BeauForT: Robust Byzantine Fault Tolerance for Client-centric Mobile Web Applications

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

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

Index Terms—Peer-to-peer systems, byzantine fault tolerance, web applications.

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 [1], [2]. Mobile applications are also more and more purely web-based clients, where the execution environment is just a browser-based process for a mobile web application. Web 3.0 can be defined as the decentralized web where users are in control of their data, and that replaces centralized intermediaries with decentralized networks and platforms. Community-driven, decentralized networks can open the road to many use cases for the sharing economy [3] or shared loyalty programs for local communities [4]. 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 [5], Automerge [6], [7], [8], and OWebSync [9] focus on full replication and eventual consistency between trusted clients. Each replica can modify all data, and all modifications are automatically replicated to all replicas. These protocols lack Byzantine Fault Tolerance (BFT). Yet, they are easy to set up and applications from *trusted* parties can leverage these to synchronize and modify a shared data set between them.

Decentralized interactions between *distrusting* parties can be enabled by using a classical BFT consensus protocol such as PBFT [10], BFT-SMaRt [11], or HotStuff [12]. These classical BFT protocols are very fast and have a high throughput, but typically assume server-to-server communication with low-latency network connections, and assume every node is connected to all other nodes. Other classical BFT consensus protocols, such as Tendermint [13], relax the requirement that every node is connected to every other node. Nakamoto consensus [14], used in several blockchains such as Bitcoin and Ethereum [15], 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 [16] tries to solve the scalability problem by using the concept of metastability. Only a small subset of replicas needs to be sampled in each round to reach consensus. However, a replica still needs a connection to every other replica, as the replicas that they need to sample change continuously.

Ultimately, a decentralized mobile application should be able to run in a robust and resilient way over a network of online client devices such as smartphones. We target an environment with 10-100 lightweight and mobile web clients. Such devices have a permanent yet unstable internet connection over a data subscription, and are operational and reactive most of the time. I.e., we assume those mobile devices always have a 3G or 4G connection, but this kind of connection is less stable than a wired connection and short disruptions are commonplace. Many existing protocols use all-to-all communication, which is simply not possible in a web-based environment. A browser can keep a connection open to 10-20 other browsers, but after that performance deteriorates quickly. Alternatively, there exist gossip-based protocols, such as Tendermint, that do not require a connection to every other node. However, Tendermint is leader-based, which in practice means that when this leader

- *K. Jannes, B. Lagaisse and W. Joosen are with imec-DistriNet, KU Leuven, 3001 Leuven, Belgium.*
  E-mail: {kristof.jannes, bert.lagaisse, wouter.joosen}@kuleuven.be.
- *E. Heydari Beni is with imec-DistriNet and imec-COSIC, KU Leuven, 3001 Leuven, Belgium.*
  E-mail: emad.heydaribeni@kuleuven.be.

 https://doi.org/10.1109/TPDS.2023.3241963

fails, consensus will be delayed until the next leader is elected. Moreover, these 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, that are running a BFT protocol between them. These citizens rather want to use their existing hardware such as a low-end computer, or even a mobile device.

In this paper, we present BeauForT, a novel peer-to-peer data synchronization framework for decentralized web applications between mistrusting parties. BeauForT combines the efficient operation and lightweight setup of a peer-to-peer data synchronization framework with the resilience and fault tolerance of a BFT consensus protocol. The novel BFT protocol, optimized for unstable network conditions with higher latencies, does not require that all replicas are directly connected to each other. It also does not rely on a leader, removing the need for a costly leader-election procedure when this leader is malicious or loses its network connection temporarily. The latter scenario is common in our target environment. Each browser replica only maintains the current authenticated state, and does not need to keep track of an operation log or transaction history, keeping the storage footprint small. To further reduce the storage and bandwidth requirements, we use an aggregate signature scheme called BLS [17]. This also reduces the computational requirements, as you can verify multiple signatures at once. The authenticated state and consensus votes are replicated over multiple hops using a gossip protocol.

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

1) Lightweight, leaderless, client-centric Byzantine fault tolerant consensus.
2) Resilient and robust, state-based synchronization of both the data and the votes for the consensus protocol using state-based CRDTs and Merkle-trees.
3) Delayed verification and aggregation of signatures using the BLS signature scheme.

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

This paper is structured as follows. Section 2 presents a motivational use case. Section 3 presents BeauForT's lightweight BFT consensus protocol and the state-based replication strategy. The detailed web-based middleware architecture of BeauForT is elaborated in Section 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

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

Loyalty programs. Integrated loyalty programs can be more effective than traditional loyalty programs that are limited to a single company [18]. 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.

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

3 BeauForT protocol

This section explains the state-based consensus protocol used in BeauForT. First, it describes the adversary model and its properties. Then it explains the protocol specification. Proofs can be found in Appendix A.

3.1 System model

We assume a partially synchronous network [19]. Messages can be delayed, dropped or delivered out of order. An adversary might corrupt up to f replicas of the $n \geq 3f + 1$ total replicas. They can deviate from the protocol in any arbitrary way. Such replicas are called Byzantine, while the replicas that are strictly following the protocol are called honest. At least $2f+1$ honest replicas should be able to make a connection to each other. In practice, they are transitively connected to each other, but only directly connected to a few replicas. The topology can change over time. If no progress is being made on a new proposal, replicas will close some existing connections and connect to a few different replicas. Each replica will gossip its neighbors to every replica it connects to. We assume attackers are computationally bounded

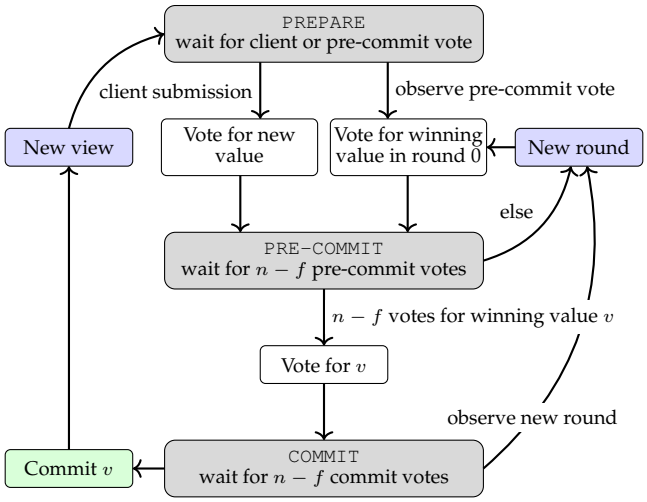

Fig. 1. State transition diagram of the BeauForT consensus protocol.

and it is infeasible to forge the used asymmetric signatures or find collisions for the used cryptographic hash functions.

We address in this paper a replicated key-value store for which replicas coordinate agreement using a Byzantine Fault Tolerant consensus protocol, such that the following classical properties hold [20]:

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

All writes to a key-value pair are atomic, meaning that only a single state transition can happen at any time. Extra application-level conditions can be applied to limit who can write to it, and which values are acceptable given the previous value. BeauForT does not use a leader to coordinate the protocol, removing a common single-point-of-failure compared to many existing BFT protocols. In such leader-based protocols, the failure of a leader leads to a long delay before consensus can be reached. This is even the case for rotating leader protocols such as HotStuff [21]. The set of replicas is fixed, and changes to the replica set have to be made outside the protocol, e.g., by halting the protocol, updating the set of replicas on all replicas, and start the protocol again. Consensus is reached for each key-value pair separately, which means that each key has its own instance of the BeauForT protocol.

3.2 Protocol specification

The specification of the protocol is shown in Algorithm 1. The state of a replica consists of three parts. The first part is the current value (line 1) and a quorum certificate (line 2). The quorum certificate contains signatures of a supermajority of $n-f$ replicas, and proves the validity of the value. The second part is a map, which maps rounds to a collection of votes for the next value (line 4). In each round, there can be multiple proposed values. The third part consists of a new

Algorithm 1 Basic protocol for replica id.

1: $value \leftarrow \perp$ ▷ Current accepted value
2: $qc \leftarrow \perp$ ▷ Quorum certificate for $value$
3: **for** $v \leftarrow 1, 2, 3, \ldots$ **do** ▷ view
4: $votes \leftarrow \emptyset$ ▷ $round \mapsto votesInRound$
5: $value' \leftarrow \perp$ ▷ Next value
6: $qc' \leftarrow \emptyset$ ▷ Next quorum certificate
▷ PREPARE phase
7: **as** a proposing replica:
8: **wait** for value val from client
9: $votes[0] \leftarrow \{\text{VOTE}(v, 0, val, \text{PRE-COMMIT})\}$
10: **as** a non-proposing replica:
11: **wait** for any value in $votes$
12: **for** $r \leftarrow 0, 1, 2, 3, \ldots$ **do** ▷ round
▷ PRE-COMMIT phase
13: **if** $\neg$HASVOTED($votes[r]$) **then**
14: $val \leftarrow$ WINNINGVALUE($votes[0]$)
15: $vote \leftarrow \text{VOTE}(v, r, val, \text{PRE-COMMIT})$
16: $votes[r] \leftarrow votes[r] \cup \{vote\}$
17: **wait** for $(n-f)$ votes in $votes[r]$
18: $val \leftarrow$ WINNINGVALUE($votes[r]$)
19: $valVotes \leftarrow$ VOTESFORVALUE($votes[r], val$)
20: **if** $\text{LEN}(valVotes) \geq (n-f)$ **then**
21: $vote \leftarrow \text{VOTE}(v, r, val, \text{COMMIT})$
22: $value' \leftarrow val$
23: $qc' \leftarrow qc' \cup \{vote\}$
24: **else**
25: $val \leftarrow$ WINNINGVALUE($votes[0]$)
26: $vote \leftarrow \text{VOTE}(v, r+1, val, \text{PRE-COMMIT})$
27: $votes[r+1] \leftarrow \{vote\} \cup votes[r+1]$
28: **continue**
▷ COMMIT phase
29: **wait** for $(n-f)$ votes in qc':
30: **if** $\text{LEN}(votes) - 1 > r$ **then**
31: $value' \leftarrow \perp$
32: $qc' \leftarrow \emptyset$
33: **continue**
34: $value \leftarrow value'$
35: $qc \leftarrow qc'$

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

proposed value (line 5) and a partial quorum certificate for that value (line 6).

Consensus is reached in two steps, first a supermajority needs to be reached in the last round of the $votes$, then a supermajority needs to be reached for the next quorum certificate. The first step will establish a resilient quorum, while the second step will guarantee that sufficiently many replicas know that such a quorum has been achieved. The flow of the protocol is shown in Fig. 1.

3.2.1 Proposing new values

To write a new value, a replica has to propose a new value to the other replicas. This process is the `PREPARE` phase in Algorithm 1. The proposing replica adds the new value and its vote to round 0 of $votes$ (line 9). 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 in the next view. The non-proposing replicas will receive the new proposal(s) via the gossip protocol, and also enter into the next phase.

3.2.2 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 (line 13-16). If multiple values have the same number of votes, the lexicographic order of the hash of those values is taken as a tiebreaker. If a round has reached a supermajority of votes for a single value, then no new round can be started anymore, and the replicas will start creating a new quorum certificate (line 20-23). If a supermajority of the replicas have voted in a round, but not a single value reaches a supermajority, a new round is started (line 24-28) and all replicas can vote again in this new round. The replicas are only allowed to vote on the current winner in round 0 according to their local state (line 13-16). Because each replica might have a different state on the current set of votes in round 0, there can still be multiple values in the next round without any supermajority for a single value.

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

Once a replica enters the `COMMIT` phase, it will wait for $n-f$ replicas to also confirm that the proposed value can be committed (line 29). A malicious replica can trick an honest replica to enter this phase without support of enough honest replicas. For this reason, during this waiting period, if the replica observes that other replicas started a new round, it will realise its mistake and remove the partial commit certificate and go back to the `PRE-COMMIT` phase (line 30-33). The malicious replica can also be detected, as there will be two signatures of him signing two votes for two different values in the same round.

If $n-f$ replicas agree and add their vote to the quorum certificate for the next value, the value will be accepted and the quorum certificate will be stored to later convince other replicas that the value is indeed correct (line 35).

3.2.3 Correctness

The integrity and validity properties are trivially satisfied. We can now reformulate the agreement and termination properties more precisely as a safety and liveness property:

Theorem 1 (correctness). Let $\mathfrak{R}$ be a cluster of n replicas with f Byzantine replicas and $n \geq 3f+1$. BeauForT's correctness is defined by the following two properties:

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

We prove that BeauForT satisfies these properties in Appendix A.

3.2.4 State-based replication protocol

During all phases in the algorithm, the state is continuously replicated to the other replicas. The full state, including all votes in the consensus protocol, is replicated by using a state-based gossip protocol. A major feature of gossip-based communication is its reliability [22]. Each time a new state is received, the local state is merged with the remote state. This protocol synchronizes data peer-to-peer using state-based Conflict-free Replicated Data Types (CRDTs) [23] combined with a Merkle-tree [24] to efficiently replicate the updated state, similar to OWebSync [9] or Merkle Search Trees [25]. All key-value pairs are put inside a Merkle-tree. Each key-value pair is a separate instance of the consensus protocol in Algorithm 1. The Merkle tree is used to efficiently replicate the state between any two replicas. A replica will first send its own root hash to another replica. If those hashes are equal, that replica knows that both replicas have the same state, and the gossip protocol ends. If however the hashes are not equal, that replica will descend in the Merkle-tree and send all hashes in the next level of the tree to the first replica. This process continues until a specific key-value pair is reached, and then the full state of the consensus protocol in Algorithm 1 is sent ($value$, qc, v, $votes$, $value'$ and qc'). The state of the protocol can be represented as

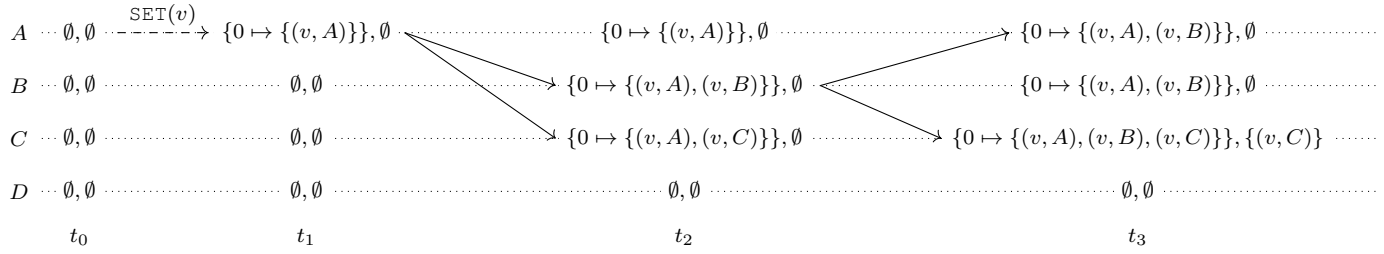

Fig. 2. Example of the state-based synchronization with 4 replicas A, B, C, D. Only the current $votes$ and qc' are shown. Arrows represent a state transfer.

a CRDT: $votes$ and qc' are Grow-only Sets [23], and a state associated with a higher $view$ number overwrites any older state, much similar to a LWWRegister [23]. There are two extra constraints imposed on the CRDTs due to the Byzantine nature. First, signatures have to be correct, no replica may accept any invalid signature, if a replica does send a wrong signature, it can be considered Byzantine, and the other replicas will drop their connection to it. Secondly, not all states are valid. For example, $votes$ keeps track of the different rounds, but no new round can be started unless $n - f$ votes in the previous round are present, and no consensus has been reached yet. When a replica receives an invalid state, it will be ignored, and the other replica can be considered Byzantine. If those $n-f$ votes are all for the same next value, then no new round is started. These constraints, signatures and invalid states, are verified before the CRDTs are merged. By using a state-based approach, rather than the operation-based approach of operation-based CRDTs [23], blockchains [14], or traditional BFT protocols, we only need to store the current state together with some metadata. There is no need to store the full log of all operations to later convince replicas that were temporarily offline of the new state. Replicas also do not need to keep track of the state of other replicas, or which messages are already received by which replica. If a new value and quorum certificate with a higher view are received, then the protocol will accept the new state, and the protocol will reset back to line 3 of Algorithm 1 with that newer view. Note that we do not explicitly show the gossiping in Algorithm 1 to keep the algorithm compact. During the whole protocol, the state is continuously gossiped between the replicas. This way, $votes$ or qc' will eventually contain enough votes to continue in the protocol specification. The state-based replication also helps with the consensus protocol. Instead of only sending proposals and decisions to other replicas, the full state of $votes$ and qc' is sent. This approach allows replicas to hold each other accountable when they cast their vote. Their $votes$ should support why they voted for a specific value, otherwise they will be considered Byzantine and excluded from the network.

3.2.5 Examples

An example of this replication process is shown in Fig. 2. There are four non-Byzantine replicas with an empty set of $votes$ and empty qc' at t_0. The scenario starts at t_1 with replica A proposing a new value v (line 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_2, and those replicas merge the received state with their local state. Since B and C did not yet vote in this view and round, they will cast their vote for the current winning value (line 10-15 of Algorithm 1). This process continues at t_3 when replica B sends its state to replica A and C. At t_3, replica C observes that a supermajority of the replicas support value v, and it starts working on a new quorum certificate to determine if at least a supermajority of the replicas also knows about this (line 17-21 of Algorithm 1).

Imagine now the same four non-Byzantine replicas. Replica A again proposes a new value v_1, but concurrently replica B proposes another value v_2. If we use the same gossiping path as in Fig. 2, then at t_2 replica B and C receive the vote from replica A. Replica B will not vote anymore, because it already voted for his own value v_2. At t_3, replica B gossips its state to replica A and C. Replica A will now have one vote v_1 (his own) and one vote for v_2 (from B). Replica C however will now have two votes for v_1 (from A and C) and one vote for v_2 (from B). Since replica C now has $n - f = 3$ votes in round 0, but there are only two votes for the winning value, it will start a new round and vote for the winning value in $votes[0]$, which is v_1. B will now also vote for v_1 in $votes[1]$ and a commit certificate can be created after the round 1.

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

Since replicas will vote for the first value they observe, a well-placed replica that is able to send its request to enough other replicas first is able to prevent requests from other replicas from ever being accepted. This does satisfy the liveness constraint that was specified formally in Section 3.2.3: in which we specify that when new values are proposed, some value should be eventually accepted. The protocol does not provide deterministic fairness, i.e., no guarantees are made for a single proposed value. In practice, we have two arguments in favor of our model. First, when a replica notices that no progress is being made on a proposal, it will close some connections randomly and open new connections to other replicas. This makes it much harder for such a well-placed replica to be well-placed for a long time. Second, our use-case of loyalty points across small-scale

merchants prevents any problems because only the client (customer) is able to sign a message to spend loyalty points at a certain merchant. In this case, only a single proposal will ever be present if the client is honest, and it will always be eventually accepted by the network.

3.2.6 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 BeauForT will use a fast path and delay the verification of any incoming signatures. BeauForT will just accept and replicate them, until a decision needs to be made, such as starting a new round or starting to create a new proposed quorum certificate. Only then, all signatures will be verified in one batch. If all signatures are valid, the protocol can continue as normal. If there are invalid signatures, then those will be removed and BeauForT will continue to collect more signatures and verify them on arrival. This hybrid approach enables very fast consensus when all replicas are honest, while gracefully degrading to a slower, more costly protocol that can detect which replicas are actively acting Byzantine.

4 Architecture and Implementation

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

4.1 Overall architecture

The BeauForT middleware architecture consists of five main components (Fig. 3): (i) a *public interface* that offers an API for developers, (ii) a *peer-to-peer network* component to communicate directly with other browsers, (iii) a *consensus* component to handle the consensus protocol described in the previous section, (iv) a *membership* component to handle all cryptographic operations, and (v) a *store* component to save all state to persistent storage. The last three components run on a different browser thread by using Web Workers.

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

Apart from those functions, the middleware also provides a constructor function to initialize the middleware by passing the following four configuration parameters: the list of all members of the network together with their public key, the private key of the replica, the URL to the signaling server to set up the peer-to-peer connections, and 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. At startup, every replica will connect to some other replicas randomly. In our implementation, a connection will be made to at least seven other replicas. This number is arbitrarily, but performed best in our experimental evaluation. A higher number will increase resource usage, and decrease the potential to batch multiple updated states together. A lower number will increase the number of hops, and therefor increase the latency. To defend against an eclipse attack, where few Byzantine neighbors try to surround an honest replica to break liveness, a replica can periodically create new connections to other peers and drop older connections when no updates are being gossiped to them, or when proposals are not being voted on. This is similar on how Bitcoin works [14].

(iii) Consensus. The *Consensus* component handles the consensus protocol described in Section 3. It maintains a Merkle-tree of all key-value pairs and uses the state-based CRDT framework OWebSync [9] 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 [17]. Section 4.2 provides more details about the BLS implementation. It is implemented in C and compiled to WebAssembly.

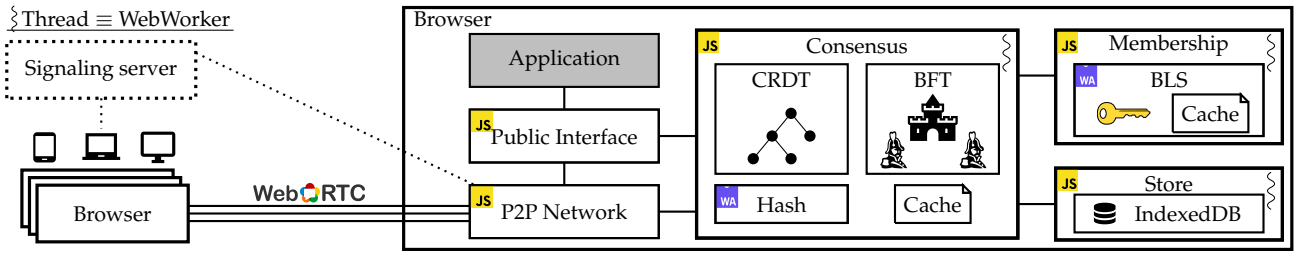

Fig. 3. Browser-based architecture of BeauForT.

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

1) *Parameters Generation:* $\mathsf{PGen}(\kappa)$ sets up a bilinear group $(q, \mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_t, e, g_0, g_1)$ as described by [26]. e is an efficient non-degenerating bilinear map $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_t$. g_0 and g_1 are generators of the groups $\mathbb{G}_0$ and $\mathbb{G}_1$. It outputs $params \leftarrow (q, \mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_t, e, g_0, g_1)$.
2) *Key Generation:* $\mathsf{KGen}(params)$ is a probabilistic algorithm that 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. 4. Formal specification of the optimized BLS signature scheme.

(v) Store. At last, the *Store* component saves all state to the IndexedDB database. IndexedDB is a key-value datastore built inside the browser. Each value and the Merkle-tree are serialized to bytes and stored there under the respective key. This enables users to close the browser and continue afterwards 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) [17] 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 [27] 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 [28] and HotStuff [12] also use aggregate or threshold signatures. However, they use it in a different way. They let the leader compute the aggregate signature. BeauForT uses a different approach, once a proposed quorum certificate has reached a supermajority of the votes, any replica can aggregate these into one single aggregated BLS signature. BeauForT makes a trade-off between performance, bandwidth and storage space. Verifying a single signature is expensive, however, aggregation is cheap in performance. For this reason, BeauForT will delay the verification of the signatures until the latest possible moment (as explained in Section 3.2.6). Only then the individual signatures are aggregated and verified. If the verification fails, a binary search can be conducted to find the invalid signatures and remove them. This leads to a higher bandwidth usage, compared to always aggregating two shares immediately. But allows for cheaper recovery when a Byzantine replica is sending invalid signatures. Once a signature is aggregated and verified, the individual shares are discarded, saving both bandwidth and storage space.

The standard scheme is vulnerable to rogue public key attacks. The state-of-the-art approach [26] 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 [26], and we obtain more efficient aggregations at scale. We provide the mathematical background and formal specification of the optimized BLS scheme in Fig. 4.

5 Evaluation

We validated the BeauForT middleware with the loyalty points use case presented in Section 2. The first subsec-

tion presents this validation. Next, we present three different benchmarks with different scales. The first benchmark shows the performance results in the optimistic scenario without network failures or Byzantine failures. The second benchmark evaluates the performance in a more realistic scenario with some network failures. The last benchmark evaluates the performance in the presence of a Byzantine replica.

5.1 Validation in the loyalty points use case

The deployment of the loyalty points use case consists of three services: a web application running in a browser for each merchant, a web server to serve the static web application files, and a signaling server to set up WebRTC peer-to-peer connections between the browsers. The web server is optional. Every merchant can also store those application files themselves and load them from their local file system. The signaling server is a trusted component. However, if trust is not present, you can set up multiple signaling servers to reduce potential misbehavior. No actual data is sent to the signaling server. It is only used to discover other peers on the network. To have a baseline, we compare BeauForT to two other existing state-of-the-art systems for BFT consensus: BFT-SMaRt [11], [29] and Tendermint [13], [30]. BFT-SMaRt is a more traditional BFT protocol, similar to PBFT [31], where all replicas are connected to each other, and one leader drives the protocol. If that leader fails, a new one will have to be elected before any progress can be made. Tendermint uses gossip for communication between the replicas. There is still a leader, however, that leader changes frequently.

5.2 Test setup

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

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

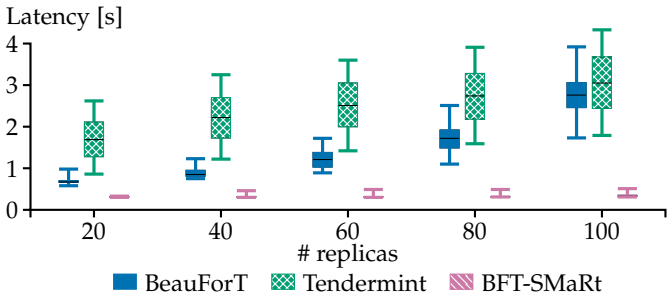

Fig. 5. Latency in the optimistic scenario without failures.

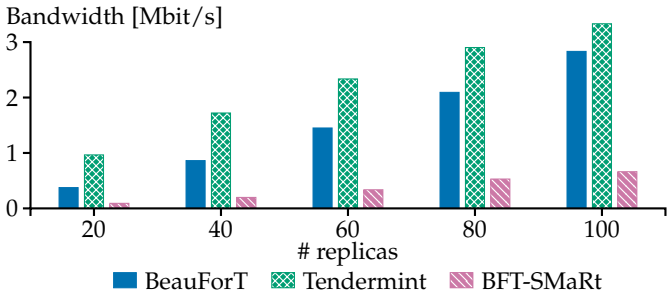

Fig. 6. Network usage in the optimistic scenario without failures.

5.3 Optimistic scenario

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

Fig. 5 shows the latency for the different technologies. For the use case of loyalty points, transactions must be confirmed fast, as people are waiting at checkout to receive or redeem loyalty points. BeauForT can confirm transactions within 4 seconds, even with a network of one hundred browsers. BFT-SMaRt can confirm transactions within half a second. This is because all replicas communicate directly with each other. However, having all replicas directly connected to each other is not realistic in a mobile peer-to-peer network. In contrast, BeauForT and Tendermint use gossip and need multiple hops before all replicas are reached. This also causes the increased latency. Furthermore, BFT-SMaRt uses HMAC to authenticate requests, which are an order of magnitude faster than the asymmetric signatures used in BeauForT and Tendermint. We can see a similar pattern in the bandwidth requirements shown in Fig. 6. In the large-scale scenario with 100 browsers, BeauForT uses less than 3 Mbit/s, which is acceptable for a typical mobile network.

5.4 Realistic scenario

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

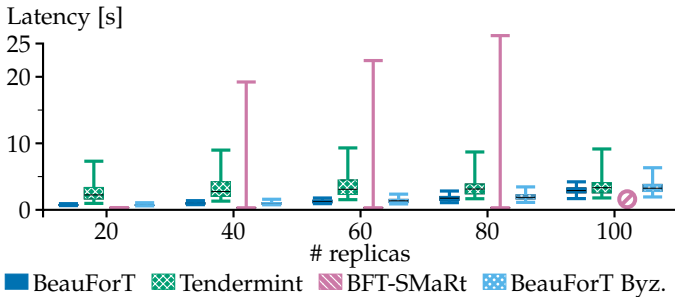

Fig. 7. Latency in the realistic scenario with network failures. For BeauForT we included an extra scenario in which a Byzantine replica tries to halt the network. For BFT-SMaRt only the upper whisker is visible, the box is situated at the bottom.

Fig. 7 shows the latency in this scenario. BeauForT is not impacted much by the failing replicas and can still confirm transactions within 5 seconds. The impact on Tendermint is also small, but the tail latency is doubled to about 10 seconds. BFT-SMaRt however needs to use a costly leader election protocol when the current leader fails. This process takes some time, during which no transaction can be committed. Once a leader is chosen, the same fast performance can be achieved again. This behavior is clearly visible in Fig. 7. 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 [25]. 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.

5.5 Byzantine scenario

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

The latency in this Byzantine scenario is shown in Fig. 7. BeauForT can handle Byzantine replicas very well for smaller networks, however, for networks of size 100 replicas, the tail latency becomes 7 seconds. Which might already be quite high for the use case of loyalty points. This is mostly due to the cost to verify more BLS signatures. We did not test the effect of Byzantine replicas for BFT-SMaRt or Tendermint. As they do not use a fast path when everyone is honest, the impact is less. However, if the current elected leader happens to be Byzantine, it can delay the consensus until some timers end and a new leader is elected [32].

5.6 Discussion and conclusions

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

6 Related work

Several client-side frameworks for data synchronization between web applications exist: Legion [5], Automerge [7], and OWebSync [9]. They make use of various kinds of Conflict-free Replicated Data Types (CRDTs) [23] 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 [33], [34]. Recent work [35], [36] also looked into making CRDTs Byzantine fault-tolerant in the eventually consistency model. BeauForT provides strong consistency.

Permissioned blockchains such as Hyperledger Fabric [37] have closed membership and often use a BFT consensus protocol to order transactions. For example BFT-SMART in HyperLedger Fabric [11], [29]. The first known BFT protocol is Practical Byzantine Fault Tolerance (PBFT) [10]. Other protocols bring improvements to the original PBFT protocol. Zyzzyva [38] 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 [39] provides an abstraction for these BFT algorithms. These protocols are targeting a small number of replicas in a local network. They generally work in two phases: the first guarantees proposal uniqueness, and the second guarantees that a new leader can convince replicas to vote for a safe proposal. HotStuff [12] 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 [32]. GeoBFT [40] is a topology-aware, decentralized consensus protocol, designed for geo-distributed scalability. AWARE [41] is a variant of BFT-SMaRt that dynamically changes the voting power of a replica depending on its latency over time, decreasing the consensus latency. BeauForT gives every replica equal voting power. In future work, BeauForT could be extended to associate a weight to each vote. While we believe this would be especially beneficial for our target environment with mobile and unreliable clients, special care will have to be given to ensure safety will stay intact. BeauForT does not use a leader and replicas communicate only to a subset of the other replicas using a gossip-like protocol.

WebBFT [42] shares a similar vision of client-centric, decentralized web applications. However, they only interface to a backend BFT-SMaRt cluster, instead of running the BFT protocol directly between browsers. Similarly, earlier work [43] extended the Web Services Atomic Transactions specification to include BFT. However, also here the protocol is running between the backend servers, rather then between the actual web clients.

Tendermint [13], [30], 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 [44], HoneyBadger [45], Dumbo [46] and BEAT [47] use distributed coin flipping for consensus. HoneyBadger [45] uses threshold encryption [31] for censorship resilience. Algorand [48] uses Verifiable Random Functions [49] to select a random committee for the next round. Avalanche [16], [50] uses meta-stability to reach consensus by sampling other replicas without any leader. While Avalanche is lightweight and scalable, it needs to be able to sample all other validators directly. The number of connections one can open in a browser without performance loss is limited. BeauForT supports propagation of votes over multiple hops.

Several BFT consensus protocols use a leader-less approach. Although most determinstic BFT consensus protocols designate a special leader, there exist determinstic protocols that are fully leader-free [51]. However, the algorithm only terminates in $f+3$ rounds in the best case, even without failures. [52] provides a leaderless algorithm that is optimal, and also provides a fast path in good conditions. It assumes replicas can directly broadcast to every other honest replica. A hybrid approach is also possible, DBFT [53] uses a so-called weak-coordinator which is not required to reach consensus, but can speed up consensus when this weak coordinator is honest. Messages are broadcasted to every other replica. Our protocol only maintains the state of the protocol, and state-changes are gossiped by dynamically computing a diff using the Merkle-tree. This naturally allows to batch multiple votes and state changes in a single network request.

There are several proposals to improve the performance and response time of BFT consensus. StreamChain [54] reaches consensus over a stream of transactions instead of blocks. FabricCRDT [55] 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 [25] supports commuting transactions to be applied out-of-order. A novel design for gossip in Fabric [56] improves the block propagation latency and bandwidth. Other approaches dynamically adapt the number of faults the system can withstand in reaction to threat level changes [57]. While these improvements make BFT faster, none of them try to reduce the infrastructure requirements to be able to easily set up an untrusted peer-to-peer network.

Open or permissionless blockchains such as Bitcoin [14] and Ethereum allow everyone to participate and use Proof-of-Work (PoW) to reach agreement over the ledger. However, PoW has several flaws [58]. PoW uses a lot of processing power and energy [59] 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 [14] for clients can reduce the resource usage at the cost of decentralization.

ByzCoin [60] 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) [61] 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 multi-signature and assumes a synchronous network.

The Lightning Network or state channels for Bitcoin [62] or Ethereum [63], [64] 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 [65]. BeauForT uses a similar state-transitioning protocol where only the latest collectively agreed state needs to be stored. However, BeauForT can tolerate both failing and malicious replicas, without resorting to a blockchain or a trusted third party.

Another approach is to use a trusted hardware component [66], [67], [68], [69], [70], [71]. These are faster and less computationally intensive but require specialized hardware to be present. Moreover, trusted execution environments have been broken in the past [72], [73].

7 Conclusion

In this paper, we presented BeauForT. A browser-based middleware for decentralized, community-driven web applications. BeauForT uses a client-centric, leaderless BFT consensus protocol, combined with a robust and efficient state-based synchronization protocol. BeauForT uses an optimized BLS scheme for efficient computation and storage of signatures. It supports a client-centric, browser-based, state-based, permissioned datastore with a low infrastructure and storage footprint for small-scale, citizen-driven networks. Compared to other state-of-the-art protocols, BeauForT offers consistent and robust confirmation times to achieve finality of transactions in the order of seconds, even in failure settings and Byzantine environments. In optimal environments, with no crashes or Byzantine failure, a leader-based protocol confirms transactions faster than BeauForT. In contrast to traditional blockchains, BeauForT does not store a transaction log or blockchain, keeping the overall storage footprint small.

References

[1] 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, 2015.

[2] 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. ACM, 2019.

[3] 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. SciTePress, 2019.

[4] 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. ACM, 2019.

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

[6] M. Kleppmann and A. R. Beresford, "A conflict-free replicated json datatype," *IEEE Transactions on Parallel & Distributed Systems*, vol. 28, no. 10, 2017.

[7] M. Kleppman and A. R. Beresford, "Automerge: Real-time data sync between edge devices," in *1st UK Mobile, Wearable and Ubiquitous Systems Research Symposium*, ser. MobiUK '18, 2018.

[8] M. Kleppmann, D. P. Mulligan, V. F. Gomes, and A. R. Beresford, "A highly-available move operation for replicated trees," *IEEE Transactions on Parallel & Distributed Systems*, vol. 33, no. 07, 2022.

[9] K. Jannes, B. Lagaisse, and W. Joosen, "OWebSync: Seamless synchronization of distributed web clients," *IEEE Transactions on Parallel & Distributed Systems*, vol. 32, no. 9, 2021.

[10] 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. USENIX, 1999.

[11] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with BFT-SMART," in *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '14. IEEE, 2014.

[12] 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*, ser. PODC '19. ACM, 2019.

[13] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," 2018.

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

[15] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," ethereum.org, White paper, 2013.

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

[17] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *Advances in Cryptology*, ser. ASIACRYPT 2001. Springer, 2001.

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

[19] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, 1988.

[20] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011.

[21] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: A dag-based mempool and efficient bft consensus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. ACM, 2022.

[22] D. Cason, N. Milosevic, Z. Milosevic, and F. Pedone, "Gossip consensus," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. ACM, 2021.

[23] M. Shapiro, N. Perguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, ser. SSS'11. Springer, 2011.

[24] R. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology*, ser. CRYPTO '87. Springer, 1988.

[25] 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*, ser. SRDS '20. IEEE, 2020.

[26] D. Boneh, M. Drijvers, and G. Neven, "Compact multi-signatures for smaller blockchains," in *Advances in Cryptology*, ser. ASIACRYPT 2018. Springer, 2018.

[27] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *Advances in Cryptology*, ser. EUROCRYPT 2003. Springer, 2003.

[28] 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*, ser. DSN '19. IEEE, 2019.

[29] 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*, ser. DSN '18. IEEE, 2018.

[30] D. Cason, E. Fynn, N. Milosevic, Z. Milosevic, E. Buchman, and F. Pedone, "The design, architecture and performance of the tendermint blockchain network," in *40th International Symposium on Reliable Distributed Systems*, ser. SRDS '21. IEEE, 2021.

[31] V. Shoup, "Practical threshold signatures," in *Advances in Cryptology*, ser. EUROCRYPT 2000. Springer, 2000.

[32] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, "RBFT: Redundant byzantine fault tolerance," in *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ser. ICDCS '13. IEEE, 2013.

[33] 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, 2020.

[34] 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*, ser. ICDCN '21. ACM, 2021.

[35] M. Kleppmann, "Making crdts byzantine fault tolerant," in *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '22. ACM, 2022.

[36] K. Jannes, B. Lagaisse, and W. Joosen, "Secure replication for client-centric data stores," in *Proceedings of the 3rd International Workshop on Distributed Infrastructure for Common Good*, ser. DICG '22. ACM, 2022.

[37] 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 sys-

tem for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. ACM, 2018.

[38] 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. ACM, 2007.

[39] 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, 2015.

[40] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, "ResilientDB: Global scale resilient blockchain fabric," *Proc. VLDB Endow.*, vol. 13, no. 6, 2020.

[41] C. Berger, H. P. Reiser, J. Sousa, and A. Bessani, "Resilient wide-area byzantine consensus using adaptive weighted replication," in *38th Symposium on Reliable Distributed Systems*, ser. SRDS '19. IEEE, 2019.

[42] C. Berger and H. P. Reiser, "WebBFT: Byzantine fault tolerance for resilient interactive web applications," in *Distributed Applications and Interoperable Systems*. Springer, 2018.

[43] L. E. Moser, H. Zhang, W. Zhao, P. Melliar-Smith, and H. Chai, "Trustworthy coordination of web services atomic transactions," *IEEE Transactions on Parallel & Distributed Systems*, vol. 23, no. 08, 2012.

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

[45] 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. ACM, 2016.

[46] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous bft protocols," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. ACM, 2020.

[47] 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. ACM, 2018.

[48] 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. ACM, 2017.

[49] S. Micali, S. Vadhan, and M. Rabin, "Verifiable random functions," in *40th Annual Symposium on Foundations of Computer Science*, ser. FOCS '99. IEEE, 1999.

[50] T. Rocket, "Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies," avalabs.org, White paper, 2018.

[51] F. Borran and A. Schiper, "A leader-free byzantine consensus algorithm," in *Distributed Computing and Networking*, ser. ICDCN 2010. Springer, 2010.

[52] K. Antoniadis, A. Desjardins, V. Gramoli, R. Guerraoui, and I. Zablotchi, "Leaderless consensus," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, 2021.

[53] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, "Dbft: Efficient leaderless byzantine consensus and its application to blockchains," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, 2018.

[54] 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. ACM, 2018.

[55] 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. ACM, 2019.

[56] 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*, ser. ICDCS '20. IEEE, 2020.

[57] D. S. Silva, R. Graczyk, J. Decouchant, M. Völp, and P. Esteves-Verissimo, "Threat adaptive byzantine fault tolerant state-machine replication," in *40th International Symposium on Reliable Distributed Systems*, ser. SRDS '21. IEEE, 2021.

[58] 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. ACM, 2018.

[59] 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. IET, 2014.

[60] 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. USENIX, 2016.

[61] 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*, ser. S&P '16. IEEE, 2016.

[62] 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. ACM, 2019.

[63] 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*. Springer, 2019.

[64] 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*. Springer, 2020.

[65] 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. ACM, 2019.

[66] 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, 2013.

[67] 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. ACM, 2012.

[68] T. Distler, C. Cachin, and R. Kapitza, "Resource-efficient byzantine fault tolerance," *IEEE Transactions on Computers*, vol. 65, no. 9, 2016.

[69] 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. USENIX, 2017.

[70] 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. ACM, 2017.

[71] 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, 2018.

[72] 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*, ser. USENIX Security '18. USENIX, 2018.

[73] 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)*. IEEE, 2019.

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 Prof. 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 PostDoc researcher in the Department of Computer Science and Electrical Engineering at KU Leuven in Belgium, and a member of the research groups imec-DistriNet and imec-COSIC. He received his PhD degree in computer science from KU Leuven in 2021. His main research interests are in the area of applied cryptography in general, and in particular computing on encrypted data.

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

Wouter Joosen is 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.

APPENDIX A
SAFETY AND LIVENESS

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.

A.1 Safety

Lemma 1 (Safety). 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 quorum certificates qc_1 for value $value_1$ and qc_2 for value $value_2$ at view v, then $value_1 = value_2$.

We will first prove this for the simplified case when both quorum certificates belong to the same round, and we will then prove that once a quorum 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 quorum certificates qc_1 and qc_2 for value $value_1$ and $value_2$ respectively with $qc_{1\ view} = qc_{2\ view}$ and $qc_{1\ round} = qc_{2\ round}$, then $value_1 = value_2$.

Proof: Assume two different replicas R_1 and R_2 have constructed a quorum certificate qc_1 and qc_2 for value $value_1$ and $value_2$ respectively with $qc_{1\ view} = qc_{2\ view}$ and $qc_{1\ round} = qc_{2\ round}$. They are constructed in the same round, so of the n possible votes, at least $n - f$ replicas have voted on $value_1$, and at least $n - f$ replicas have voted on $value_2$. Honest replicas will never vote twice in the same view and round. Therefore, at least $n - 2f$ honest replicas have voted on $value_1$ and $n - 2f$ *different* honest replicas have voted on $value_2$. In total, we have $(n - 2f) + (n - 2f) + f \equiv 2n - 3f$ replicas that have voted. We defined $n \geq 3f + 1$ before, which gives $2n - 3f \geq 3f + 2 \geq n + 1$ 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 $value_1$ and $value_2$ have to be equal. □

Lemma 3. If replicas $R_1, R_2 \in \mathfrak{R}$ are able to construct quorum certificates qc_1 and qc_2 for value $value_1$ and $value_2$ respectively with $qc_{1\ view} = qc_{2\ view}$, then $qc_{1\ round} = qc_{2\ round}$.

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

A.2 Liveness

When a new value is proposed, eventually the protocol will end and a valid quorum certificate is created for a new value. This value is not necessarily the first proposed value, and it is not even guaranteed that a specific value ever gets committed as long as other values continue to be proposed. Safety is always chosen over liveness. When there are not enough honest replicas online to reach a supermajority, no consensus can be reached and the protocol will simply block and wait for more votes. However, all those replicas do not need to be online at the same time, since the state is replicated to all available replicas over time, and votes can be verified by all replicas in the end.

Lemma 4 (Liveness). Let $\mathfrak{R}$ be a cluster of n replicas with f Byzantine nodes and with $n > 3f$. If an honest replica $R \in \mathfrak{R}$ proposes a new value at view v, eventually a replica will be able to construct a quorum certificate qc for some value at view v.

Lemma 5. If only a single replica $R \in \mathfrak{R}$ proposes a new value $value_1$, eventually a replica will be able to construct a valid quorum certificate qc.

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

Lemma 6. If x replicas $R_{1..x} \in \mathfrak{R}$ propose values $value_{1..x}$, and no Byzantine replicas vote twice in the same round, eventually a replica will be able to construct a valid quorum certificate qc.

Proof: Either a single value reaches a quorum, in which case the previous lemma holds. Or a split vote occurs and a new round will be started after $n - f$ votes are observed. All replicas will base their vote for this new round on the winning value that they observed from round 0. At least $n - f$ votes are known, and only f votes are still unknown. "Known" means known to the one replica that is making some decision and going ahead in the protocol. But to make progress, at least $n - f$ replicas need to know about $n - f$ votes. These votes that are known, are not necessarily the same for all $n - f$ replicas, but eventually, all honest replicas will know about the exact same votes. As

long as not all votes are known to all voting replicas, the winning value might change. In each new round, either all unknown votes stay unknown, or one becomes known. In the former case, then the currently known votes will all be the same, and a proposed quorum certificate can be started. In the latter case, one extra vote is known, which might again result in the system ending up in a split vote, and a new round will be started. However, this last case can only happen at most f times. After $f + 1$ rounds, all replicas will have voted in round 0, and every replica will observe the same winning value, and a quorum certificate can be created. □

Lemma 7. If x replicas $R_{1..x} \in \mathfrak{R}$ propose values $value_{1..x}$, eventually a replica will be able to construct a valid quorum certificate qc.

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