WebLedger: a Client-centric Web-based BFT Ledger for Decentralized and Resilient Community Apps

Kristof Jannes, Emad Heydari Beni, Wouter Joosen and Bert Lagaisse

Abstract—One of the visions of Tim Berners-Lee, the founder of the web, is that the web should shift to a client-centric, decentralized model where web clients become the leading execution environment for application logic and data storage. Both Gartner and the Web3 foundation consider client-centric decentralization as one of the key properties of Web 3.0.
However, existing peer-to-peer web middleware only support operation in a fully trusted client network. Other decentralized solutions often use a heavyweight blockchain platform in the backend. Moreover, traditional Byzantine consensus protocols are not well-suited for a decentralized client-centric web with many network disruptions or node failures.
In this paper, we present WebLedger, a browser-based middleware for decentralized applications in small, community-driven networks. We propose a novel, optimistic, leaderless consensus protocol, tolerating Byzantine replicas, combined with a robust and efficient state-based synchronization protocol. This state-based protocol with authenticated data structures makes WebLedger resilient against network failures, and does not require that all replicas are directly connected to each other. WebLedger uses an optimized implementation of the standard BLS scheme for efficient aggregation and storage of signatures. No large backend infrastructure is required, as the middleware is purely browser-based. Using a state-based protocol, no transaction log or blockchain is stored, keeping the overall storage footprint small for client-centric devices.
Our performance evaluation shows that WebLedger can achieve finality of transactions within seconds in community-driven networks of mobile web clients, even in the context of network problems, node failures, and Byzantine behavior.

Index Terms—Peer-to-peer, Byzantine fault tolerance, Web Applications

1 Introduction

Web 3.0 can be defined as the decentralized web where users are in control of their data [1], and that replaces centralized intermediaries with decentralized networks and platforms [2], [3], [4]. Community-driven, decentralized networks can open the road to many use cases for the sharing economy or shared loyalty programs for local communities [5]. 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. WebLedger can also serve as a framework to explore many other collaborative use cases, that were previously not possible for the average person to set up. 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 unilateral at any moment.

In the last decade, decentralized interaction between distrusting parties has gained a lot of attention, starting with the Bitcoin blockchain. Bitcoin uses Nakamoto consensus [6] to solve the double-spending problem in a peer-to-peer electronic cash system, and uses Proof-of-Work (PoW) [7] for Sybil control. Another PoW blockchain is Ethereum [8], which allows everyone to run arbitrary application code on the blockchain. Unfortunately, these PoW blockchains 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. Next to the computationally expensive Nakamoto consensus protocol, classical Byzantine fault tolerant (BFT) consensus protocols can be used such as PBFT [9], BFT-SMaRt [10], Tendermint [11], Algorand [12], Ouroboros [13], or HotStuff [14]. These protocols are much faster than Nakamoto consensus, but they do not scale well with an increasing number of participants. At last, Avalanche consensus [15] tries to solve the scalability problem by using the concept of meta-stability.

Unfortunately, all these blockchain frameworks and consensus protocols are designed for a rather heavy-weight infrastructure that has lots of CPU or GPU power, storage space, and a 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 blockchain network between them. They want to use their existing hardware such as a low-end computer, or even a mobile device. Their internet connection is often only a domestic cable connection, unstable WiFi, or a slow 4G connection which brings higher latency and packet loss. They could make use of a public blockchain network, at the cost of paying a fee for every transaction. This transaction fee lowers the economic viability of this approach. A private network between the citizens without fees is more suitable. This also has the advantage that not all data is publicly readable by the whole world.

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

Another trend is Progressive Web Apps [16], which focus on client-centric web application architectures with native-app features. Browsers and client-side web technology offer more and more 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 [17], [18]. The web browser might be able to function as a ubiquitous substrate to build the envisioned collaborative, decentralized applications without the need for a heavy-weight blockchain. However, current state-of-the-art peer-to-peer data synchronization frameworks for the browser such as Legion [19], Yjs [20], Automerge [21], and OWeb-Sync [22] focus on full replication and 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.

In this paper, we present WebLedger, a peer-to-peer data synchronization framework for decentralized web applications between mistrusting parties. WebLedger combines the efficient operation and lightweight setup of a peer-to-peer data synchronization framework with the resilience of a blockchain in a Byzantine environment. WebLedger does not keep track of an operation log or transaction history, keeping the storage footprint small. The ledger is fully maintained and agreed upon by browser-clients. The replicas do not need to be connected to every other replica directly, as the authenticated state can be replicated over multiple hops. WebLedger builds upon the following technical contributions:

- Lightweight, leaderless, client-side Byzantine fault tolerant synchronization and consensus.
- Robust, state-based synchronization of both the data and the votes for the consensus protocol using state-based CRDTs and Merkle-trees.
- Efficient computation and compact storage of signatures using the BLS signature scheme.

Our evaluation, using our application use case of a shared loyalty program between small-scale merchants, shows that WebLedger is a practical solution for these kinds of community-driven use cases. WebLedger achieves transaction finality in the order of seconds, even in networks with 100 clients, or in unstable network conditions. No complex infrastructure is required, the participating merchants only need a browser and an internet connection.

Section 2 further discusses some motivating use cases and background in more depth. Section 3 presents WebLedger's lightweight BFT consensus protocol and the state-based replication strategy. The detailed web-based middleware architecture of WebLedger is elaborated in Section 4. Our evaluation in Section 5 focuses on many aspects of performance in both the optimistic scenario as well as more realistic and even Byzantine scenarios. Section 6 elaborates on important related work. We conclude in Section 7.

2 Motivation and background

This section further motivates the need for a lightweight, robust consensus middleware by describing several community-driven use cases. Then we give some background on state-of-the-art approaches using a blockchain and BFT consensus.

2.1 Motivational use cases

We describe three initial use cases that would benefit from the lightweight consensus offered by WebLedger. They all involve business transactions happening in real life and need interactive performance, rather than high throughput.

Sharing economy. Small communities, such as an apartment building or local neighborhood, can share tools or cars [23] with each other using a P2P platform to keep track of the current possession and reservation of tools and cars [24]. When a tool is being exchanged, it is checked on potential damage which can be registered in the network.

Loyalty programs. Integrated loyalty programs can be more effective than traditional loyalty programs that are limited to a single company [25]. 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 P2P network can enable fast and secure creation, redemption, and exchange of loyalty points across the different merchants.

Microloans. Microloans enable individuals, rather than banks, to issue loans to other individuals or small businesses. This has the advantage that also individuals with a bad credit rating or without enough collateral can receive a loan. This community initiative can prevent loan sharks, especially in developing countries.

Vision. We envision that communities will be able to use WebLedger 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 paper can be easily ported towards native mobile and lightweight desktop applications. WebLedger does not need any complex infrastructure, and it currently provides a simple JavaScript-based API, which allows many developers to start developing decentralized applications. Extensions of WebLedger, such as SCEW [26], a smart-contract abstraction on top of WebLedger, can make the barrier even lower for developers. 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 open-source application and do not need to concern themselves with a complex infrastructure set up to run the application.

2.2 Background on blockchains and BFT consensus

Existing blockchains can be roughly split into two categories: public and permissioned blockchains. Public blockchains are open for everyone to participate in. Two examples are Bitcoin [6] and Ethereum [8]. Bitcoin allows everyone to host a replica node and submit transactions. However, Bitcoin is quite slow, as a new block is only created every 10 minutes on average. This means that transactions take on average 10 minutes to be confirmed by the network. But as multiple conflicting chains can occur, one must wait for at least 6 blocks to be sure that a transaction

will not be reverted. This increases the total latency to one hour, which is too slow for many of the motivational use cases. Ethereum is another public blockchain with a much faster average block time, and consequently a lower latency. Ethereum allows everyone to write *smart-contracts* to be executed by the Ethereum network. Each invocation of a contract costs a small amount of Ether (called gas). This makes Ethereum infeasible for small business transactions such as loyalty points, as the total cost will become too high.

Permissioned or private blockchains use access control to limit who can see and create transactions on the blockchain. Because they can only be accessed by a limited number of known parties, transaction fees are not required to reward miners and combat spam. An example is Hyperledger Fabric [27]. These private blockchains can use a Byzantine fault tolerant consensus protocol to reach consensus over which transactions to execute and in which order. They have much smaller latency and can process more transactions per second compared to the public blockchains. However, to set up Hyperledger Fabric in a decentralized fashion, there is a large back-end infrastructure required. The actual blockchain network consists of many nodes: peers, orderers, REST-API servers, database servers, and a certificate authority. Setting up and managing these services requires a lot of infrastructural management for small merchants. They do not have the knowledge nor budget for such a deployment, especially considering the maintenance overhead and resource costs. These small merchants want to quickly set up an integrated loyalty network with minimal back-end setup. However, most of them already own a desktop or a mobile computer such as a laptop or tablet.

Two existing state-of-the-art protocols for BFT consensus are BFT-SMaRt [10] and Tendermint [11], [28]. BFT-SMaRt is a more traditional BFT protocol, similar to PBFT [29], 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. BFT-SMaRt can be used in Hyperledger Fabric [30]. Tendermint [28] uses Gossip for communication between the replicas. There is still a leader, however, that leader changes frequently. Tendermint is used in the Cosmos blockchain [31].

3 Optimistic state-based BFT consensus

This section explains the state-based consensus protocol used in WebLedger. First, it describes the adversary model and its properties. Then it explains the protocol specification. At last, this section provides safety and liveness proofs.

3.1 Overview and adversary model

The core protocol is an asynchronous, leaderless, Byzantine fault tolerant consensus protocol. In an asynchronous network, messages are eventually delivered, but no timing assumption is made [32]. An adversary might also 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. We assume attackers cannot forge the used asymmetric signatures or find collisions for the used cryptographic hash functions.

The protocol is used to implement an Atomic Register [33] that can hold a single value that can be read and written by multiple replicas. All writes are atomic, meaning that only a single state transition can happen at any time. Extra conditions can be applied to limit who can write to it, and which values are acceptable.

The protocol 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. The consensus protocol presented here uses voting, where every replica has exactly one vote. The set of replicas is fixed, and changes to the replica set have to be made outside the protocol. Unlike contemporary blockchains, consensus is reached for each register separately, and there is no chain of transactions. Only the current state, its commit certificate, and proposals for the next state are stored.

Formal properties: Let $\mathfrak{R}$ be a cluster of n replicas with f Byzantine nodes and with $n \geq 3f + 1$. WebLedger guarantees the following properties:

- **Non-divergence:** If replicas $R_1, R_2 \in \mathfrak{R}$ are able to construct commit certificates c_1 for value v_1 and c_2 for value v_2 at version V, then $v_1 = v_2$.
- **Termination:** If an honest replica $R \in \mathfrak{R}$ proposes a new value v at version V, eventually a replica will be able to construct a valid commit certificate c for some value at version V.

The first property is a safety property and guarantees that all state changes are atomic for the whole network. The second property is a liveness property and guarantees that non-conflicting transactions will be eventually executed by all replicas.

3.2 Protocol specification

Each atomic register has its own state which consists of three parts. The first part is the current value and a commit certificate that proves the validity of the value. It contains signatures of a supermajority of $n - f$ replicas. The second part is an array of rounds that contains proposals for new values. In each round, there can be multiple proposed values. The third part consists of a new proposed value and a partial commit certificate for that value. This state is shown at the top of Fig. 2. The atomic register is implemented as a register CRDT with a `GET`, `SET` and `MERGE` operation.

Consensus is reached in two steps, first a supermajority needs to be reached in the last round of the proposals, then a supermajority needs to be reached for the proposed commit 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.

3.2.1 State-base replication protocol

The current value, proposals, and commit certificates are replicated by using a state-based Gossip protocol. This protocol is a peer-to-peer version of OWebSync [22], which uses state-based Conflict-free Replicated Data Types (CRDT) [34] combined with a Merkle-tree [35] to efficiently replicate the updated state. If the state of two replicas is the same, only

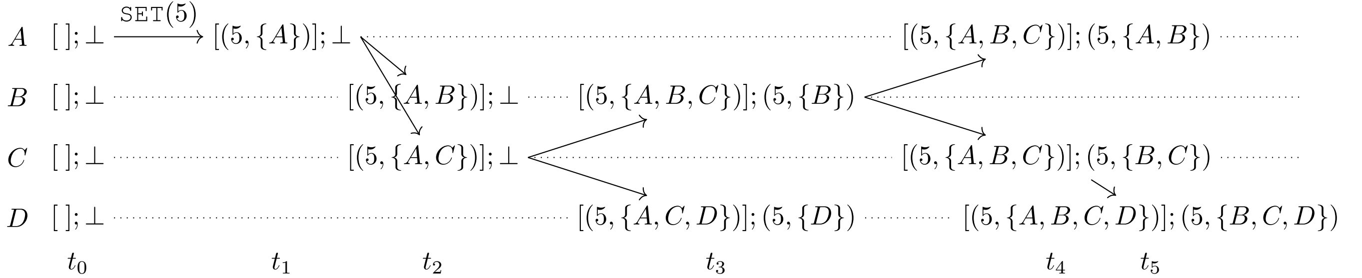

Fig. 1. State-based synchronization of an Atomic Register with 4 replicas $A, B, C, D \in replicaId$. Only the current $proposals$ and $(proposedValue, proposedCertificate)$ are shown for brevity. $Version$ and $round$ of the $proposedCertificate$ are not shown as they stay always the same in this example.

the root hash is sent and compared, which limits the network usage. If the states differ, the protocol descends in the tree looking for the mismatching hashes to find out which registers must be synchronized. By using a state-based approach, rather than the operation-based approach of Operational Transformation [36], operation-based CRDTs [34], or blockchains [6], 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 [37].

The replicas execute a Gossip protocol to exchange their current state with each other. Each time a new state is received, the local state is merged with the remote state. An example of this process is shown in Fig. 1. There are four non-Byzantine replicas with an empty set of proposals. Each proposal lists the value and the set of signatures of the replicas that voted for that proposal. The scenario starts with replica A proposing a new value. The state is replicated to the other replicas randomly, and all replicas collect the votes in the set of signatures.

3.2.2 *Reading and writing*

The GET operation will return the currently accepted value. This request is always executed on the local replica and does not involve any network requests.

The SET operation will propose a new value to the other replicas. This value is not immediately visible via the GET operation, first consensus will have to be reached by at least a supermajority of the replicas. The replica that wants to set a new value, can add the value to round 0 with his vote. Replicas are only allowed to vote once in each round for each version, so if the replica already voted for another value in that round, it will have to wait until consensus is reached for the current set of proposals, and propose the new value for the version after it.

3.2.3 *Consensus*

The MERGE will combine the state of the same register from two different replicas. The detailed specification of MERGE is depicted in Fig. 2. This MERGE operation gets as input the state of another replica and advances the current local state. The new value will be the value associated with the highest version number. Since each accepted value is always accompanied by a commit certificate which is signed by a supermajority of the replicas, it can be accepted without the need to verify intermediate versions. Proposals that belong to a smaller or equal version than the currently accepted value can be discarded. The new proposals are the union between the local and received proposals at each round of the proposals. However, instead of simply taking the union, a replica will first verify if the newly received state is actually valid. For example, a new round can only be started when a supermajority of the replicas voted for the previous round, but no single value reached a supermajority.

Honest replicas will always vote for the value with the most votes in round 0. 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 proposed commit certificate. If a supermajority of the replicas have voted, but not a single value reached a supermajority, a new round is started and all replicas can vote again in this new round. To ensure rapid convergence, the replicas will vote on the current winner in round 0. This selection procedure is shown in Fig. 3. Because each replica might have different views 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, eventually the view of all the replicas on the votes in round 0 will become the same, and the winning value can be chosen unanimously. We will prove the correctness of this in Section 3.3.

Once a replica observes that a supermajority of the replicas supports a single value, it starts working on a proposed certificate to determine if at least a supermajority of the replicas also knows about this. In the example in Fig. 1, at t_3 both replica B and replica D observe a supermajority for value 5, and they start creating a new proposed commit certificate. At t_5, replica D has a proposed commit certificate signed by a supermajority of the replicas. This means that the new value 5 can be committed. The proposed commit certificate becomes the new commit certificate and the proposals are removed. When another replica now receives the state of replica D, that replica will notice that it has a value associated with a valid commit certificate with a larger version number as his own. Therefore it will accept this new value and remove all of its own proposals and proposed certificate if any.

3.2.4 *Optimistic fast path*

For brevity, we did not show the actual verification of signatures in Fig.2. However, in the basic protocol, each

```
1:  initial state
2:      REPLICA_ID
3:      QUORUM                                                        ▷ QUORUM ≡ n − f
4:      value ← ⊥
5:      certificate ← ⊥           ▷ (version ∈ Integers, round ∈ Integers, votes ∈ P({REPLICA_ID}))
6:      proposals ← [ ]                              ▷ round ↦ (value ↦ P({REPLICA_ID}))
7:      proposedValue ← ⊥
8:      proposedCertificate ← ⊥   ▷ (version ∈ Integers, round ∈ Integers, votes ∈ P({REPLICA_ID}))
9:      byzReplicaIds ← ∅                                             ▷ P({REPLICA_ID})
10: procedure MERGE(remote)
11:     if remote.certificate.version > certificate.version then   ▷ accept newer committed values
12:         value ← remote.value; certificate ← remote.certificate
13:         proposals ← [ ]; proposedValue ← ⊥; proposedCertificate ← ⊥
14:     for ∀ round_i ∈ remote.proposals do
15:         if round_i > 0 ∧ round_i ∉ proposals then                ▷ detect start of illegal rounds
16:             if ∑_v SIZE(remote.proposals[round_i − 1][v]) < QUORUM
                  ∨ ∃ v ∈ remote.proposals[round_i − 1] : SIZE(remote.proposals[round_i − 1][v] \ byzReplicaIds) ≥ QUORUM then
17:                 byzReplicaIds ← byzReplicaIds ∪ {remote.REPLICA_ID}
18:                 break
19:         proposals[round_i] ← proposals[round_i] ⋁ remote.proposals[round_i]      ▷ merge proposals
20:         if ∃ id : id ∈ proposals[round_i][v_1] ∧ id ∈ proposals[round_i][v_2] ∧ v_1 ≠ v_2 then   ▷ detect duplicate votes
21:             byzReplicaIds ← byzReplicaIds ∪ {id}
22:         for ∀ v ∈ proposals[round_i] \ GET_POSSIBLE_WINNING_VALUES(proposals[0..round_i]) do   ▷ detect illegal votes
23:             byzReplicaIds ← byzReplicaIds ∪ proposals[round_i][v]
24:         if ¬HAS_VOTED(proposals[round_i]) then                  ▷ cast vote for round proposal
25:             v ← GET_WINNING_VALUE(proposals)
26:             proposals[round_i][v] ← proposals[round_i][v] ∪ {REPLICA_ID}
27:     if proposedCertificate ≠ ⊥ ∧ proposedCertificate.round < SIZE(proposals) − 1 then
28:         proposedValue ← ⊥; proposedCertificate ← ⊥
29:     if remote.proposedCertificate ≠ ⊥ then
30:         if remote.proposedCertificate.round ≥ SIZE(proposals) then   ▷ reject start of illegal proposed certificate
31:             byzReplicaIds ← byzReplicaIds ∪ {remote.REPLICA_ID}
32:         else if remote.proposedCertificate.round = SIZE(proposals) − 1 then
33:             if SIZE(proposals[lastRound][remote.proposedValue]) < QUORUM then
34:                 byzReplicaIds ← byzReplicaIds ∪ {remote.REPLICA_ID}
35:             else if proposedCertificate ≠ ⊥ then                   ▷ merge proposed certificates
36:                 proposedCertificate.votes ← proposedCertificate.votes ∪ remote.proposedCertificate.votes
37:             else                                                ▷ accept newer proposed certificate
38:                 proposedValue ← remote.proposedValue; proposedCertificate ← remote.proposedCertificate
39:     if proposedCertificate ≠ ⊥ then
40:         if SIZE(proposedCertificate.votes \ byzReplicaIds) ≥ QUORUM then                  ▷ commit
41:             value ← proposedValue; certificate ← proposedCertificate
42:             proposals ← [ ]; proposedValue ← ⊥; proposedCertificate ← ⊥
43:         else if ¬HAS_VOTED(proposedCertificate) then          ▷ cast vote for new commit certificate
44:             v ← GET_WINNING_VALUE(proposals)
45:             if v = proposedValue then
46:                 proposedCertificate.votes ← proposedCertificate.votes ∪ {REPLICA_ID}
47:             else                                                             ▷ start new round
48:                 proposals[lastRound + 1][v] ← {REPLICA_ID}
49:     else if ∑_v SIZE(proposals[lastRound][v] \ byzReplicaIds) ≥ QUORUM then
50:         if ∃ v : SIZE(proposals[lastRound][v] \ byzReplicaIds) ≥ QUORUM then   ▷ start vote for new commit certificate
51:             proposedValue ← v
52:             proposedCertificate ← (certificate.version + 1, lastRound, {REPLICA_ID})
53:         else                                                                 ▷ start new round
54:             v ← GET_WINNING_VALUE(proposals)
55:             proposals[lastRound + 1][v] ← {REPLICA_ID}
56: end procedure
```

Fig. 2. Consensus protocol for the Atomic Register.

```
1: procedure GET_WINNING_VALUE(proposals)
2:     for ∀ round_i ∈ proposals do
3:         if ∃ v ∈ proposals[round_i] : (∀ v' ∈ proposals[round_i] ∧ v ≠ v' :
               SIZE(proposals[round_i][v] \ byzReplicaIds) > SIZE(proposals[round_i][v'] ∪ byzReplicaIds)) then
4:             return v
5:     return v ∈ proposals[lastround] : (∀ v' ∈ proposals[lastround] ∧ v ≠ v' :
               SIZE(proposals[lastround][v] \ byzReplicaIds) ≥ SIZE(proposals[lastround][v'] \ byzReplicaIds))
6: end procedure
7: procedure GET_POSSIBLE_WINNING_VALUES(proposals)
8:     if ∃ id : id ∈ proposals[lastRound][v_1] ∧ id ∈ proposals[lastRound][v_2] ∧ v_1 ≠ v_2 then
9:         return {v ∈ proposals[0]}
10:    else
11:        return {GET_WINNING_VALUE(proposals)} ∪ {v_1 ∈ proposals[lastRound] : SIZE(proposals[lastRound][v_1]) > f}
12: end procedure
```

Fig. 3. Selection procedure for the current value to endorse.

time a new signature is received, it needs to be verified. This can become quite costly, and therefore WebLedger will use an optimistic approach. WebLedger will delay the verification of any incoming signatures and 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 commit certificate. Only then, all signatures will be verified in one batch. If all of the signatures are valid, the protocol can continue as normal. If there are invalid signatures, then those will be removed and WebLedger will continue to collect more signatures. However, WebLedger will remember this occurrence and from now on verify all signatures once they come in. Once consensus is reached for this version, WebLedger will move back to the optimistic fast path. 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 Correctness

This section sketches the proof that the algorithm provides safety and liveness. The protocol described before guarantees both safety and liveness when there are at least $2f + 1$ honest replicas available.

3.3.1 Safety

The safety property is defined as *non-divergence*.

Lemma 1 (Non-divergence). Let $\mathfrak{R}$ be a cluster of n replicas with f Byzantine nodes and with $n > 3f$. If replicas $R_1, R_2 \in \mathfrak{R}$ are able to construct commit certificates c_1 and c_2 for value v_1 and v_2 respectively with $c_{1\ version} = c_{2\ version}$, then $v_1 = v_2$.

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

Lemma 2. If replicas $R_1, R_2 \in \mathfrak{R}$ are able to construct commit certificates c_1 and c_2 for value v_1 and v_2 respectively with $c_{1\ version} = c_{2\ version}$ and $c_{1\ round} = c_{2\ round}$, then $v_1 = v_2$.

Proof: Assume two different replicas R_1 and R_2 have constructed a commit certificate c_1 and c_2 for value v_1 and v_2 respectively with $c_{1\ version} = c_{2\ version}$ and $c_{1\ round} = c_{2\ round}$. They are constructed in the same round, so of the n possible votes, at least $n-f$ replicas have voted on v_1, and at least $n - f$ replicas have voted on v_2. Honest replicas will never vote twice in the same version and round. Therefore at least $n - 2f$ honest replicas have voted on v_1 and $n - 2f$ *different* honest replicas have voted on v_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 $2n - 3f \geq 3f + 2 \geq n+1$ replicas. 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 v_1 and v_2 have to be equal. □

Lemma 3. If replicas $R_1, R_2 \in \mathfrak{R}$ are able to construct commit certificates c_1 and c_2 for value v_1 and v_2 respectively with $c_{1\ version} = c_{2\ version}$, then $c_{1\ round} = c_{2\ round}$.

Proof: Assume two different replicas R_1 and R_2 have constructed a commit certificate c_1 and c_2 for value v_1 and v_2 respectively with $c_{1\ version} = c_{2\ version}$ and $c_{1\ round} < c_{2\ round}$. Since c_1 is accepted, at least $n - f$ replicas vote on the proposed commit certificate and at least $n - f$ replicas voted on v_1 in round $r_{1\ round}$. The fact that $n - f$ replicas voted on the proposed commit certificate means that at least $n - 2f$ honest replicas observed $n - f$ votes for v_1. Of those votes, at least $n - 2f$ are coming from honest replicas. The only way to now construct a commit certificate c_2 for v_2 is to start a new round. To start a new round, a replica needs to not have voted for the proposed commit certificate c_1, and observe a different winning value v_2. Yet, at least $n - 2f$ honest replicas observed that at least $n - 2f$ honest replicas think that v_1 is the winning value. This leaves only $2f$ replicas who can prefer another value v_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 v_1 is the winning value, together with f Byzantine replicas. Value v_2 has only $2f$ supporting replicas, which is not enough to start a proposed commit certificate. So, at least one replica currently supporting v_1 needs to switch votes in a future round. However, once a replica has voted for a proposed commit certificate, it will not change their 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 commit certificate, it is impossible that a valid new round can be started. □

3.3.2 Liveness

The liveness property is defined as *termination*. When a new value is proposed, eventually the protocol will end and a valid commit certificate is created for a new value. 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.

Lemma 4 (Termination). If an honest replica $R \in \mathfrak{R}$ creates a proposal p for a new value v, eventually the replica will be able to construct a valid commit certificate c.

Lemma 5. If only a single replica $R \in \mathfrak{R}$ creates a proposal p for a new value v, eventually the replica will be able to construct a valid commit certificate c.

Proof: As there is only a single proposed value, all honest replicas who observe this will cast their vote for that value. Eventually, one replica will observe $n-f$ votes for V and a new proposed commit certificate will be constructed. Eventually, $n-f$ votes will be cast to this proposed commit certificate and a valid commit certificate c is constructed and v is committed. □

Lemma 6. If x replicas $R_{1..x} \in \mathfrak{R}$ create proposals $p_{1..x}$ for values $v_{1..x}$, and no Byzantine replicas vote twice in the same round, eventually the replica will be able to construct a valid commit certificate c.

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. As long as not all votes are known to all voting replicas, the winning value might change. In each new round, either an unknown vote stays unknown, or it becomes known. In the former case, then the currently known votes will all be the same, and a proposed commit 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, and every replica will observe the same winning value, and a commit certificate can be created. □

Lemma 7. If x replicas $R_{1..x} \in \mathfrak{R}$ create proposals $p_{1..x}$ for values $v_{1..x}$, eventually the replica will be able to construct a valid commit certificate c.

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. If this happens repeatedly, 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 the first round, or the first few rounds when Byzantine replicas vote twice, 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 commit certificate can be started, which will be committed. □

4 Architecture and Implementation

This section describes the architecture, deployment, and implementation of WebLedger. This middleware architecture is key to support the BFT consensus and synchronization protocol described in the previous section. The middleware is fully web-based 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 ledger. The last subsection lists several performance optimization tactics.

4.1 Overall architecture

The WebLedger middleware architecture consists of five main components (Fig. 4): (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.

(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 of the atomic register at the given key,
- `SET(key, value)` submits a proposal to update the atomic register at the given key,
- `DELETE(key)` deletes the atomic register 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 register is confirmed by the network.

Apart from those functions, the middleware also provides a constructor function to initialize the middleware by passing the following configuration as parameters:

- the list of all members of the network, together with their public key,
- the private key of the replica,

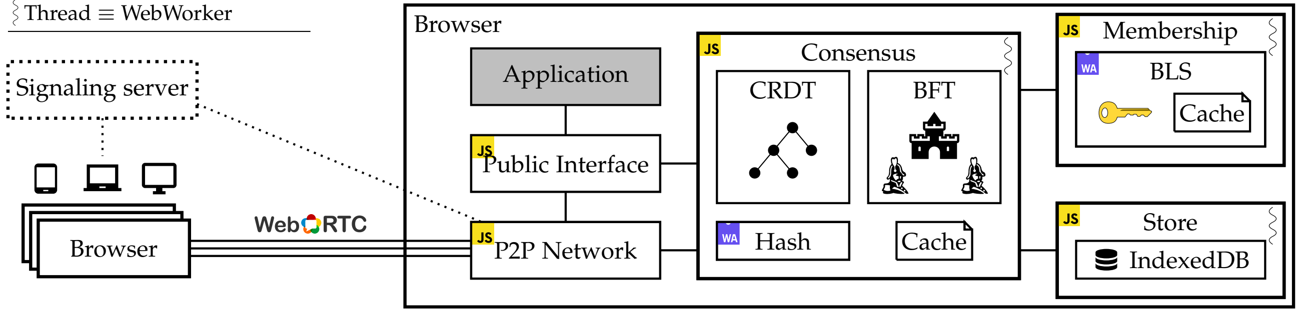

Fig. 4. Browser-based architecture of WebLedger.

- the URL to the signaling server to set up the peer-to-peer connections (explained in Section 4.1, § *(ii)*),
- an access-control callback to verify state-changes.

This access-control 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 requiring each new value to be signed by the owner. This value can only be changed by the owner, and also 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) [38]. 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 of 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.

(iii) Consensus. The *Consensus* component handles the consensus protocol described in Section 3. It maintains a Merkle-tree of all atomic registers and uses the state-based CRDT framework OWebSync [22] to replicate the local state to other replicas using the *P2P Network* component. The Merkle-tree is constructed using the Blake3 [39] cryptographic hash function.

(iv) Membership. The *Membership* component contains all cryptographic material and is responsible for the signing and verification operations. The *Consensus* component uses this for all cryptographic operations. We implemented two different versions of this component, one using ECDSA for signatures using the built-in WebCrypto [40] browser API (not shown in Fig. 4), and a second implementation using an aggregate signature scheme called BLS [41]. Section 4.2 provides more details about the BLS implementation.

(v) Store. At last, the *Store* component saves all state to the IndexedDB [42] database. IndexedDB is a key-value datastore built inside the browser. Each atomic register and the Merkle-tree are serialized to bytes and stored here under the respective key. This enables users to close the browser and continue afterward without losing the current state.

4.2 Aggregate signatures using BLS

The consensus protocol in Section 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, our protocol requires short and compact signatures to reduce storage and network footprint.

Boneh–Lynn–Shacham (BLS) [41] 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 [43] outputs a single aggregate signature as short and compact as the individual signatures, unlike other approaches that rely on ECDSA or DSA (e.g. Schnorr [44]).

Other state-of-the-art BFT systems such as SBFT [45] and HotStuff [14] also use aggregate or threshold signatures. However, they use it in a different way. They let the leader compute the aggregate signature. WebLedger uses a different approach, once a proposed commit certificate has reached a supermajority of the votes, any replica can aggregate these into one single aggregated BLS signature.

Efficient aggregation. The protocol described in Section 3 performs a considerable number of signature aggregations. But the standard scheme is vulnerable to rogue public-key attacks. The state-of-the-art approach [46] 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,

$\mathbb{G}_0$ and $\mathbb{G}_1$ are two multiplicitive cyclic groups of prime order q. $\mathsf{H}_0 : \{0,1\}^* \rightarrow \mathbb{G}_0$ and $\mathsf{H}_1 : \{0,1\}^* \rightarrow \mathbb{Z}_q$ 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 [46]. e is an efficient non-degenerating bilinear map $e : \mathbb{G}_0 \times \mathbb{G}_1 \rightarrow \mathbb{G}_t$. 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 take 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 $e(g_1, \sigma) \stackrel{?}{=} e(apk, \mathsf{H}_0(m))$.

Fig. 5. Formal specification of the BLS signature scheme.

and as a result, these computations are performed once. Now, any replica can run Agg by only computing $\sigma_1 ... \sigma_n$. The security properties of BLS remain intact [46], and we obtain more efficient aggregations at scale. For the interested reader, we provide the mathematical background and formal specification of our optimized BLS scheme in Fig. 5.

4.3 Performance optimization tactics for browsers

This section contains four important performance optimizations to host this middleware inside web browsers at scale.

Polyglot middleware. WebAssembly [47] is a binary instruction format that addresses the problem of safe, fast, and portable low-level code on the Web. Higher-level languages such as C, C++, and Rust can be compiled to WebAssembly and can be executed in a modern browser on any platform independent from the underlying hardware. WebAssembly executes significantly faster than JavaScript [48], however, it is not as fast as native code [49]. We used WebAssembly for two key components that are computationally intensive. These components are the hashing component to build the Merkle-tree and the BLS module for aggregate signatures. They are implemented in the Rust programming language [50] and C respectively, and they are compiled to WebAssembly to run inside a browser. Besides the performance improvement of WebAssembly over JavaScript, using Rust and C also enabled us to make use of well-tested libraries (BLAKE3 [51] and blst [52]) instead of implementing these components ourselves in JavaScript.

Parallellization using Web Workers. Web Workers [53] are separate browser threads, which enable us to run computations off the main thread to keep the User Interface responsive. The middleware is distributed over four different threads. The *Public interface* and *P2P Network* components run on the main thread together with the application. The *P2P Network* component is also located on the main thread because WebRTC is not available inside Web Workers. The other three components: *Consensus*, *Membership* and *Store*, are each located in a separate Web Worker. This enables long-running computations, for example, BLS-signature verification, to run in a separate thread without blocking concurrent operations in the other threads.

Caching. Caching is used in several places for performance reasons. The most important place is in the *Membership* component in WebAssembly. While WebAssembly itself is fast, the boundary between JavaScript and WebAssembly is not. Function calls between the two environments can only use numbers directly. Any other data structure has to be serialized to bytes and be allocated a spot in the WebAssembly memory buffer. In WebAssembly, these bytes can be decoded to the appropriate Rust data structure. For this reason, all cryptographic material such as public keys and the private key are passed to WebAssembly at initialization, avoiding this costly transfer for subsequent operations. In the *Consensus* component, all CRDT and Merkle-tree structures are cached in memory. As such, a costly fetch from disk and decoding from bytes can be avoided.

Batching of writes for IndexedDB. The last important optimization concerns IndexedDB [42]. IndexedDB is an in-browser database for structured data supporting fast reads and lookups by using indexes. We found that when too many write requests are sent to IndexedDB, latency significantly starts to increase up to one second or even more. When one atomic register is updated, also all intermediate nodes until the root node of the Merkle-tree are updated. This is due to the tree-shaped structure of the Merkle-tree. So, one write somewhere down the tree, leads to a cascading of writes, and every write causes the root node to be written as well. To reduce the high latency, we batched all writes to IndexedDB in-memory in the *Store* component. If multiple writes for the same key happen in the same batch, only the last one is executed. At fixed intervals of 5 seconds, the whole batch is written to IndexedDB. Since many duplicate writes are now avoided, the number of writes is reduced significantly. This solved the problem of high read latency. As not everything is immediately written to disk, failure can happen and lead to data loss. For updates received through the P2P network, this is not a problem as those updates can be synchronized again later since the Merkle-tree will detect the missing updates. Local update operations by the user on this replica, are immediately written to disk and bypass the write-batching to avoid data loss.

5 Evaluation

We validated the WebLedger middleware with the loyalty points use case. The first section presents this validation. Next, we present four different benchmarks with different scales. The first benchmark shows the performance results in the optimistic scenario with no network failure or Byzantine failures. The second benchmark evaluates the performance in a more realistic scenario with some network failures.

The third benchmark evaluates the performance in the presence of a Byzantine replica. The last benchmark compares two different implementations of WebLedger. The default version uses BLS signatures which supports signature aggregation using WebAssembly as explained in Section 4.2. The other version uses ECDSA signatures using the built-in native WebCrypto [40] APIs from the browser.

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 WebLedger to two other existing state-of-the-art systems for BFT consensus: BFT-SMaRt [10] and Tendermint [11], [28].

Test setup. To test the performance of the middleware, 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 [54], which simulates the latency of a 4G network [55]. Every test is executed 10 times to ensure the results are reliable.

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, network bandwidth, and disk usage 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 [56].

5.2 Optimistic scenario

In the optimistic scenario, every replica is honest and no replicas fail, meaning that the fast path can be used. One single aggregate signature is verified before each 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.

Fig. 6 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. WebLedger can confirm transactions within 4 seconds, even with a network of one hundred browsers. BFT-SMaRt 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, WebLedger 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 sign requests, which are an order of magnitude faster than the asymmetric signatures used in WebLedger and Tendermint. We can see a similar pattern in the bandwidth requirements shown in Fig. 7. In the large-scale scenario with 100 browsers, WebLedger uses less than 3 Mbit/s, which is acceptable for a typical mobile network.

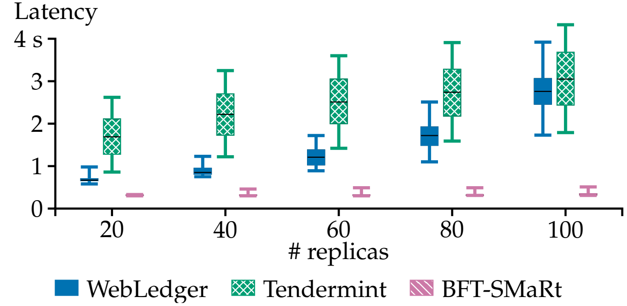

Fig. 6. Latency in the optimistic scenario with no failures.

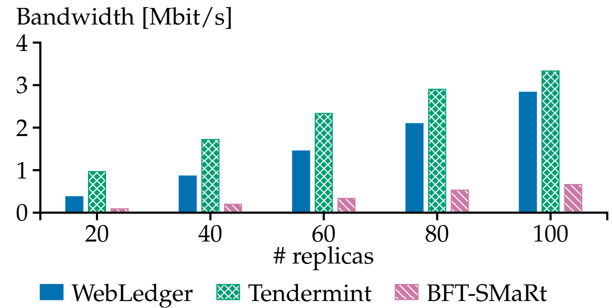

Fig. 7. Network usage in the optimistic scenario with no failures.

5.3 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

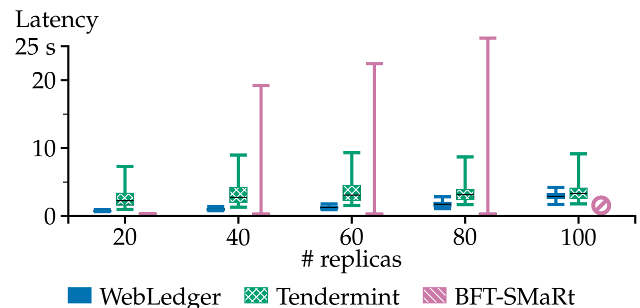

Fig. 8. Latency in the realistic scenario with network failures.

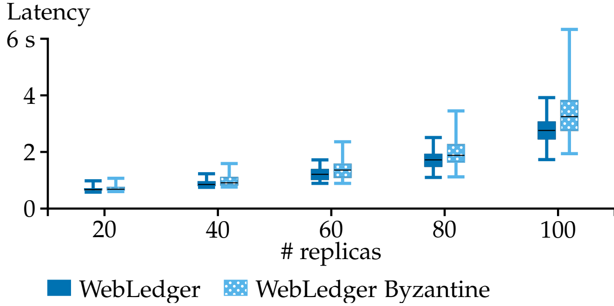

Fig. 9. Comparison of the latency in the normal scenario with a scenario where a Byzantine replica tries to halt the network.

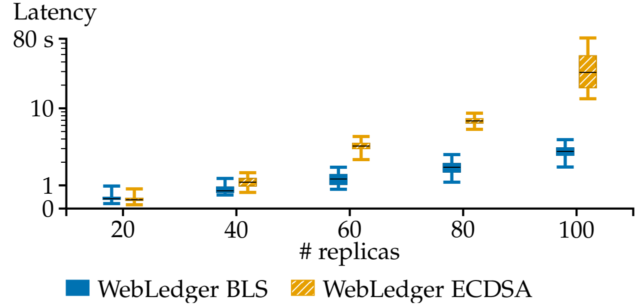

Fig. 10. Comparison of the latency in the normal scenario between the use of BLS signatures in WebAssembly and the ECDSA signatures the browser provides.

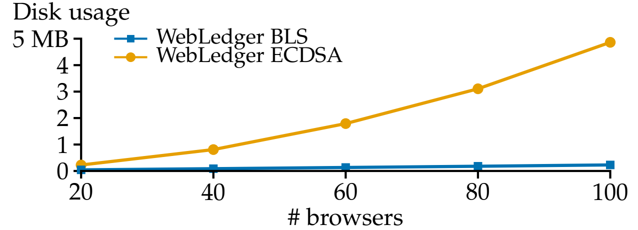

Fig. 11. Average disk usage for WebLedger.

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.

Fig. 8 shows the latency in this scenario. WebLedger is not impacted much by the failing replicas, and can still confirm transactions within 5 seconds. The impact on Tendermint is also small, but 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 Fig. 8. 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 on mobile networks. This has been shown in the literature before [57]. 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.

5.4 Byzantine scenario

For WebLedger, we performed an extra benchmark with Byzantine replicas. As long as the honest replicas are still using the optimistic fast path, the Byzantine replicas 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 replicas are Byzantine. Once the optimistic 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 replicas keep the signature intact to avoid being detected. However, they will try to slow down the consensus by not voting themselves.

The latency in this Byzantine scenario is shown in Fig. 9. WebLedger 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. 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 [58].

5.5 Benefits of BLS vs ECDSA

WebLedger uses BLS signatures to limit both the overhead of signature verification and storage. With BLS, only one aggregate signature of the q replicas needs to be verified, compared to q separate signature verifications for ECDSA. Fig. 10 compares the latency of the default implementation using BLS signatures with an alternative implementation using ECDSA signatures. The ECDSA implementation performs well for small networks but needs too much time in the larger networks with 80 and 100 replicas.

The BLS signature verifications are performed using WebAssembly. While WebAssembly can be much faster than JavaScript, the resulting WebAssembly code is still an order of magnitude slower compared to a native variant in *x64* assembly (which cannot be executed on the web). For the ECDSA signatures on the other hand, we used the built-in Web Cryptography API [40]. These are implemented in the browser using native functions. But even with the more optimized implementation of ECDSA, the version of WebLedger using BLS and WebAssembly is much faster for larger networks. In the future, browsers could add the BLS signature scheme natively, which would result in even better performance for WebLedger.

Fig. 11 shows the storage usage for both implementations of WebLedger. BLS improves disk usage about 20 times for the scenario with 100 browsers. Both implementations need less than 5 MB to store 1000 tokens. This disk usage does not increase over time, as only the current value and proposals are stored. We do not store a chain of all transactions that have happened so far. This is a big difference with blockchains that grow in size with every transaction that is executed and stored in the blockchain. This makes our approach feasible for resource-constrained devices that do not have hundreds of gigabytes of storage capacity to store a full blockchain.

5.6 Discussion and conclusions

We have shown that WebLedger can be used for the loyalty points use case with up to 100 different merchants, even when some of them are acting maliciously. WebLedger can achieve similar latencies as other Gossip-based BFT protocols, such as Tendermint. Traditional leader-based BFT protocols, such as BFT-SMaRt, are much faster in the optimistic setting. However, in the more realistic and mobile environment we envision, this dependability on a long-term leader results in long tail-latencies when that leader fails. WebLedger does not use a leader and is especially robust against network and node failures, which are typical in a mobile setting. BFT-SMaRt also requires that the leader is connected to all other replicas, and at least a supermajority of the replicas need to be online at the same time. WebLedger does not impose this, consensus can be reached gradually over time, as the full state of the proposals and votes propagates through the network. WebLedger can confirm transactions fast, in the order of seconds, without needing a complex back-end setup or wasting a lot of energy. WebLedger has a small storage footprint due to its state-based nature.

6 Related work

Several client-side frameworks for data synchronization between web applications exist: Legion [19], Yjs [20], [59], Automerge [21], and OWebSync [22]. They make use of various kinds of Conflict-free Replicated Data Types (CRDT) [34] 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 [60], [61]. None of them can tolerate Byzantine parties.

Open or permissionless blockchains such as Bitcoin [6] and Ethereum [62], [8] allow everyone to participate and use Proof-of-Work (PoW) to reach agreement over the ledger [63]. However, PoW has several flaws [64]. PoW uses a lot of processing power and energy [65] and performs poorly in terms of latency. It assumes a synchronous network to guarantee safety. 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. Blockchains require a lot of storage space, as the full blockchain typically needs to be stored on every node. Simplified Payment Verification (SPV) mode [6] for clients can reduce the resource usage, at the cost of decentralization. PoW gains its security from the fact that one needs a lot of CPU power to control the network, which is too costly for an attacker compared to the revenue for a successful attack. Other variants of resource consumption exist such as Proof-of-Space [66] or Proof-of-Storage [67].

ByzCoin [68] uses PoW for a separate identity chain to guard against Sybil attacks but uses a BFT protocol to actually order transactions. ByzCoin makes use of collective signatures (CoSi) [69] and a balanced tree for the communication flow. CoSi makes use of aggregate signatures by constructing a Schnorr multisignature [44]. However, CoSi needs multiple communication round-trips through the peer-to-peer network to generate the multi-signature and assumes a synchronous network.

Tendermint [11], [28], used in Cosmos [31], 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 cryptocurrency to work together to control the network.

Instead of reaching consensus between all the replicas of the network, Stellar Consensus Protocol [70], [71] uses quorum slices to reach federated Byzantine agreement in an open network. Replicas should choose adequate quorum slices for safety. However, today's Stellar network is highly centralized and many replicas use the same few validators. Two failing validators can make the entire system fail [72].

Other protocols use a randomized approach. Ouroboros [13], HoneyBadger [73] and BEAT [74] use distributed coin flipping for consensus. HoneyBadger [73] also uses threshold signatures [29] for censorship resilience. Algorand [12] uses Verifiable Random Functions [75] to select a random committee for the next round. Avalanche [15], [76] 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. WebLedger supports propagation of votes over multiple hops.

Permissioned blockchains such as Hyperledger Fabric [27] have closed membership and often use a BFT consensus protocol to order transactions. The first known BFT protocol is Practical Byzantine Fault Tolerance (PBFT) [9]. Other protocols bring improvements to the original PBFT protocol. Zyzzyva [77] 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 [78] provides an abstraction for these BFT algorithms. These protocols are targeting a small number of replicas deployed on a local area network. They generally work in two phases: the first phase guarantees proposal uniqueness, and the second phase guarantees that a new leader can convince replicas to vote for a safe proposal. HotStuff [14] proposed a three-phase protocol to reduce complexity and simplify leader replacement. This makes HotStuff much more scalable. All of these algorithms use a leader to drive the protocol. When the leader is malicious, performance can degrade quickly [58]. GeoBFT [79] is a topology-aware and decentralized consensus protocol, designed for scalability in a geo-distributed setting.

Another approach is to use a trusted hardware component [80], [81], [82], [83], [84]. These approaches are faster and less computationally intensive but require specialized hardware to be present. Moreover, trusted execution environments have been broken in the past [85], [86], [87].

There are several proposals to improve the performance and response time of Hyperledger Fabric. StreamChain [88] reaches consensus over a stream of transactions instead of blocks. FabricCRDT [89] 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 [57] supports commuting transactions to be applied out-of-order. A novel design for gossip in Fabric [90] improves the block propagation latency and bandwidth. While these improvements make Hyperledger Fabric faster, none of them try to reduce the infrastructure requirements to be able to easily set up an untrusted peer-to-peer network.

The Bitcoin Lightning Network [91] or state channels for Bitcoin [92] or Ethereum [93], [94], [95] 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 assumption is violated, the underlying blockchain needs to be used to resolve the conflict, or a trusted third party can be used [96]. WebLedger uses a similar state-transitioning protocol where only the latest collectively agreed state needs to be stored. However, WebLedger can tolerate both failing and malicious replicas, without resorting to a blockchain or a trusted third party.

7 Conclusion

In this paper, we presented WebLedger. A browser-based middleware for decentralized, community-driven, web applications. WebLedger uses an optimistic, leaderless BFT consensus protocol, combined with a robust and efficient state-based synchronization protocol based on state-based CRDTs and Merkle-trees. WebLedger uses an optimized BLS scheme for efficient computation and storage of signatures. It supports a client-centric, browser-based, state-based, permissioned ledger with a low infrastructure and storage footprint for small-scale, citizen-driven, networks. WebLedger offers consistent and robust confirmation times to achieve finality of transactions in the order of seconds, even in failure settings and Byzantine environments. In contrast with traditional blockchains, WebLedger does not store a transaction log or blockchain, keeping the overall storage footprint small.

References

[1] T. Berners-Lee. (2017) Three challenges for the web, according to its inventor. World Wide Web Foundation. [Online]. Available: https://webfoundation.org/2017/03/web-turns-28-letter/

[2] (2020) About. Web3 Foundation. [Online]. Available: https://web3.foundation/about/

[3] H. Farahmand, "Guidance for assessing blockchain platforms," Gartner, Tech. Rep., 2019.

[4] (2019) Blockchain's big bang: Web 3.0. Gartner. [Online]. Available: https://blogs.gartner.com/avivah-litan/2019/08/08/blockchains-big-bang-web-3-0/

[5] K. Jannes, B. Lagaisse, and W. 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*, ser. SERIAL '19. NY, USA: ACM, 2019, p. 3–8.

[6] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[7] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer, "Karma: A secure economic framework for peer-to-peer resource sharing," in *Workshop on Economics of Peer-to-peer Systems*, vol. 35, no. 6, 2003.

[8] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[9] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. USA: USENIX Association, 1999, pp. 173–186.

[10] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with bft-smart," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN 2014. USA: IEEE, June 2014, pp. 355–362.

[11] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, University of Guelph, 2016.

[12] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. NY, USA: ACM, 2017, pp. 51–68.

[13] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Advances in Cryptology – CRYPTO 2017*. Cham: Springer, 2017, pp. 357–388.

[14] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. NY, USA: ACM, 2019, p. 347–356.

[15] T. Rocket, "Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies," AVA Labs, Tech. Rep., 2018. [Online]. Available: https://avalanchelabs.org/avalanche.pdf

[16] T. Steiner, "What is in a web view: An analysis of progressive web app features when the means of web access is not a web browser," in *Companion Proceedings of the The Web Conference 2018*, ser. WWW '18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2018, p. 789–796.

[17] K. Jannes, B. Lagaisse, and W. 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*, ser. EdgeSys '19. NY, USA: ACM, 2019, pp. 7–11.

[18] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, p. 37–42, Sep. 2015.

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

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

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

[22] K. Jannes, B. Lagaisse, and W. Joosen, "Owebsync: Seamless synchronization of distributed web clients," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2338–2351, 2021.

[23] A. Madhusudan, I. Symeonidis, M. A. Mustafa, R. Zhang, and B. 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. Portugal: SciTePress, 2019, pp. 163–171.

[24] PwC, "The sharing economy," Consumer Intelligence Series, Tech. Rep., 2015.

[25] S. Fromhart and L. Therattil, "Making blockchain real for customer loyalty rewards programs," Deloitte, Tech. Rep., 2016.

[26] M. Sauwens, K. Jannes, B. Lagaisse, and W. 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*, ser. PaPoC '21. NY, USA: ACM, 2021.

[27] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system

for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. NY, USA: ACM, 2018.
[28] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," 2018.
[29] V. Shoup, "Practical threshold signatures," in *International Conference on the Theory and Applications of Cryptographic Techniques*, ser. Eurocrypt 2000, Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 207–220.
[30] J. Sousa, A. Bessani, and M. 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)*, IEEE. USA: IEEE, 2018, pp. 51–58.
[31] J. Kwon and E. Buchman, "Cosmos whitepaper: A network of distributed ledgers." cosmos.network, White paper, 2019. [Online]. Available: https://cosmos.network/cosmos-whitepaper.pdf
[32] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, p. 288–323, Apr. 1988.
[33] L. Lamport, "On interprocess communication," *Distributed Computing*, vol. 1, no. 2, pp. 86–101, 1986.
[34] M. Shapiro, N. Perguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems*, ser. Lecture Notes in Computer Science, X. Défago, F. Petit, and V. Villain, Eds., vol. 6976. Berlin, Heidelberg: Springer Berlin Heidelberg, Oct. 2011, pp. 386–400.
[35] R. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology — CRYPTO '87*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378.
[36] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," *SIGMOD Rec.*, vol. 18, no. 2, p. 399–407, Jun. 1989.
[37] P. S. Almeida, A. Shoker, and C. Baquero, "Delta state replicated data types," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 162 – 173, 2018.
[38] C. Jennings, H. Boström, J.-I. Bruaroey, A. Bergkvist, D. Burnett, A. Narayanan, B. Aboba, and T. Brandstetter, "WebRTC 1.0: Real-time communication between browsers," W3C, Candidate Recommendation, 2019. [Online]. Available: https://www.w3.org/TR/2019/CR-webrtc-20191213/
[39] J. O'Connor, J.-P. Aumasson, S. Neves, and Z. Wilcox-O'Hearn, "Blake3: one function, fast everywhere," 2020. [Online]. Available: https://blake3.io/
[40] M. Watson, "Web cryptography api," W3C, Recommendation, 2017. [Online]. Available: https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126/
[41] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 514–532.
[42] A. Alabbas and J. Bell, "Indexed database api 2.0," W3C, Candidate Recommendation, 2018. [Online]. Available: https://www.w3.org/TR/2018/REC-IndexedDB-2-20180130/
[43] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *International Conference on the Theory and Applications of Cryptographic Techniques*, Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 416–432.
[44] C.-P. Schnorr, "Efficient signature generation by smart cards," *Journal of Cryptology*, vol. 4, no. 3, pp. 161–174, Jan 1991.
[45] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "Sbft: a scalable and decentralized trust infrastructure," in *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. USA: IEEE, 2019, pp. 568–580.
[46] D. Boneh, M. Drijvers, and G. Neven, "Compact multi-signatures for smaller blockchains," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer. Cham: Springer, 2018, pp. 435–464.
[47] A. Rossberg, "Webassembly core specification," W3C, Recommendation, 2019. [Online]. Available: https://www.w3.org/TR/2019/REC-wasm-core-1-20191205/
[48] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," *SIGPLAN Not.*, vol. 52, no. 6, p. 185–200, Jun. 2017.
[49] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of webassembly vs. native code," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 107–120.
[50] N. D. Matsakis and F. S. Klock, "The rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14. NY, USA: ACM, 2014, p. 103–104.
[51] "Blake3," 2021. [Online]. Available: https://github.com/BLAKE3-team/BLAKE3/
[52] "blst," 2021. [Online]. Available: https://github.com/supranational/blst/
[53] I. Hickson, "Web workers," W3C, Working Draft, 2015. [Online]. Available: http://www.w3.org/TR/2015/WD-workers-20150924/
[54] W. Almesberger, "Linux network traffic control – implementation overview," 1999.
[55] OpenSignal, "Mobile network experience report," https://www.opensignal.com/reports/2019/01/usa/mobile-network-experience, 2019.
[56] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07, vol. 41(6). NY, USA: ACM, 2007, pp. 205–220.
[57] L. Bonniot, C. Neumann, and F. Taïani, "Pnyxdb: a lightweight leaderless democratic byzantine fault tolerant replicated datastore," in *The 39th IEEE International Symposium on Reliable Distributed Systems (SRDS '20)*, ser. The 39th IEEE International Symposium on Reliable Distributed Systems. Shanghai, China: IEEE, 2020.
[58] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, "Rbft: Redundant byzantine fault tolerance," in *IEEE 33rd International Conference on Distributed Computing Systems*. USA: IEEE, 2013, pp. 297–306.
[59] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, "Near real-time peer-to-peer shared editing on extensible data types," in *Proceedings of the 19th International Conference on Supporting Group Work*, ser. GROUP '16. NY, USA: ACM, 2016, pp. 39–49.
[60] A. van der Linde, J. a. Leitão, and N. Preguiça, "Practical client-side replication: Weak consistency semantics for insecure settings," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2590–2605, Jul. 2020.
[61] M. Barbosa, B. Ferreira, J. a. Marques, B. Portela, and N. Preguiça, "Secure conflict-free replicated data types," in *International Conference on Distributed Computing and Networking 2021*, ser. ICDCN '21. NY, USA: ACM, 2021, p. 6–15.
[62] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," ethereum.org, White paper, 2014.
[63] S. Gupta and M. Sadoghi, *Blockchain Transaction Processing*. Cham: Springer, 2018, pp. 1–11.
[64] C. Berger and H. P. Reiser, "Scaling byzantine consensus: A broad analysis," in *Proceedings of the 2Nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, ser. SERIAL'18. NY, USA: ACM, 2018, pp. 13–18.
[65] K. J. O'Dwyer and D. Malone, "Bitcoin mining and its energy footprint," in *Proceedings of the 2014 IET Irish Signals and Systems Conference*, ser. ISSC 2014/CIICT 2014. USA: IEEE, 2014, pp. 280–285.
[66] G. Ateniese, I. Bonacina, A. Faonio, and N. Galesi, "Proofs of space: When space is of the essence," in *Security and Cryptography for Networks*. Cham: Springer, 2014, pp. 538–557.
[67] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. NY, USA: ACM, 2007, p. 598–609.
[68] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *Proceedings of the 25th USENIX Conference on Security Symposium*, ser. SEC'16. USA: USENIX Association, 2016, p. 279–296.
[69] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping authorities "honest or bust" with decentralized witness cosigning," in *2016 IEEE Symposium on Security and Privacy (SP)*, ser. SP '16. USA: IEEE, May 2016, pp. 526–545.

[70] D. Mazieres, "The stellar consensus protocol: A federated model for internet-level consensus," Stellar Development Foundation, Tech. Rep., 2015.

[71] M. Lokhava, G. Losa, D. Mazières, G. Hoare, N. Barry, E. Gafni, J. Jove, R. Malinowsky, and J. McCaleb, "Fast and secure global payments with stellar," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. NY, USA: ACM, 2019, p. 80–96.

[72] K. Minjeong, K. Yujin, and K. Yongdae, "Is stellar as secure as you think?" in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*. USA: IEEE, 2019, pp. 377–385.

[73] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. NY, USA: ACM, 2016, pp. 31–42.

[74] S. Duan, M. K. Reiter, and H. Zhang, "Beat: Asynchronous bft made practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. NY, USA: ACM, 2018, p. 2028–2041.

[75] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *40th Annual Symposium on Foundations of Computer Science*, ser. FOCS '99, IEEE. USA: IEEE, 1999, pp. 120–130.

[76] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless bft consensus through metastability," 2019.

[77] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. NY, USA: ACM, 2007, p. 45–58.

[78] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, Jan. 2015.

[79] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, "Resilientdb: Global scale resilient blockchain fabric," *Proc. VLDB Endow.*, vol. 13, no. 6, p. 868–883, Feb. 2020.

[80] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient byzantine fault-tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[81] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "Cheapbft: Resource-efficient byzantine fault tolerance," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. NY, USA: ACM, 2012, p. 295–308.

[82] J. Behl, T. Distler, and R. Kapitza, "Hybrids on steroids: Sgx-based high performance bft," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. NY, USA: ACM, 2017, p. 222–237.

[83] F. Zhang, I. Eyal, R. Escriva, A. Juels, and R. Van Renesse, "Rem: Resource-efficient mining for blockchains," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, p. 1427–1444.

[84] J. Liu, W. Li, G. O. Karame, and N. Asokan, "Scalable byzantine consensus via hardware-assisted secret sharing," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 139–151, 2018.

[85] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 973–990.

[86] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*. USA: IEEE, 2019, pp. 1–19.

[87] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "Lvi: Hijacking transient execution through microarchitectural load value injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*. USA: IEEE, 2020.

[88] Z. István, A. Sorniotti, and M. Vukolić, "Streamchain: Do blockchains need blocks?" in *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, ser. SERIAL'18. NY, USA: ACM, 2018, p. 1–6.

[89] P. Nasirifard, R. Mayer, and H.-A. Jacobsen, "Fabriccrdt: A conflict-free replicated datatypes approach to permissioned blockchains," in *Proceedings of the 20th International Middleware Conference*, ser. Middleware '19. NY, USA: ACM, 2019, p. 110–122.

[90] N. Berendea, H. Mercier, E. Onica, and E. Riviere, "Fair and efficient gossip in hyperledger fabric," in *IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. USA: IEEE, 2020.

[91] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2016. [Online]. Available: https://lightning.network/lightning-network-paper.pdf

[92] J. Lind, O. Naor, I. Eyal, F. Kelbert, E. G. Sirer, and P. Pietzuch, "Teechain: A secure payment network with asynchronous blockchain access," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. NY, USA: ACM, 2019, p. 63–79.

[93] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," 2017. [Online]. Available: https://plasma.io/plasma-deprecated.pdf

[94] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," in *Financial Cryptography and Data Security*. Cham: Springer, 2019, pp. 508–526.

[95] P. McCorry, C. Buckland, S. Bakshi, K. Wüst, and A. Miller, "You sank my battleship! a case study to evaluate state channels as a scaling solution for cryptocurrencies," in *Financial Cryptography and Data Security*. Cham: Springer, 2020, pp. 35–49.

[96] P. McCorry, S. Bakshi, I. Bentov, S. Meiklejohn, and A. Miller, "Pisa: Arbitration outsourcing for state channels," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, ser. AFT '19. NY, USA: ACM, 2019, p. 16–30.

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

Emad Heydari Beni is a Ph.D. candidate in the Department of Computer Science at KU Leuven in Belgium, and a member of the research group imec-DistriNet. His research activities are under the supervision of Prof. Dr. Wouter Joosen and Dr. Bert Lagaisse. He received his Master's degree in computer science from the University of Antwerp in 2014. His main research interests are in the area of adaptive and reflective middleware, cloud platforms, and applied cryptography.

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

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