol for Subjective and Decentralized Applications”. In: *Proceedings of the 6th ACM Conference on Information-Centric Networking*. ICN ’19. Macao, China: Association for Computing Machinery, 2019, pp. 1–11. DOI: 10.1145/3357150.3357396 → 100.

[Vaq+09] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. “A Break in the Clouds: Towards a Cloud Definition”. In: *SIGCOMM Comput. Commun. Rev.* 39.1 (Dec. 2009), pp. 5–55. DOI: 10.1145/1496091.1496100 → 2.

[Ver+13] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. “Efficient byzantine fault-tolerance”. In: *IEEE Transactions on Computers* 62.1 (2013), pp. 16–30. DOI: 10.1109/TC.2011.221 → 83.

[Vri+22] Pieter-Jan Vrielynck, Emad Heydari Beni, Kristof Jannes, Bert Lagaisse, and Wouter Joosen. “DeFIRED: Decentralized Authorization with Receiver-Revocable and Refutable Delegations”. In: *Proceedings of the 15th European Workshop on Systems Security*. EuroSec ’22. Rennes, France: Association for Computing Machinery, 2022, pp. 57–63. DOI: 10.1145/3517208.3523759 → 15, 132.

[Wei+22] Matthew Weidner, Heather Miller, Huairui Qi, Maxime Kjaer, Ria Pradeep, Benito Geordie, and Christopher Meiklejohn. *Collabs: Composable Collaborative Data Structures*. 2022. arXiv: 2212.02618 → 18.

[Woo14] Gavin Wood. *Ethereum: a secure decentralized generalized transaction ledger*. Yellow paper. ethereum.org, 2014 → 95.

[Yin+19] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. “HotStuff: BFT Consensus with Linearity and Responsiveness”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. PODC ’19. Toronto ON, Canada:

Association for Computing Machinery, 2019, pp. 347–356. DOI: 10.1145/3293611.3331591 → 8, 55, 56, 75, 81.

[Zaw+13] Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro, and Nuno Preguiça. *SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine*. Research Report RR-8347. INRIA, Oct. 2013. URL: https://hal.inria.fr/hal-00870225 → 20, 49.

[Zaw+15] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. "Write Fast, Read in the Past: Causal Consistency for Client-Side Applications". In: *Proceedings of the 16th Annual Middleware Conference*. Middleware '15. Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 75–87. DOI: 10.1145/2814576.2814733 → 49.

[Zha+17] Fan Zhang, Ittay Eyal, Robert Escriva, Ari Juels, and Robbert Van Renesse. "REM: Resource-Efficient Mining for Blockchains". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC '17. Vancouver, BC, Canada: USENIX, 2017, pp. 1427–1444. DOI: 10.5555/3241189.3241300 → 83.

List of publications

Journal papers

Kristof Jannes, Bert Lagaisse, and Wouter Joosen. "OWebSync: Seamless Synchronization of Distributed Web Clients". In: *IEEE Transactions on Parallel and Distributed Systems* 32.9 (2021), pp. 2338–2351. DOI: 10.1109/TPDS.2021.3066276

Kristof Jannes, Emad Heydari Beni, Bert Lagaisse, and Wouter Joosen. "BeauForT: Robust Byzantine Fault Tolerance for Client-centric Mobile Web Applications". In: *IEEE Transactions on Parallel and Distributed Systems* 34.4 (2023), pp. 1241–1252. DOI: 10.1109/TPDS.2023.3241963

International conference and workshop papers

Kristof Jannes, Bert Lagaisse, and Wouter Joosen. "The Web Browser as Distributed Application Server: Towards Decentralized Web Applications in the Edge". In: *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*. EdgeSys '19. Dresden, Germany: Association for Computing Machinery, 2019, pp. 7–11. DOI: 10.1145/3301418.3313938

Kristof Jannes, Bert Lagaisse, and Wouter Joosen. "You Don't Need a Ledger: Lightweight Decentralized Consensus Between Mobile Web Clients". In: *Proceedings of the 3rd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. SERIAL '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 3–8. DOI: 10.1145/3366611.3368143

Martijn Sauwens, Kristof Jannes, Bert Lagaisse, and Wouter Joosen. "SCEW: Programmable BFT-Consensus with Smart Contracts for Client-Centric P2P Web Applications". In: *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '21. Online, United Kingdom: Association for Computing Machinery, 2021. DOI: 10.1145/3447865.3457965

Martijn Sauwens, Emad Heydari Beni, Kristof Jannes, Bert Lagaisse, and Wouter Joosen. “ThunQ: A Distributed and Deep Authorization Middleware for Early and Lazy Policy Enforcement in Microservice Applications”. In: *Service-Oriented Computing*. Springer International Publishing, 2021, pp. 204–220. DOI: 10.1007/978-3-030-91431-8_13

Pieter-Jan Vrielynck, Emad Heydari Beni, Kristof Jannes, Bert Lagaisse, and Wouter Joosen. “DeFIRED: Decentralized Authorization with Receiver-Revocable and Refutable Delegations”. In: *Proceedings of the 15th European Workshop on Systems Security*. EuroSec ’22. Rennes, France: Association for Computing Machinery, 2022, pp. 57–63. DOI: 10.1145/3517208.3523759

Kristof Jannes, Bert Lagaisse, and Wouter Joosen. “Secure Replication for Client-centric Data Stores”. In: *Proceedings of the 3nd International Workshop on Distributed Infrastructure for Common Good*. DICG ’22. Quebec, Quebec City, Canada: Association for Computing Machinery, 2022, pp. 31–36. DOI: 10.1145/3565383.3566111

Kristof Jannes, Vincent Reniers, Wouter Lenaerts, Bert Lagaisse, and Wouter Joosen. “DEDACS: Decentralized and dynamic access control for smart contracts in a policy-based manner”. In: *Proceedings of the 38th Annual ACM Symposium on Applied Computing*. SAC ’23. Tallinn, Estonia: Association for Computing Machinery, 2023. DOI: 10.1145/3555776.3577676

International presentations and demonstrations

Kristof Jannes, Bert Lagaisse, and Wouter Joosen. “Seamless Synchronization for Collaborative Web Services”. In: *Service-Oriented Computing – ICSOC 2021 Workshops*. Springer International Publishing, 2022, pp. 311–314. DOI: 10.1007/978-3-031-14135-5_27

Kristof Jannes. “Secure and Resilient Data Replication for the Client-centric Decentralized Web”. In: *Proceedings of the 23nd International Middleware Conference Doctoral Symposium*. Middleware ’22 Doctoral Symposium. Quebec, Quebec City, Canada: Association for Computing Machinery, 2022, pp. 1–4. DOI: 10.1145/3569950.3569961

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMEC-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Leuven
kristof.jannes@kuleuven.be
https://distrinet.cs.kuleuven.be/