MobBFT: Client-centric State-based BFT for Decentralized and Resilient Web Applications

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

imec-DistriNet, KU Leuven, Belgium
`{kristof.jannes, emad.heydaribeni, bert.lagaisse, wouter.joosen}@kuleuven.be`

Abstract. The web is shifting to a client-centric, decentralized model where web clients become the leading execution environment for application logic and data storage. However, current solutions to build decentralized web applications with multiple distrusting parties often involve a decentralized backend of servers running a BFT protocol between them. In this paper, we present MobBFT, a purely browser-based platform for decentralized BFT consensus in client-centric, community-driven web applications. We propose a novel, optimistic, leaderless consensus protocol, tolerating Byzantine replicas, combined with a robust and efficient state-based synchronization protocol. This protocol makes MobBFT well suited for the decentralized client-centric web and its dynamic nature with many network disruptions or node failures. Using a state-based protocol, no transaction log is stored, keeping the storage footprint small for client-centric devices.

1 Introduction

Browsers and client-side web technologies offer increasing capabilities to enable fully client-side web applications that can operate independently and in a standalone fashion, in contrast to the server-centric model [22,16]. Web 3.0 can be defined as the decentralized web where users are in control of their data, and that replaces centralized intermediaries with decentralized networks and platforms. Community-driven, decentralized networks can open the road to many use cases for the sharing economy [35] or shared loyalty programs for local communities [23]. Such client-centric collaborations can, for example, enable a small network of merchants in a local shopping street, or at a farmer's market to set up a shared loyalty program between the merchants in an ad-hoc fashion. These small-scale, specialized collaborative networks can empower motivated citizens to bring value to their local community, without involving an incumbent big-tech company that can change the rules unilateral at any moment.

However, current state-of-the-art peer-to-peer data synchronization frameworks for the browser such as Legion [33], Automerge [27], and OWebSync [24] focus on full replication and eventual consistency between trusted clients. Each replica can modify all data, and all modifications are automatically replicated to all replicas. These protocols lack Byzantine Fault Tolerance (BFT). Yet, they are easy to set up and *trusted* parties can quickly use these to synchronize and modify a shared data set between them.

Decentralized interactions between *distrusting* parties can be enabled by using a classical BFT consensus protocol such as PBFT [13], BFT-SMaRt [8], Tendermint [12], Algorand [17], Ouroboros [26], or HotStuff [47]. These classical BFT protocols are very fast and have a high throughput, but typically assume server-to-server communication with low-latency network connections, and assume every node is connected to all other nodes. Nakamoto consensus [40], used in several blockchains such as Bitcoin and Ethereum, relaxes this requirement and only requires a loosely coupled network. However, blockchains based on Nakamoto consensus are too slow for many use cases. They need minutes, or even an hour, to confirm a transaction with high probability. Moreover, they consume a large amount of energy and need a lot of processing power. At last, Avalanche consensus [42] tries to solve the scalability problem by using the concept of meta-stability. Only a small subset of replicas need to be sampled to reach consensus. However, you still need a connection to every other replica, as the replicas that you need to sample change continuously.

Ultimately, a decentralized web application should be able to run in a robust and resilient way over a network of online client devices such as smartphones. Such devices have a permanent yet unstable internet connection over a data subscription, and are operational and reactive most of the time. However, the existing BFT consensus protocols are designed for more server-like infrastructure that has lots of processing power, storage space, and a stable, low-latency network connection. The motivated citizens in our envisioned use cases do not have this kind of knowledge, budget, and infrastructure available to set up a private network of servers running a BFT protocol between them. They rather want to use their existing hardware such as a low-end computer, or even a mobile device. They could use a public blockchain network, at the cost of paying a fee for every transaction, which 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.

In this paper, we present MobBFT, a novel peer-to-peer data synchronization framework for decentralized web applications between mistrusting parties. MobBFT combines the efficient operation and lightweight setup of a peer-to-peer data synchronization framework with the resilience and fault tolerance of a BFT consensus protocol. The novel BFT protocol, optimized for unstable network conditions, does not require that all replicas are connected to each other. It also does not rely on a leader, removing the need for a costly leader-election procedure when this leader is malicious or loses its network connection temporary. The latter scenario is common in our target environment. Each browser replica only maintains the current authenticated state, and does not need to keep track of an operation log or transaction history, keeping the storage footprint small. To further reduce the storage and bandwidth requirements, we use an aggregate signature scheme called BLS [10]. This also reduces the computational requirements when all replicas are honest, as only a single aggregate signature has to be verified. The authenticated state and consensus votes are replicated over multiple hops using a gossip protocol.

This paper is structured as follows. Section 2 presents MobBFT's lightweight BFT consensus protocol and the state-based replication strategy. The detailed web-based middleware architecture of MobBFT is elaborated in Section 3. Our evaluation in Section 4 focuses on many aspects of performance in both the optimistic scenario as well as more realistic and even Byzantine scenarios. Section 5 elaborates on important related work. We conclude in Section 6.

2 State-based BFT protocol

This section explains the state-based consensus protocol used in MobBFT. First, it describes the adversary model and its properties. Then it explains the protocol specification[1].

Overview and adversary model. The protocol is a partially synchronous [15], leaderless, Byzantine fault tolerant consensus protocol. An adversary might corrupt up to f replicas of the $n \geq 3f + 1$ total replicas. They can deviate from the protocol in any arbitrary way. Such replicas are called Byzantine, while the replicas that are strictly following the protocol are called honest. We assume attackers are computationally bounded and it is infeasible to forge the used asymmetric signatures or find collisions for the used cryptographic hash functions.

The protocol is used to implement a register [31] 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 application-level conditions can be applied to limit who can write to it, and which values are acceptable given the previous value. MobBFT 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 set of replicas is fixed, and changes to the replica set have to be made outside the protocol. Consensus is reached for each register separately, which means that each register has its own instance of the MobBFT protocol.

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

- **Non-divergence:** If replicas $R_1, R_2 \in \mathfrak{R}$ are able to construct quorum certificates qc_1 for value val_1 and qc_2 for value val_2 at view v, then $val_1 = val_2$.
- **Termination:** If an honest replica $R \in \mathfrak{R}$ proposes a new value at view v, eventually every replica will be able to construct a quorum certificate qc for *some* value at view v.

[1] For reviewing purposes, the interested reviewer can find a formal safety and liveness proof of this protocol here: https://kristofjannes.com/reports/MobBFT-safety-and-liveness.pdf. We did not include it due to space constraints.

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. Notice that the value that is committed in this property is not necessarily the originally proposed value. It is not guaranteed that a value will be committed, as long as other concurrent values are proposed as well.

Protocol specification. The specification of the protocol is shown in Algorithm 1. Each register has its own state which consists of three parts. The first part is the current value and a quorum certificate. The quorum certificate contains signatures of a supermajority of $n - f$ replicas, and proves the validity of the value. The second part is a map, which maps rounds to a collection of votes for the next value. In each round, there can be multiple proposed values. The third part consists of a new proposed value and a partial quorum certificate for that value. This state is shown at the first 5 lines of Algorithm 1.

Consensus is reached in two steps, first a supermajority needs to be reached in the last round of the *votes*, then a supermajority needs to be reached for the next quorum certificate. The first step will establish a resilient quorum, while the second step will guarantee that sufficiently many replicas know that such a quorum has been achieved.

Reading and writing. When reading the value of a register, it will return the currently accepted value. This request is always executed on the local replica and does not involve any network requests. To write a new value, a replica has to propose a new value to the other replicas. This process is the `PREPARE` phase in Algorithm 1. The proposing replica adds the new value and its vote to round 0 of *votes*. As the protocol is leaderless, any replica can be a proposing replica and multiple replicas can propose a new value simultaneously. Replicas are only allowed to vote once in each round for each view, so if the replica already voted for another value in that round, it will have to wait until consensus is reached for the current set of *votes*, and propose the new value for the view after it.

State-based replication protocol. The full state is replicated by using a state-based Gossip protocol. Each time a new state is received, the local state is merged with the remote state. This protocol is a peer-to-peer version of OWebSync [24], which uses state-based Conflict-free Replicated Data Types (CRDTs) [43] combined with a Merkle-tree [37] to efficiently replicate the updated state. The CRDTs being used are Observed-Removed Maps [24] and Grow-only Sets [43]. There are extra constraints imposed on the CRDTs due to the Byzantine nature. The Merkle tree is used to efficiently replicate the state between any two replicas. If the state of two replicas is the same, only the root hash is sent and compared, which limits the network usage. If the states differ, the protocol descends in the tree looking for mismatching hashes to find out which registers must be synchronized. By using a state-based approach, rather than the operation-based approach of operation-based CRDTs [43], blockchains [40], or traditional BFT

Algorithm 1 Basic protocol (for replica r).

```
 1: $value \leftarrow \bot$                                  ▷ Current accepted value
 2: $commitQC \leftarrow \bot$                        ▷ Quorum certificate for $value$
 3: for $view \leftarrow 1, 2, 3, ...$ do
 4:     $votes \leftarrow \emptyset$                          ▷ $round \mapsto votesInRound$
 5:     $nextCommitQC \leftarrow \emptyset$
      ▷ PREPARE phase
 6:     as a proposing replica:
 7:         wait for value $val$ from client
 8:         $votes[0] \leftarrow \{\textsc{vote}(view, 0, val, \texttt{PRE-COMMIT})\}$
 9:     as a non-proposing replica:
10:         wait for any value in $votes$
11:     for $round \leftarrow 0, 1, 2, 3, ...$ do
      ▷ PRE-COMMIT phase
12:         if $\neg\textsc{hasVoted}(votes[round])$ then
13:             $val \leftarrow \textsc{winningValue}(votes[0])$
14:             $votes[round] \leftarrow votes[round] \cup \{\textsc{vote}(view, round, val, \texttt{PRE-COMMIT})\}$
15:         wait for $(n - f)$ votes in $votes[round]$
16:         $val \leftarrow \textsc{winningValue}(votes[round])$
17:         $valVotes \leftarrow \textsc{votesForValue}(votes[round], val)$
18:         if $\textsc{len}(valVotes) \geq (n - f)$ then
19:             $nextCommitQC \leftarrow nextCommitQC \cup \{\textsc{vote}(view, round, val, \texttt{COMMIT})\}$
20:         else
21:             $val \leftarrow \textsc{winningValue}(votes[0])$
22:             $votes[round + 1] \leftarrow \{\textsc{vote}(view, round + 1, val, \texttt{PRE-COMMIT})\}$
23:             continue
      ▷ COMMIT phase
24:         wait for $(n - f)$ votes in $nextCommitQC$:
25:             if $\textsc{len}(votes) - 1 > round$ then
26:                 $nextCommitQC \leftarrow \emptyset$
27:                 continue
28:         $value \leftarrow \textsc{value}(nextCommitQC)$
29:         $commitQC \leftarrow nextCommitQC$

30: function $\textsc{winningValue}(votesInRound)$
31:     return $argmax_{val}\textsc{len}(\{v \in votesInRound : v.val = val\})$
32: function $\textsc{votesForValue}(votesInRound, val)$
33:     return $\{v \in votesInRound : v.val = val\}$
34: function $\textsc{hasVoted}(votesInRound)$
35:     return $\exists\ v \in votesInRound : v.r = r$
36: function $\textsc{vote}(view, round, val, type)$
37:     return $\text{Vote}(val, r, \textsc{sign}(view, round, val, type, r))$
```

protocols, we only need to store the current state together with some metadata. There is no need to store the full log of all operations to later convince replicas that were temporarily offline of the new state. Replicas also do not need to keep track of the state of other replicas, or which messages are already received by which replica. If a new value and quorum certificate with a higher view are received, then the protocol will accept the new state, and the protocol will reset back to line 3 of Algorithm 1 with that newer view. Note that we do not explicitly show the gossiping in Algorithm 1 to keep the algorithm compact. During all phases in the algorithm, the state is continuously replicated to the other replicas.

Consensus. Consensus about which value will be accepted in a view is reached in two phases, called `PRE-COMMIT` and `COMMIT` in Algorithm 1. 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 quorum certificate. If a supermajority of the replicas have voted in a round, but not a single value reaches a supermajority, a new round is started and all replicas can vote again in this new round. The replicas are only allowed to vote on the current winner in round 0 according to their local state. Because each replica might have a different state on the current set of votes in round 0, there can still be multiple values in the next round without any supermajority for a single value. Another factor is Byzantine nodes trying to halt the system by voting not according to the rules. However, the set of possible values to vote on gets smaller with every round, and eventually the view of all the replicas on the votes in round 0 will become the same, and the winning value can be chosen unanimously. The reason for this is that a replica does not simply send a message with his vote to the others, but instead gossips the entire state. This includes all votes for the previous rounds. This means that when two replicas disagree with each other in a certain round, once they communicate with each other, they will learn each other's state. In the next round they will both vote for the same value (as their local state of $votes[0]$ will be the same). Malicious replicas can try to shift the balance to violate liveness, but with each round they have less possibility to do so. Because when they gossip $votes[i]$ they also gossip the previous rounds which should show why they voted on a certain value. If a replica detects that another replica is Byzantine, it will exclude this Byzantine replica permanently, and its votes do not count anymore.

Example. An example of this replication process is shown in Fig. 1. There are four non-Byzantine replicas with an empty set of $votes$ and empty $nextCommitQC$. The scenario starts at t_0 with replica A proposing a new value v (line 7-8 of Algorithm 1). The state is replicated to the other replicas randomly. In the example, the state is gossiped to replica B and C at t_1, and those replicas merge the received state with their local state. Since B and C did not yet vote in this view and round, they will cast their vote for the current winning value (line 10-14 of Algorithm 1). This process continues at t_2 when replica B sends its state to replica A and C. At t_2, replica C observes that a supermajority of the replicas

$$\begin{tikzcd}
A & \{0 \mapsto \{(v,A)\}\},\emptyset \arrow[r, dotted, no head] \arrow[dr] \arrow[ddr] & \{0 \mapsto \{(v,A)\}\},\emptyset \arrow[r, dotted, no head] & \{0 \mapsto \{(v,A),(v,B)\}\},\emptyset \arrow[r, dotted, no head] & {} \\
B & \emptyset,\emptyset \arrow[r, dotted, no head] & \{0 \mapsto \{(v,A),(v,B)\}\},\emptyset \arrow[ur] \arrow[dr] \arrow[r, dotted, no head] & \{0 \mapsto \{(v,A),(v,B)\}\},\emptyset \arrow[r, dotted, no head] & {} \\
C & \emptyset,\emptyset \arrow[r, dotted, no head] & \{0 \mapsto \{(v,A),(v,C)\}\},\emptyset \arrow[r, dotted, no head] & \{0 \mapsto \{(v,A),(v,B),(v,C)\}\},\{(v,C)\} & \\
D & \emptyset,\emptyset \arrow[r, dotted, no head] & \emptyset,\emptyset \arrow[r, dotted, no head] & \emptyset,\emptyset \arrow[r, dotted, no head] & {} \\
 & t_0 & t_1 & t_2 &
\end{tikzcd}$$

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

support value v, and it starts working on a new quorum certificate to determine if at least a supermajority of the replicas also knows about this (line 16-19 of Algorithm 1).

Delaying signature verification. For brevity, we did not show the actual verification of signatures in Algorithm 1. However, in the basic protocol, each time a new signature is received, it needs to be verified. This can become quite costly, and therefore MobBFT will use a fast path and delay the verification of any incoming signatures. MobBFT will just accept and replicate them, until a decision needs to be made, such as starting a new round or starting to create a new proposed quorum certificate. Only then, all signatures will be verified in one batch. If all signatures are valid, the protocol can continue as normal. If there are invalid signatures, then those will be removed and MobBFT will continue to collect more signatures and verify them on arrival. This hybrid approach enables very fast consensus when all replicas are honest, while gracefully degrading to a slower, more costly protocol that can detect which replicas are actively acting Byzantine.

3 Architecture and implementation

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

Overall architecture. The MobBFT middleware architecture consists of five main components (Fig. 2): (i) a *public interface* that offers an API for developers, (ii) a *peer-to-peer network* component to communicate directly with other browsers, (iii) a *consensus* component to handle the consensus protocol described in the previous section, (iv) a *membership* component to handle all cryptographic operations, and (v) a *store* component to save all state to persistent storage. The last three components run on a different browser thread by using Web Workers.

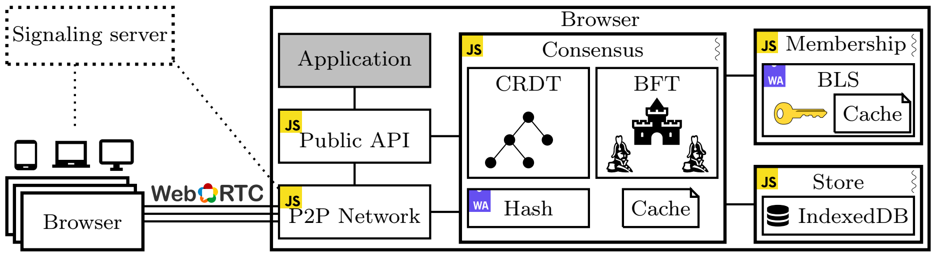

Fig. 2. Browser-based architecture of MobBFT.

(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 register at the given key, `SET(key, value)` submits a proposal to update the register at the given key, `DELETE(key)` deletes the 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 four configuration parameters: the list of all members of the network together with their public key, the private key of the replica, the URL to the signaling server to set up the peer-to-peer connections, and 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 supports passing ownership by changing the embedded public key.

(ii) Peer-to-peer network. The *P2P Network* component manages the peer-to-peer network and is responsible for the replication of the state-based CRDTs. Many browser-based replicas are connected to each other using WebRTC (Web Real-Time Communications). WebRTC enables a browser to communicate peer-to-peer. However, to set up those peer-to-peer connections, WebRTC needs a signaling server to exchange several control messages. Once the connection is set up, all communication can happen peer-to-peer, without a central server. Another WebRTC peer-connection can also be used as a signaling layer, so once a replica is connected to another one, it can also connect to all of its peers, without the need 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 2. It maintains a Merkle-tree of all registers and uses the state-based CRDT framework OWebSync [24] to replicate the local state to other replicas using the *P2P Network* component. The Merkle-tree is constructed using the Blake3 cryptographic hash function. For performance reasons, the hash function is implemented in Rust and compiled to WebAssembly.

(iv) Membership. The *Membership* component contains all cryptographic material and is responsible for all cryptographic operations such as signing and verification of signatures. We use an aggregate signature scheme called BLS [10]. Section 3 provides more details about the BLS implementation. It is implemented in C and compiled to WebAssembly.

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

Aggregate signatures using BLS. The consensus protocol in Section 2 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) [10] 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 outputs a single aggregate signature as short and compact as the individual signatures, unlike other approaches that rely on ECDSA, DSA or Schnorr. Other state-of-the-art BFT systems such as SBFT [18] and HotStuff [47] also use aggregate or threshold signatures. However, they use it in a different way. They let the leader compute the aggregate signature. MobBFT uses a different approach, once a proposed quorum certificate has reached a supermajority of the votes, any replica can aggregate these into one single aggregated BLS signature.

The standard scheme is vulnerable to rogue public key attacks. The state-of-the-art approach [9] to mitigate such attacks is to compute $(t_1, ..., t_n) \leftarrow \mathsf{H}_1(pk_1, ..., pk_n)$ for each Agg invocation and compute $\sigma \leftarrow \prod_{i=1}^{n} \sigma_i^{t_i}$, where pk_i is the public key of replica i, H_1 is a hash function, and σ_i is a signature produced by replica i. Although the t_i values can be cached, the computation of σ would be costly. Moreover, Agg does not take as input the same set of public keys at different states of a transaction in our consensus protocol. Therefore, we distribute the computations by moving the calculations of the t_i and $\sigma_i^{t_i}$ values to the signing parties, and as a result, these computations are performed only once. Now, any replica can run Agg by only computing $\sigma_1 ... \sigma_n$. The security properties of BLS remain intact [9], and we obtain more efficient aggregations at scale.

4 Evaluation

We validated the MobBFT middleware with a loyalty points use case [23]. The first section presents this validation. Next, we present three 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 last benchmark evaluates the performance in the presence of a Byzantine replica.

Validation in a loyalty points use case. Integrated loyalty programs can be more effective than traditional loyalty programs that are limited to a single company. Think about airlines that award *miles* which can be redeemed with several partners. Such collaborations usually introduce an extra trusted intermediary and add more layers of management and operational logistics. This trusted party can charge high transaction costs to be part of the integrated network. For small merchants on a farmer's market or in a local shopping street, this operational overhead is too much of a burden. A decentralized peer-to-peer network can enable fast and secure creation, redemption, and exchange of loyalty points across different merchants.

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 MobBFT to two other existing state-of-the-art systems for BFT consensus: BFT-SMaRt [8,45] and Tendermint [12]. BFT-SMaRt is a more traditional BFT protocol, similar to PBFT [44], where all replicas are connected to each other, and one leader drives the protocol. If that leader fails, a new one will have to be elected before any progress can be made. Tendermint [12] uses Gossip for communication between the replicas. There is still a leader, however, that leader changes frequently.

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, which simulates the latency of a 4G network. 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 and network bandwidth with a different number of browsers. We show a boxplot of the latency results instead of only the average, as all users should experience fast confirmation times, and not only the average user.

Optimistic scenario. In the optimistic scenario, every replica is honest and no replicas fail, so the fast path can be used. One single aggregate signature is verified 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. 3a 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. MobBFT 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, MobBFT and Tendermint use Gossip and need multiple hops before all replicas are reached. This also causes the increased latency. Furthermore, BFT-SMaRt uses HMAC to authenticate requests, which are an order of magnitude faster than the asymmetric signatures used in MobBFT and Tendermint. We can see a similar pattern in the bandwidth requirements shown in Fig. 3b. In the large-scale scenario with 100 browsers, MobBFT uses less than 3 Mbit/s, which is acceptable for a typical mobile network.

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

Fig. 4a shows the latency in this scenario. MobBFT is not impacted much by the failing replicas and can still confirm transactions within 5 seconds. The impact on Tendermint is also small, but the latency is doubled to about 10 seconds. BFT-SMaRt however needs to use a costly leader election protocol when

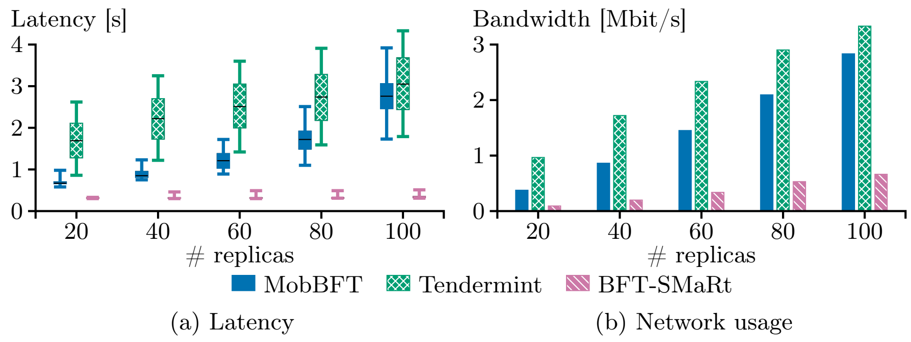

Fig. 3. Performance in the optimistic scenario with no failures.

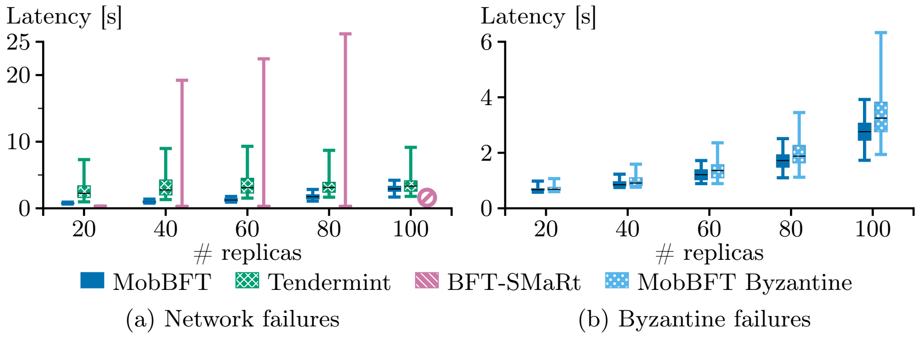

Fig. 4. Latency in the realistic scenario with network or Byzantine failures.

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. 4a. 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 [11]. 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.

Byzantine scenario. For MobBFT, we performed an extra benchmark with Byzantine replicas. As long as the honest replicas are still using the 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 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. 4b. MobBFT 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 [2].

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

5 Related work

Several client-side frameworks for data synchronization between web applications exist: Legion [33], Automerge [27], and OWebSync [24]. They make use of various kinds of Conflict-free Replicated Data Types (CRDTs) [43] 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 [34,3]. None of them can tolerate Byzantine parties.

WebBFT [7] shares a similar vision of client-centric, decentralized web applications. However, they only interface to a backend BFT-SMaRt cluster, instead of running the BFT protocol directly between browsers.

Open or permissionless blockchains such as Bitcoin [40] and Ethereum allow everyone to participate and use Proof-of-Work (PoW) to reach agreement over the ledger. However, PoW has several flaws [6]. PoW uses a lot of processing power and energy 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. Simplified Payment Verification (SPV) mode [40] for clients can reduce the resource usage at the cost of decentralization.

ByzCoin [29] uses PoW for a separate identity chain to guard against Sybil attacks but uses a BFT protocol to order transactions. ByzCoin makes use of collective signatures (CoSi) and a balanced tree for the communication flow. CoSi makes use of aggregate signatures by constructing a Schnorr multisignature. However, CoSi needs multiple communication round-trips to generate the multisignature and assumes a synchronous network.

Tendermint [12], used in Cosmos, uses Proof-of-Stake (PoS), where voting power is based on the amount of cryptocurrency owned by each replica. Because block times are short, in the order of seconds, there is a limited number of validators Tendermint can have because finality needs to be reached for each block. It is also not resistant to cartel forming, which allows those with a lot of cryptocurrencies to work together to control the network.

Other protocols use a randomized approach. Ouroboros [26], HoneyBadger [39], Dumbo [19] and BEAT [14] use distributed coin flipping for consensus. HoneyBadger [39] uses threshold encryption [44] for censorship resilience. Algorand [17] uses Verifiable Random Functions [38] to select a random committee for the next round. Avalanche [42] 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. MobBFT supports propagation of votes over multiple hops.

Permissioned blockchains such as Hyperledger Fabric [1] have closed membership and often use a BFT consensus protocol to order transactions. For example BFT-SMART in HyperLedger Fabric [8,45]. The first known BFT protocol is Practical Byzantine Fault Tolerance (PBFT) [13]. Other protocols bring improvements to the original PBFT protocol. Zyzzyva [30] 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. These protocols are targeting a small number of replicas in a local network. They generally work in two phases: the first guarantees proposal uniqueness, and the second guarantees that a new leader can convince replicas to vote for a safe proposal. HotStuff [47] proposed a three-phase protocol to reduce complexity and simplify leader replacement. This makes HotStuff more scalable. All these algorithms use a leader to drive the protocol. When the leader is malicious, the performance can degrade quickly [2]. GeoBFT [20] is a topology-aware, decentralized consensus protocol, designed for geo-distributed scalability. MobBFT does not use a leader and replicas communicate only to a subset of the other replicas using a gossip-like protocol. Another approach is to use a trusted hardware component [46,25,4]. These are faster and less computationally intensive but require specialized hardware to be present. Moreover, trusted execution environments have been broken in the past [28].

There are several proposals to improve the performance and response time of Hyperledger Fabric. StreamChain [21] reaches consensus over a stream of transactions instead of blocks. FabricCRDT [41] 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 [11] supports commuting transactions to be applied out-of-order. A novel design for gossip in Fabric [5] 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 Lightning Network or state channels for Bitcoin [32] or Ethereum are *off-chain* protocols that run on top of a blockchain. A new state channel between known participants is created by interacting with the blockchain. After its creation, participants can use this channel to execute state transitions by collectively signing the new state. These transactions do not involve the blockchain and have fast confirmation times and no transaction costs. However, state channels assume all participants to be always online and honest. If this is violated, the underlying blockchain needs to be used to resolve the conflict, or a trusted third party can be used [36]. MobBFT uses a similar state-transitioning protocol where only the latest collectively agreed state needs to be stored. However, MobBFT can tolerate both failing and malicious replicas, without resorting to a blockchain or a trusted third party.

6 Conclusion

In this paper, we presented MobBFT. A browser-based middleware for decentralized, community-driven web applications. MobBFT uses an client-centric, leaderless BFT consensus protocol, combined with a robust and efficient state-based synchronization protocol. MobBFT uses an optimized BLS scheme for efficient computation and storage of signatures. It supports a client-centric, browser-based, state-based, permissioned datastore with a low infrastructure and storage footprint for small-scale, citizen-driven networks. MobBFT 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 to traditional blockchains, MobBFT does not store a transaction log or blockchain, keeping the overall storage footprint small.

References

1. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S.W., Yellick, J.: Hyperledger Fabric: A distributed operating system for permissioned blockchains. In: EuroSys (2018)
2. Aublin, P.L., Mokhtar, S.B., Quéma, V.: RBFT: Redundant byzantine fault tolerance. In: ICDCS (2013)
3. Barbosa, M., Ferreira, B., Marques, J.a., Portela, B., Preguiça, N.: Secure conflict-free replicated data types. In: ICDCN (2021)
4. Behl, J., Distler, T., Kapitza, R.: Hybrids on steroids: SGX-based high performance BFT. In: EuroSys (2017)
5. Berendea, N., Mercier, H., Onica, E., Riviere, E.: Fair and efficient gossip in Hyperledger Fabric. In: ICDCS (2020)
6. Berger, C., Reiser, H.P.: Scaling byzantine consensus: A broad analysis. In: SERIAL (2018)
7. Berger, C., Reiser, H.P.: WebBFT: Byzantine fault tolerance for resilient interactive web applications. In: DAIS (2018)
8. Bessani, A., Sousa, J., Alchieri, E.E.P.: State machine replication for the masses with BFT-SMART. In: DSN (2014)
9. Boneh, D., Drijvers, M., Neven, G.: Compact multi-signatures for smaller blockchains. In: ASIACRYPT (2018)
10. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. In: ASIACRYPT (2001)
11. Bonniot, L., Neumann, C., Taïani, F.: PnyxDB: a lightweight leaderless democratic byzantine fault tolerant replicated datastore. In: SRDS (2020)
12. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus (2018)
13. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. In: OSDI (1999)
14. Duan, S., Reiter, M.K., Zhang, H.: BEAT: Asynchronous BFT made practical. In: CCS (2018)
15. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM (1988)
16. Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A., Barcellos, M., Felber, P., Riviere, E.: Edge-centric computing: Vision and challenges. SIGCOMM Comput. Commun. Rev. (2015)
17. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: SOSP (2017)
18. Gueta, G.G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M., Seredinschi, D.A., Tamir, O., Tomescu, A.: SBFT: a scalable and decentralized trust infrastructure. In: DSN (2019)
19. Guo, B., Lu, Z., Tang, Q., Xu, J., Zhang, Z.: Dumbo: Faster asynchronous bft protocols. In: CCS (2020)
20. Gupta, S., Rahnama, S., Hellings, J., Sadoghi, M.: ResilientDB: Global scale resilient blockchain fabric. VLDB (2020)
21. István, Z., Sorniotti, A., Vukolić, M.: StreamChain: Do blockchains need blocks? In: SERIAL (2018)
22. Jannes, K., Lagaisse, B., Joosen, W.: The web browser as distributed application server: Towards decentralized web applications in the edge. In: EdgeSys (2019)

23. Jannes, K., Lagaisse, B., Joosen, W.: You don't need a ledger: Lightweight decentralized consensus between mobile web clients. In: SERIAL (2019)
24. Jannes, K., Lagaisse, B., Joosen, W.: OWebSync: Seamless synchronization of distributed web clients. IEEE Trans. on Parallel and Distributed Systems (2021)
25. Kapitza, R., Behl, J., Cachin, C., Distler, T., Kuhnle, S., Mohammadi, S.V., Schröder-Preikschat, W., Stengel, K.: CheapBFT: Resource-efficient byzantine fault tolerance. In: EuroSys (2012)
26. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: CRYPTO (2017)
27. Kleppman, M., Beresford, A.R.: Automerge: Real-time data sync between edge devices (2018)
28. Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: S&P (2019)
29. Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., Ford, B.: Enhancing bitcoin security and performance with strong consistency via collective signing. In: SEC (2016)
30. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative byzantine fault tolerance. In: SOSP (2007)
31. Lamport, L.: On interprocess communication. Distributed Computing (1986)
32. Lind, J., Naor, O., Eyal, I., Kelbert, F., Sirer, E.G., Pietzuch, P.: Teechain: A secure payment network with asynchronous blockchain access. In: SOSP (2019)
33. van der Linde, A., Fouto, P., Leitão, J.a., Preguiça, N., Castiñeira, S., Bieniusa, A.: Legion: Enriching internet services with peer-to-peer interactions. In: WWW (2017)
34. van der Linde, A., Leitão, J.a., Preguiça, N.: Practical client-side replication: Weak consistency semantics for insecure settings. VLDB (2020)
35. Madhusudan, A., Symeonidis, I., Mustafa, M.A., Zhang, R., Preneel, B.: SC2Share: Smart contract for secure car sharing. In: ICISSP (2019)
36. McCorry, P., Bakshi, S., Bentov, I., Meiklejohn, S., Miller, A.: Pisa: Arbitration outsourcing for state channels. In: AFT (2019)
37. Merkle, R.: A digital signature based on a conventional encryption function. In: CRYPTO (1987)
38. Micali, S., Rabin, M., Vadhan, S.: Verifiable random functions. In: FOCS (1999)
39. Miller, A., Xia, Y., Croman, K., Shi, E., Song, D.: The honey badger of BFT protocols. In: CCS (2016)
40. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
41. Nasirifard, P., Mayer, R., Jacobsen, H.A.: FabricCRDT: A conflict-free replicated datatypes approach to permissioned blockchains. In: Middleware (2019)
42. Rocket, T., Yin, M., Sekniqi, K., van Renesse, R., Sirer, E.G.: Scalable and probabilistic leaderless BFT consensus through metastability (2019)
43. Shapiro, M., Perguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: SSS (2011)
44. Shoup, V.: Practical threshold signatures. In: Eurocrypt (2000)
45. Sousa, J., Bessani, A., Vukolic, M.: A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In: DSN (2018)
46. Veronese, G.S., Correia, M., Bessani, A.N., Lung, L.C., Verissimo, P.: Efficient byzantine fault-tolerance. IEEE Trans. on Computers (2013)
47. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: HotStuff: BFT consensus with linearity and responsiveness. In: PODC (2019)