You Don't Need a Ledger: Lightweight Decentralized Consensus Between Mobile Web Clients

Kristof Jannes
imec-DistriNet, KU Leuven
kristof.jannes@cs.kuleuven.be

Bert Lagaisse
imec-DistriNet, KU Leuven
bert.lagaisse@cs.kuleuven.be

Wouter Joosen
imec-DistriNet, KU Leuven
wouter.joosen@cs.kuleuven.be

Abstract

Centralized systems relying on a trusted third party are being replaced by decentralized systems using proof-of-work blockchains to reach consensus between multiple mistrusting parties. Due to the high energy usage of such systems, many solutions using a Byzantine fault-tolerant algorithm to reach consensus have emerged. While those systems solve the energy and safety concerns of proof-of-work blockchains, they still require you to store the full ledger and need a complex backend infrastructure to get started.

This paper presents a lightweight middleware running in the browser, designed for small businesses and end-users unable to set up a complex private blockchain business network. The middleware for consensus runs entirely in the browser, with only a small server-side component used for the peer-to-peer discovery. We achieve fast confirmation times while guaranteeing safety and liveness for honest users. We also do not keep a ledger, reducing the overall storage footprint.

1 Introduction

Web applications are beneficial compared to native programs as they do not require installation and can be updated by simply serving the new source code. Moreover, they can run on all kind of devices, even on mobile phones. Therefore, we expect that the browser will become the platform of choice to deploy applications [9]. However, current peer-to-peer (P2P) data-synchronization systems for the browser like Legion [19] and Yjs [14] focus on full replication and consistency, rather than security. They allow users to modify all data and lack Byzantine fault-tolerance (BFT). BFT means that a system can both tolerate crash failures (i.e. a node that goes down or sends erroneous data), as well as malicious nodes (who actively try to attack the system).

Traditionally, consensus is often achieved using a centralized trusted party. While this is beneficial for performance, too much power is given to one entity, who can decide to manipulate the consensus and charge high transaction costs. Because trust is not always present, one can opt for a more decentralized consensus, where several mistrusting parties are all responsible for the consensus. Starting with Bitcoin [13], many proof-of-work (PoW) blockchains have emerged. However, they are too slow for many use-cases. Bitcoin needs about one hour to confirm a transaction with high probability. Moreover, PoW needs a lot of processing power and energy which are not readily available on mobile devices. They also store an immutable ledger on every device, leading to large storage overhead. Lightweight clients that use a proxy node to communicate with the blockchain do exist, but someone still needs to set up the full-node and you need to trust it. Another type of blockchain uses a BFT consensus protocol. E.g. Hyperledger Fabric [1] can use PBFT [5] or BFT-SMART [3] and achieves high throughput and low latency. However, these systems require a complex backend infrastructure, with many different servers, and still need to store the full ledger.

The contribution of this paper is a lightweight middleware for decentralized consensus that can be used by mobile clients in their web browser. The middleware is designed to function between small businesses without the infrastructure and capital to set up a private permissioned blockchain, and without the trust in a trusted third-party. The middleware:

1. tolerates both crash failures as well as malicious nodes,
2. guarantees consensus finality once a decision is made,
3. uses an efficient state-based replication protocol to propagate updates and votes through the P2P network,
4. and is designed for lightweight setups, with only one backend component, used for the P2P discovery.

By using a state-based protocol, there is no need to keep an operation log to synchronize offline clients.

This paper first presents the motivation using two industrial case studies and lists the requirements in Section 2. Section 3 explains the architecture and the consensus protocol. Section 4 shows the preliminary evaluation. We discuss related work in Section 5 and conclude in Section 6.

2 Motivation and requirements

In general, the middleware is designed to set up a network between mutually mistrusting parties (called participants), who want to offer integrated services to their customers. This section first describes two (anonymized) industrial case studies. Then it states the generic requirements for our middleware. At last, it discusses the adversary model.

eLoyalty. eLoyalty offers white-label software for loyalty programs. This is normally deployed as a client-server system, where the backend of eLoyalty handles all requests from the client systems deployed at the customer's premises. Multiple companies can use this system to integrate their loyalty network. This model works well for large companies, but smaller local stores are often not able to participate due

to the high legal burden and monetary cost to set up such a consortium. Decentralization is especially important for small stores in emerging markets, which lack trust in centralized large corporations or the government. This use-case consists of several small stores who want to create an integrated loyalty network, where customers can redeem their points at any participant. Store A can award a customer with some points, which they can redeem at any other store B of the consortium. Later, store A has to pay back store B for the reward given to the customer. There are two misuse cases. First, the customer can try to use its loyalty points at multiple places, leading to the classic double-spending problem. Second, store A can refuse to pay back store B.

eLoans. eLoans is an integrated network of banks (the participants) that offer loans to companies using unpaid invoices as collateral. A company can use an unpaid invoice at any bank of the network, but the bank where the invoice will be paid need to verify that it isn't already paid yet. The use-case has four misuse cases. A company can use the same invoice twice or use a fake one. The banks can try to boycott each other by not verifying an invoice or can collude with companies to let other banks accept fake invoices.

Requirements. This section lists four general requirements, based on those two case studies. First, double-spending of points or invoices (in general: *tokens*) needs to be prevented, because the customers and the participants do not trust each other. Second, different participants cannot cause any harm to each other since they also do not fully trust each other. Third, the solution needs to be decentralized, with no central point of failure or trust. At last, the solution needs to be lightweight, for use in mobile environments, e.g. running on a mobile device at the local market. Therefore, it needs to be both energy and storage efficient. The network needs to be easy to set up, without a complex backend infrastructure.

Adversary model. Since we use a BFT protocol, only $\frac{1}{3} \times n - 1$ of the n participants of the network can be malicious or controlled by an attacker, which is the best you can do for asynchronous Byzantine agreement [4]. They may, however, collude and coordinate their attack. Furthermore, we assume that no attacker can delay or interrupt the network forever: eventually, some stream of messages needs to come through. Attackers also cannot control the signaling and TURN server (used for the P2P discovery). They cannot break the used signatures (elliptic curve P-256 and SHA-256) or find collisions for the used hash function (SHA-256). They cannot modify the encrypted WebRTC messages (built-in into the browser).

3 Middleware for lightweight consensus

This section first explains the web-based architecture of the middleware with its lightweight setup. Next, it explains how tokens are issued and how consensus is reached. It ends with the properties and trade-offs of the middleware.

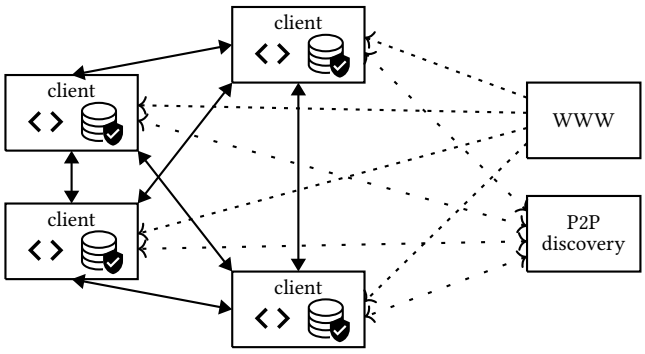

Figure 1. Overall architecture of the web middleware.

3.1 Middleware architecture and data structures

The middleware is designed to run in web browsers on mobile devices, with little server-side infrastructure. Figure 1 shows the architecture. There are two server-side components required. The first is used for the P2P discovery. It implements a signaling protocol and TURN service [9] and is only used to bootstrap the P2P connections. The second is the webserver (WWW) which serves the static resources to the browsers. This requirement can be removed when all clients have the required files stored on disk.

The client-side middleware consists of a JavaScript library that runs into the browser and includes all logic for the consensus protocol between the clients. The clients also replicate the full database locally. The middleware uses signed data structures to protect against unauthorized modifications. The signatures make sure a token cannot be forged. But to make sure that the owner cannot redeem the same token twice at different partners, the nodes first vote where the token can be redeemed. The partner only accepts a token and gives the real-life reward when a majority of the nodes agree.

As tokens we use *tickets*. Tickets can only be used once: they get issued and can be redeemed later. In contrast to coins, used by most blockchains, where you can transfer ownership to another user, instead of destroying the token. By using tickets, we don't need to keep track of who owns which coin, which is typically tracked via a distributed ledger. Furthermore, the state is synchronized using a state-based replication protocol. Thanks to those two improvements, our solution does not require any kind of distributed ledger.

State-based replication protocol. Between the nodes, data is exchanged via a state-based replication protocol which only keeps track of the current state and some meta-data. Merkle-trees are used to quickly discover changes, as in Dynamo [6]. By using a state-based approach, rather than the classical operation-based approach used in most P2P networks and blockchains, we avoid the need to store the whole operation log forever. We also get batching out-of-the-box, since multiple changes can be synchronized together. Operation-based approaches need to keep track of who still

needs to receive which operations, requiring solutions like sequence numbers, version vectors or a total ordering.

After some time, larger than the maximum time it will take to replicate the changes across all nodes, the redeemed tokens can be removed from the database. The involved parties might keep a copy of the data with the signatures outside the datastore, but for everybody else, there is no use to keep track of them. This ensures that the total storage required only grows with the number of valid tokens which are not yet redeemed. In contrast to most blockchains which keep a ledger and grow with the total number of transactions.

Membership. The protocol is designed for a closed group of partners who have a digital identity (public key), where the partners know who the other parties are. New members can be added to the network, if the other partners agree, using the same consensus protocol as described in the next section, except that there is a manual user interaction required to vote. The existing members need to review the new member, and only when they agree to add the new member, they will vote to accept him. When $\frac{2}{3} \times n + 1$ members agree, the new member is considered part of the consortium and can issue tokens and place votes. The manual verification of partners is important in our two case studies since a token often represents a promise from one partner to the others. E.g. a promise that a certain invoice is valid and not yet paid.

To make sure a change in the membership does not endanger the safety guarantees, members can only vote for tokens that are issued after they are accepted. When a token is issued, it includes the identifiers of the members who are allowed to vote for it, so every node always knows when the required $\frac{2}{3} \times n + 1$ is reached (this number is fixed for each token), regardless of membership changes in the meantime.

3.2 Consensus protocol

The consensus protocol has two parts. First, a token is given to a customer by a node of the consortium (Figure 2). Second, the customer redeems that token to gain something with a different node (Figure 3). This requires distributed consensus to protect against double-spending.

Protocol to issue tokens. The customer asks a node for a token (Figure 2, step 1). This happens when a customer buys something and gets rewarded with loyalty points, or when a customer asks their bank to verify an invoice. The customer also provides its public key, so that they can later prove ownership of the token with their private key when the customer wants to redeem the token.

The node verifies the provided information and decides to issue a new token representing some value. The token is stored in the local datastore, and its unique ID is sent back to the customer (Figure 2, step 2). It also contains a timestamp in the future, great enough to allow all nodes to replicate the token to their local datastore. The timestamp is only used to prevent that a redeemed and removed token can be added again to the datastore on a later point by a malicious node. Once a token is added to the local datastore, it stays valid forever, regardless of the timestamp, until it is redeemed.

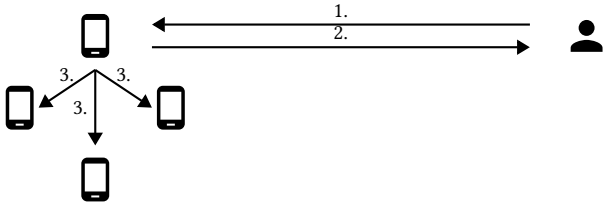

Figure 2. Part 1: protocol to issue tokens.

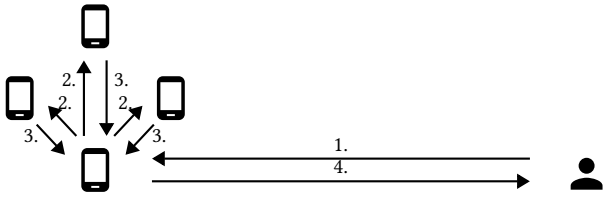

Figure 3. Part 2: protocol to redeem tokens.

Later, when the node is online, the new token is synchronized between the other nodes (Figure 2, step 3). Once the token is replicated, part 1 is completed and the customer can now redeem the token at any node.

Protocol to redeem tokens. In part 2, the customer can go to a different node and ask to redeem its token. The customer sends a request to the node, together with the ID of the token and the ID of the node, signed by its private key (Figure 3, step 1). Because this request is signed, the node is sure that the customer is the owner of the token and it can proof to the other nodes that the customer gave permission to redeem it.

The node verifies the signature. If the token is not present in the local datastore, the node does not accept it and the protocol is aborted. If the token is already redeemed, the customer is trying to commit fraud, and the protocol is aborted. Otherwise, the node proposes itself as a candidate to redeem it, using the request from the customer as proof that this node is allowed to do this. So, no node can start a vote without the cooperation of the customer. The node then votes for itself and stores the signed vote in the local datastore and replicates it to the other nodes (Figure 3, step 2).

When a node receives a new vote, it verifies the signatures and votes for the current winner and replicates the vote to the other nodes (Figure 3, step 3). The original node waits for other votes to come in, and once a majority agrees, it accepts the token and gives the customer its reward (Figure 3, step 4). The maximum number of Byzantine nodes that can be tolerated is $\frac{1}{3} \times n - 1$, where n is the total number of nodes. The majority of votes required before executing step 4 is $\frac{2}{3} \times n + 1$. Once a node has this many votes for the same value, consensus is finalized and the node can be sure that it will be recognized by all other nodes.

Detect and punish dishonest nodes. Since all data and votes are signed, other nodes can detect when a node acts maliciously. E.g. two conflicting votes for the same token, signed by the same node. Using this cryptographic proof, you can go to the other partners and decide if the dishonest partner can remain in the consortium or should be removed. A node, therefore, has little to gain for not following the protocol, any malicious activity will be detected and can be punished by removing the dishonest member.

3.3 Discussion

For the remainder of this section, we discuss some important properties of the consensus protocol described in the previous part: safety and liveness, the notion of configurable consensus and trade-offs of the middleware.

Safety and liveness. Safety means that nothing bad can happen, i.e. once one node decides that a ticket is redeemed at node X, no other node will ever decide that it is redeemed at node Y. The protocol guarantees safety because a node always waits for enough votes before deciding ($\frac{2}{3} \times n + 1$). Liveness means that eventually something good happens, i.e. the network makes progress, and will eventually decide. Liveness is only guaranteed if the customer itself is honest. With an honest customer, there can only be one valid candidate to redeem the token, since an honest customer does not try to double-spend the token. Eventually, all honest nodes will have received the new proposal and voted for the only candidate to redeem the token. No dishonest node can try to change the outcome by proposing a different candidate since it does not have access to the private key of the customer. When the customer itself is dishonest, multiple candidates can be proposed, since the customer can try to double-spend the token. As long as all nodes wait for $\frac{2}{3} \times n + 1$ votes, they can be sure that the token is not yet spent elsewhere, and safety is ensured. However, the system can end up in a split vote, where there are multiple candidates for the same token whom all received some votes. But none of them have the required majority to decide. In this case, the customer loses its value behind the token, just like in the Avalanche protocol [16]. The last possibility is that nodes are dishonest and start voting for multiple candidates for a single token. Again, this can only happen when the customer itself is dishonest, as the customer is the only one who can nominate values. Therefore, we do not require liveness. When there are at most $\frac{1}{3} \times n - 1$ dishonest nodes, there is still cooperation of $\frac{1}{2} \times h + 1$ of the h honest nodes required, before the required majority of $\frac{2}{3} \times n + 1$ is reached for a single candidate. Since an honest node doesn't vote for multiple candidates, only one candidate can reach enough votes to be accepted.

Configurable consensus. The protocol requires at least $\frac{2}{3} \times n + 1$ nodes to be online, receive the votes, vote themselves and replicate these votes back to the original node before a token can be accepted. For small transactions, where tokens are not worth that much, or when you know and trust the customer, there is no need to wait for that many confirmations. Instead, you can collect the signed request and give the real-life reward immediately to the customer. Later you can use the signed request from the customer to redeem the token. This scenario makes sense for the eLoans case where the identity of the customers is known, and the legal system can help when a customer tries to fraud banks. A third option is that you wait for some confirmations of other nodes to increase the chance that the token is not yet spent. But you do not wait for consensus finality before handing over the real-life reward. This makes sense for the eLoyalty case where you want to accept tokens fast (as customers are waiting to checkout) while not knowing the full identity of the customer. Most of the time, loyalty points are not worth that much, and you don't lose much value when a customer turnes out malicious. In both cases, you can still decide to wait for consensus finality when you estimate that the risk is too high. E.g. it is a new customer who you've never seen, or someone tries to redeem many loyalty points at once.

Trade-offs. The protocol described here is designed to work in an asynchronous environment, i.e. there are no bounds on the time a node might take. The FLP impossibility [7] states that no algorithm can solve consensus in such a setting. As a trade-off, we forfeit liveness for dishonest nodes. The system stays functional, but the tokens from the dishonest customer might be blocked forever and lost. Furthermore, we use tickets instead of coins, so for each token, the nodes only need to decide once who has redeemed it. Honest customers do not try to double-spend tokens, so there is only one possible value to decide on. One thing we sacrifice by not having a ledger is auditability of the past transactions, but since tickets can only be used once, there is little use for it.

4 Preliminary evaluation

The evaluation consists of three parts. First, we revisit the case studies and describe which misuse cases are handled through the middleware, and which are handled outside the system. Second, we implemented the middleware in a web application (plain JavaScript, no plugins required) and evaluate the performance. Third, we compare our lightweight middleware to the infrastructure that is needed to set up a permissioned Hyperledger Fabric blockchain.

Revisiting the case studies. The consensus protocol, that we use before tokens are redeemed, prevents the double-spending attacks. In the eLoyalty case, a store could refuse to pay back another store. One can go to the local government to ask for resolution (since they have digitally signed proofs) or the misbehaving participant can be removed from the network if the other participants agree. For the eLoan case, the consensus also solves the possibility to boycott other

banks since only the customer can decide who can redeem the token. Still, customers can try to use fake invoices or even collude with banks to get fake invoices admitted. Therefore, banks are responsible when they validate invoices and issue tokens. Their funds are on the line when an invoice turns out to be non-existing (handled outside the network).

Performance evaluation. We implemented this protocol in a JavaScript library which can be executed in a web browser. In our experiments, we start 10 VMs in the Azure public cloud (Standard A8v2 with 8 CPUs and 16 GB RAM), running 1-6 Docker containers with a Chromium web browser in each. We also have one server node running, which is used for bootstrapping the P2P connections. All VMs run in the same data center. To simulate a real environment, we used the Linux `tc` tool to increase the latency to 60 ms, which is the latency of a typical 4G network in the US [20].

We're interested in the time it takes to let updates propagate to all nodes. We measure these times both with the BFT protocol, as well as without, to measure the overhead of going from a fully trusted approach to one that tolerates Byzantine behavior. An update is a single token that gets redeemed. It starts with one node trying to redeem a token and ends when all nodes know that the token is redeemed. For the experiments with BFT, this requires that all nodes received at least $\frac{2}{3} \times n + 1$ correct votes, while for the experiments without BFT, one is enough. Another metric is the confirmation time, this is the time it takes for the node that made the update, to know that the network has accepted the update. This confirmation time is a lower bound, and only valid when all nodes follow the protocol. With Byzantine nodes, it needs to wait for more votes, as the node receives some conflicting votes. In this case, the confirmation time will be closer to the synchronization time.

Table 1 shows the results for these experiments for the different number of nodes. Each second, one token gets redeemed. We list both the 50th percentile (the mean), as well as the 99th percentile, which represents most of the users [6]. With only 10 nodes, consensus can be reached in at most 1.2 s. When the number of nodes increases, this increases to 8.3 s for 50 nodes. Going to 60 nodes leads to a 3-fold increase in synchronization times, as can be seen in Figure 4. The reason for this is that the main-thread has trouble receiving and sending the large WebRTC messages. Also processing these messages on the worker thread is becoming slow. The reason for the large messages, is that we need a vote for each node with the correct signatures. The protocol is, therefore, most useful in the range of 10-50 nodes.

Infrastructure requirements. We now compare the lightweight setup of the middleware described in Section 3.1, to the infrastructure required to set up a Hyperledger Fabric blockchain using Hyperledger Composer. First, a web server is required, to host the files for the web-based UI. This web application is not responsible for the consensus, instead, it

Table 1. Synchronization and confirmation times for the different number of nodes with and without the BFT voting protocol, showing both the mean and the 99th percentile. There are no confirmation times for the results without BFT, since you don't need to wait on other nodes.

	nodes	10	20	30	40	50	60
With BFT							
Sync. time [s]	50%	1.0	1.3	2.7	4.0	6.0	14.0
	99%	1.4	2.0	3.9	5.8	9.9	28.6
Conf. time [s]	50%	0.5	1.0	2.3	3.3	4.9	11.9
	99%	1.2	1.9	3.5	4.9	8.3	24.6
Without BFT							
Sync. time [s]	50%	0.6	0.7	0.8	1.0	1.0	1.2
	99%	0.9	1.3	1.8	1.9	1.9	3.1

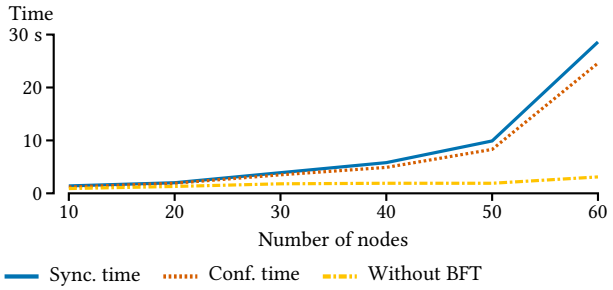

Figure 4. Increasing synchronization and confirmation times for different number of nodes for the 99th percentile.

communicates with the Composer REST server which interacts with the actual blockchain network. Since this REST server contains the private wallet to sign transactions on behalf of the different authenticated clients, each participant needs its own. Next to the web and REST clients, there also needs to be a blockchain network. The network consists of peers and orderers. The peers are required to store the ledger and execute the chain-code, while the orderers establish a total order on the transactions. An orderer is not necessary for each participant. The peers and the REST servers need a CouchDB server for each of them because they need to maintain state. Each participant shall have its own peers, REST server and CouchDB servers. At last, a membership service provider is needed, with one CA server per participant.

If we go back to our first case study, eLoyalty, where small stores want to start integrating their loyalty networks. Setting up the nodes to run chain-code and the API, the REST and CA server is too much work. They lack the knowledge and budget for this kind of deployment. The pluggable architecture of Hyperledger Fabric allows for highly customized applications and consensus with many possible enterprise applications. However, emerging markets and local stores can benefit from a more lightweight approach, such as the middleware described here, which runs inside your browser.

5 Related work

The related work consists of three parts. First, the existing P2P systems for web browsers which run in a fully trusted environment. Second, the PoW blockchains, which require too much computing power to be considered for mobile use. And third, the blockchains that use a BFT consensus protocol.

JavaScript frameworks like Legion [19] and Yjs [14] have proven that the web is suitable for P2P interactions [9]. However, they all assume that members can do any operation they want or even lack any form of authentication (Yjs).

PoW blockchains such as Bitcoin [13] protect against double-spending and tolerate malicious users, but are not suitable for small, mobile devices. It requires high computing power, comparable to that of a small country [15], as well as storage space. The size of the Bitcoin blockchain in August 2019 is 230 GB. Solutions like the simplified payments protocol [13] reduce this by only storing the headers, but the size still grows over time. PoW also has high confirmation times, i.e. it takes a long time before a transaction is accepted. Also, there is no consensus finality, multiple blocks can be mined at the same time, resulting in different forks. Transactions can still be undone by a longer chain in the future.

Another type of blockchains is not based on PoW, but on a BFT protocol. BFT based blockchains have higher performance with less energy and offer proven safety properties [2]. Hyperledger Fabric [1] is a loose architecture of components and allows you to choose your own consensus protocol. HoneyBadger [12] is an asynchronous BFT protocol which doesn't rely on any timing assumptions. It uses asymmetric threshold encryption for censorship resistance and is based on atomic broadcast. Algorand [8] is based on Verifiable Random Functions [11] to let nodes check if they are selected to participate in the agreement on the next set of transactions. Avalanche [16] uses a metastable mechanism to reach consensus. It is a probabilistic approach which converges fast, but without any guarantees. All these approaches maintain a distributed ledger with all the operations.

6 Conclusion and future work

This paper presented a lightweight middleware which can be used to reach decentralized consensus between multiple mistrusting parties and prevent double-spending of resources. We showed, using two industrial case studies, that you don't always need a ledger to solve the double-spending problem, even when there is no centralized, trusted party. The middleware is lightweight and designed to run into the browser, with little bootstrapping needed. You only need one extra server component to bootstrap the P2P connections.

We successfully tested the middleware with 50 concurrent clients and achieved confirmation times which are less than 10 s. However, scaling further, both in the number of clients and the throughput of the system, is an open problem for future work. Three improvements can increase the scalability. First, we can replace the use of elliptic curve signatures to more aggregate signatures, where multiple clients together create a single signature, e.g. collective signatures [18] or threshold signatures [17]. The second improvement can be found in the P2P network. In the current network, peers connect to other peers randomly. By moving to a structured P2P network, such as a fat-tree overlay [10], the replication can flow through the network more efficient. At last, the consensus protocol is now used for each token individually. However, blockchains typically batch multiple transactions into a single block and achieve consensus on those blocks.

References

[1] 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. Cocco Weed, and J. Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains *(EuroSys '18)*.

[2] C. Berger and H. P. Reiser. 2018. Scaling Byzantine Consensus: A Broad Analysis *(SERIAL'18)*.

[3] A. Bessani, J. Sousa, and E. E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART *(DSN 2014)*.

[4] G. Bracha and S. Toueg. 1985. Asynchronous Consensus and Broadcast Protocols. *J. ACM* (1985).

[5] M. Castro and B. Liskov. 1999. Practical Byzantine fault tolerance *(OSDI '99)*.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. 2007. Dynamo: amazon's highly available key-value store *(SOSP '07)*.

[7] C. Dwork, N. Lynch, and L. Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* (1988).

[8] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies *(SOSP'17)*.

[9] K. Jannes, B. Lagaisse, and W. Joosen. 2019. The Web Browser As Distributed Application Server: Towards Decentralized Web Applications in the Edge *(EdgeSys '19)*.

[10] E. Lavoie, L. Hendren, F. Desprez, and M. Correia. 2019. Genet: A Quickly Scalable Fat-Tree Overlay for Personal Volunteer Computing using WebRTC *(SASO '19)*.

[11] S. Micali, M. Rabin, and S. Vadhan. 1999. Verifiable random functions *(FOCS '99)*.

[12] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. 2016. The Honey Badger of BFT Protocols *(CCS '16)*.

[13] S. Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system.

[14] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma. 2015. Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types *(ICWE 2015)*.

[15] K. J. O'Dwyer and D. Malone. 2014. Bitcoin mining and its energy footprint *(ISSC 2014/CIICT 2014)*.

[16] Team Rocket. 2018. *Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies*.

[17] V. Shoup. 2000. Practical threshold signatures *(Eurocrypt 2000)*.

[18] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford. 2016. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning *(SP '16)*.

[19] A. van der Linde, P. Fouto, J. Leitão, N. Preguiça, S. Castiñeira, and A. Bieniusa. 2017. Legion: Enriching Internet Services with Peer-to-Peer Interactions *(WWW '17)*.

[20] 2019. opensignal.com. https://www.opensignal.com/reports/2019/01/usa/mobile-network-experience.