SCEW: Programmable BFT-Consensus with Smart Contracts for Client-Centric P2P Web Applications

Martijn Sauwens, Kristof Jannes, Bert Lagaisse, Wouter Joosen
imec-Distrinet, KU Leuven

Abstract

Collaborative web applications are becoming increasingly client-centric, with technologies such as *WebRTC*, *WebWorkers* and *IndexedDB* enabling a shift towards a decentralized peer-to-peer (P2P) model. Contemporary systems provide fault tolerance and consistency by using Conflict-free Replicated Data Types for synchronization. These systems tolerate crash-faults, but lack resilience against arbitrary faults and malicious users, also known as Byzantine faults. Providing Byzantine fault tolerance (BFT) in web apps is non-trivial. Web apps are executed in web browsers on end user devices. The scarce compute resources and the interactive nature of collaborative web apps do require both a lightweight and low-latency solution, while still providing the Byzantine fault tolerance required by P2P systems.

Our work aims to fill this gap by indroducing SCEW, a programming framework for client-centric P2P web apps that require BFT and interactive collaboration. SCEW achieves this by combining CvRDTs and smart contracts. SCEW represents assets shared by peers as CvRDTs with atomic register semanctics, that provide BFT through the use of BFT-consensus algorithms. SCEW employs smart contracts to define the life-cycle of these shared assets, shielding the application and it's developers from the complexity of the CvRDT's consensus protocol. Experimental results indicate that applications using SCEW can support P2P networks with 100 peers, even when Byzantine faults are present.

1 Introduction

Collaborative web applications are becoming increasingly popular and versatile. Browser technologies such as *WebRTC* [3], *WebWorkers* [7] and *IndexedDB* [1] enable the implementation of responsive and persistent peer-to-peer (P2P) applications in the browser [9]. Existing frameworks for developing P2P web apps such as *Automerge* [11], *Legion* [27] and *Yjs* [21, 22] provide crash-fault tolerance [24] by using *CRDTs* [25] and their P2P architecture. However, none of these systems can tolerate arbitrary or Byzantine faults. The source of these faults is diverse, ranging from software bugs to colluding and malicious users [14, 24]. Dealing with Byzantine faults is more complex compared to fail-stop crashes, requiring elaborate protection mechanisms.

Byzantine fault tolerance (BFT) is often required by P2P applications that collectively manage shared assets with real world value. Examples are integrated loyalty programs [10] and sharing economy [6, 15]. Abuse of shared assets in these applications may lead to real-world damage, either in the form of financial loss or damage to the reputation of participating users. To protect the shared assets against Byzantine faults, P2P applications use BFT-consensus protocols [4]. Consensus protocols ensure that state changes are only committed when a quorum of peers agree on the newly proposed state. Designing and implementing BFT-consensus protocols is sophisticated, requiring thorough testing and formal verification to prove safety. To capitalize on development costs, it is useful to make consensus algorithms programmable. An abstraction for programming BFT-consensus are smart contracts [26]. Smart contracts specify how users interact with shared assets and one another. Deploying the same contract on all peers, in combination with BFT-consensus, ensures that the protected shared state can only evolve according to the specifications of the contract.

Client-centric P2P web apps execute exclusively in web browsers of end-users. This environment is characterized by a lack of compute resources, unreliable communication and high churn rates. Due to the interactive nature of collaborative P2P web apps, a solution to BFT-consensus should not only be lightweight, but also provide low latencies, preferably in the order of seconds [23]. To fulfil these requirements we present the SCEW (Smart Contract Execution for the Web) programming framework. This framework:

1. enables development of client-centric P2P web apps that require BFT and interactive collaboration,
2. uses state-based CRDTs with atomic register semantics for efficient synchronization and BFT,
3. and both defines and manages life-cycles of individual assets with state machine based smart contracts.

Experimental results indicate that SCEW can support interactive collaboration in client-centric P2P web apps, supporting networks with up to 100 users, keeping latencies below 3.2 and 2 seconds for 99% of all transactions in scenarios with and without Byzantine faults respectively.

The remainder of this text is structured as follows: Section 2 elaborates on motivation and use cases. Section 3 provides the reader with additional background. Section 4 presents SCEW, the main contribution of this work. Section 5 discusses the evaluation. We consider related work in Section 6 and conclude in Section 7.

2 Motivation and Use Cases

This section provides two motivating examples of client-centric P2P web apps that require BFT to manage shared

assets. The first application *Loyalty Programs* demonstrates the use of BFT in protecting shared loyalty points, while the second, *Sharing Economy*, aims to mitigate distrust between users participating in a sharing economy application.

Loyalty Programs. Local shops or merchants at a marketplace can implement a shared loyalty program in which their customers can exchange earned loyalty points at any participating shop [10]. To avoid abuse by a single party, the loyalty program is set up as a client-centric P2P web app where loyalty points are managed collectively by all shop owners, rather than a single central party. Decentralizing loyalty point management involves solving BFT-consensus to prevent abuse of these points, such as double spending [18].

Sharing Economy. The sharing economy [6, 15] is based on the observation that consumer items such as tools, cars and other equipment are expensive, while remaining unused for most of the time. Therefore small communities, such as neighborhoods or apartment buildings, can decide to share their equipment to cut down costs. To avoid unwanted fees or privacy issues associated with central parties, sharing economy applications can be provided as a client-centric P2P web app. The web app is responsible for tracking the items and regulating exchange, enabling the users to trace back damage or theft. BFT-consensus eases trust requirements, expecting that users only trust the application and a supermajority of the network, rather than every participating user separately.

3 Background

This section provides the background information and terminology on CvRDTs, blockchains and BFT-consensus used in the remainder of the text. Other related technologies and systems are discussed in Section 6.

Convergent Replicated Data Types (CvRDTs) [25] are a flavor of *Conflict-free Replicated Data Types* (CRDTs) [25] providing Strong Eventual Consistency (SEC) between multiple replicas by defining a join semilattice over a shared state. This lattice has both a partial ordering relation ($\leq$) and a *Least-Upper-Bound* (LUB) operation. Replica managers periodically exchange their local copy of the state with each other to propagate any received updates, merging both the received and local state using the LUB operation. CvRDTs only require a fair-lossy channel for communication, which makes state-based protocols ideal for use in unreliable networks such as the internet. Note that attention should be paid at design time to minimize the size of the CvRDT, as the entire state is sent over the network during synchronization.

Blockchains [31] are an important application for (BFT) consensus. Well known blockchains include Bitcoin [18], Ethereum [30] and Hyperledger Fabric [2]. Blockchains replicate a data-structure called the *ledger*, that is maintained by the blockchain protocol. Users update the ledger by proposing new transactions, aggregating them into blocks. Blockchains such as Ethereum and Fabric use these transactions to initiate calls to smart contracts. When executed, smart contracts can read and write state to the ledger based on their specification, allowing them to update the ledger in a progammable manner. A consensus algorithm decides which block of transactions is added next to the chain. Blocks are chained together by hashes, making it computationally infeasible to change the contents of blocks or their order at a later date. Typical choices for consensus algorithms include *Proof-of-Work* (PoW) and BFT, depending on the context [31]. PoW requires vast amounts of computational resources, as it essentially tries to brute force a solution to a cryptographic puzzle. PoW also lacks consensus finality [29] which causes confirmation times in the order of minutes. Regardless of the used consensus algorithm, peers must also store the entire blockchain to be able to validate the state of the ledger. The high storage overhead combined with potentially high resource usage and confirmation times, makes blockchains a poor fit for client-centric interactive P2P webapps.

Tickets [9] are an alternative approach for managing shared assets in a setting with Byzantine faults. Tickets manage a single asset owned by an individual user. At their creation, Tickets are replicated across all available replica managers to be redeemed at a later date. To redeem a Ticket, the replica manager creates a new proposal that contains both asset data and a signature that approves the transaction. This proposal is then synchronized and validated by the other replica managers, which in turn cast their vote, approving the transaction. The Ticket is considered redeemed only after a supermajority has approved the proposal. Tickets do not exhibit the high storage requirements of a blockchain and are particularly lightweight, making them suitable for browser environments. However, their one-shot nature, together with a lack of support for multiple ownership, restricts Tickets to modelling only simple asset life-cycles.

4 SCEW

This section presents SCEW: a programming framework for lightweight programmable BFT-consensus in client-centric P2P web apps with interactive collaboration. We first present SCEW's core concepts, followed by an overview of its supporting components, such as the atomic register CvRDT, BMachines and primitive contracts, illustrating these concepts with an example. A schematic overview of SCEW's main components is shown in Figure 1.

The SCEW Programming Framework. We first present the SCEW programming framework for client-centric P2P web apps that require both lightweight BFT consensus and interactive collaboration. SCEW draws inspiration from smart contracts and their role in blockchains, combining contracts with the asset management model of Tickets. Applications using the SCEW programming framework must provide the following two components: BMachine smart contracts and integration logic, both of which are shown in Figure 1.

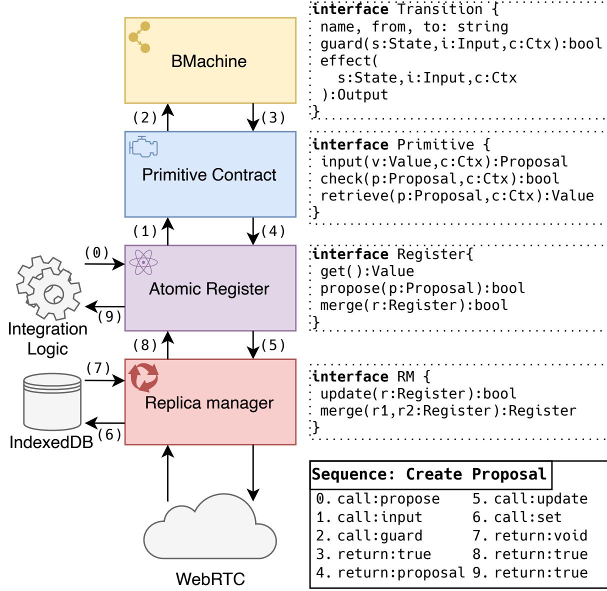

Figure 1. Schematic overview of SCEW. Components are shown as colored rectangles and communicate with each other using the interfaces on the right. The *Create Proposal* sequence at the bottom right shows the control flow for an application that successfully proposes a new value.

Smart contracts model the life-cycle of single shared assets, protecting them from undesired changes. Contracts provide transitions that, when called, transform the state of the protected asset. Note that smart contracts in SCEW manage individual assets and are not allowed to call the contracts of other assets. Resembling Tickets, assets are managed individually rather than collectively. The latter of which is common in blockchains. This behavior is desirable, as assets can be synchronized on an individual level, limiting the size of the state that is exchanged during synchronization.

The integration logic of the application acts as a consumer of the smart contract. SCEW exposes functionality that allows the application to invoke the smart contract and retrieve the shared state. This functionality can then be used to build user interfaces or provide other services.

Atomic Register CvRDTs. CvRDTs with atomic register semantics [10, 12, 13] provide both synchronization and BFT-consensus. Each register stores and protects the state $s \in \mathcal{S}$ of a single shared asset. The state s consists of a value $v \in \mathcal{V}$ and a set of proposals $P \subseteq \mathcal{V}$ for the next value of the register, with $\mathcal{S}$ and $\mathcal{V}$ the sets of register states and values respectively. Atomic registers ensure a consistent view of the shared assets by only permitting a single simultaneous change of the register's value v across the entire P2P network, thus ruling out conflict in the sequence of changes in register's value and allowing the peers to agree on the most recent value of the register. To support these semantics, the register

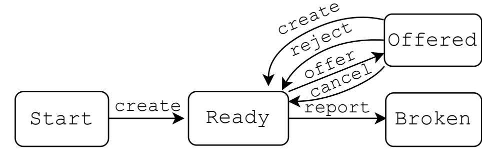

Figure 2. BMachine for sharing tools in a sharing economy use case. Users create tools with the *create* transition and offer them to others by calling *offer*. The offer can then be accepted (*accept*) or rejected (*reject*) by the recipient or can be canceled (*cancel*) by the original owner. Items can be decommissioned by calling *report*, ending the life-cycle.

CvRDT should implement a BFT-consensus protocol. This ensures that changes of the register's value v are atomic, as long as a quorum of peers behaves correctly. SCEW requires the register CvRDT to provide the *Register* interface shown in Figure 1. The *get* method enables the caller to retrieve the most recent known value of the register. Calling *propose* proposes a new value for the register and digitally signs it with the private key of the peer for authenticity. The proposal is then either accepted or rejected by the network that uses the register's BFT-consensus protocol. The consensus protocol is implemented by the *merge* method, which joins the state of two atomic registers. Joining registers enables peers to discover new proposals and reach a consensus on the next value of the register. Once consensus is reached, the peers update the value v of the register to the winning proposal $p \in P$, reflecting the new asset state. Paying attention to size, the atomic register CvRDT can be implemented with the same storage requirements and efficiency as Tickets.

BMachines. SCEW uses smart contracts to define the life-cycle of shared assets stored in atomic register CvRDTs. These contracts are specified as a variant of finite state machines we call *Byzantine Fault Tolerant State Machines* or *BMachines*. BMachines are characterized by the 7-tuple $(Q, q_0, \Sigma_i, \Sigma_s, \mathcal{N}, \delta, \Omega)$ shown in Definition 4.1.

Definition 4.1. BMachine $\mathcal{B} : (Q, q_0, \Sigma_i, \Sigma_s, \mathcal{N}, \delta, \Omega)$

- Q the set of states, with $q_0 \in Q$ the start state
- Σ_i, Σ_s the input and state values respectively
- $\mathcal{N}$ set of transition names
- the transition function $\delta : (Q \times \mathcal{N} \times \Sigma_i \times \Sigma_s) \mapsto (Q \times \Sigma_s)$
- $\Omega : (Q, \Sigma_s)$ the set of instance values for BMachines

BMachines consist of a set of states Q connected via the partial transition function δ. Each asset is stored as a BMachine instance $\mathcal{I}_{\mathcal{B}} \equiv (\mathcal{B}, \omega)$ that combines the definition of the BMachine $\mathcal{B}$ and the instance's value $\omega \in \Omega$. Users of SCEW define a BMachine $\mathcal{B}$ by providing a set of transitions $\mathcal{T}$. Each transition $(n, q_s, q_t, g, e) \in \mathcal{T}$ consists of the transition's name $n \in \mathcal{N}$, a source state $q_s \in Q$, a target state $q_t \in Q$, a *guard* $g : (\Sigma_s \times \Sigma_i) \rightarrow \mathbb{B}$ and an *effect* $e : (\Sigma_s \times \Sigma_i) \mapsto \Sigma_s$

as shown in the *Transition* interface of Figure 1. The transition is applicable if (i) the source state q_s correspond to the current state of the BMachine instance $I_{\mathcal{B}}$ and (ii) the *guard*, acting as a *precondition*, approves the transition based on the current state's associated value $\sigma_s \in \Sigma_s$ and the input $\sigma_i \in \Sigma_i$ provided by the caller. The result $(q_t, \sigma_s') \in (Q \times \Sigma_s)$ of a transition is the combination of both its target state q_t and the result of the *effect*, the latter of which is the *postcondition* of the transition that computes the next state's associated value $\sigma_s' \in \Sigma_s$ based on the same arguments as the guard. Using $\mathcal{T}$, SCEW infers the states Q and composes the partial transition function δ to form a BMachine as in Definition 4.1.

Primitive Contracts. Primitive contracts are the glue between the high-level BMachines and the low-level atomic registers, translating the states $s \in \mathcal{S}$ and methods of the atomic register into states $q \in Q$ and transitions $t \in \mathcal{T}$ of BMachines. The hosting register provides this translation by calling the primitive contract with the methods provided by the *Primitive* interface shown in Figure 1. These methods enable the primitive contract to encode the information necessary to execute the BMachine at all peers, using the set of proposals $P \subseteq V$ and value $v \in \mathcal{V}$ of the atomic register.

Register Modifications. To support primitive contracts, the atomic register has to call the *Primitive* interface when executing its methods. To initiate a state transition, users of SCEW invoke the *propose* method of the hosting register, providing the transition's input $\sigma_i \in \Sigma_i$ as arguments. The register then calls the *input* method of the primitive contract to encode the proposal in a form that allows the other peers to verify the proposal. Consequently, when fetching the value ω of the BMachine instance $I_{\mathcal{B}}$ with the register's *get* method, the register must call *retrieve* to decode the value v stored in the register. Joining register state with *merge* requires the register to check the validity of any new proposals $p \in P$ with the *check* method. This method contains the validation logic of the primitive contract and uses the stored BMachine instance $I_{\mathcal{B}}$ to make a decision.

Executing BMachines. To execute BMachines on atomic registers, the primitive contract must encode the BMachine instance $(\mathcal{B}, (q, \sigma_s))$ and any pending proposals in terms of the register state $s \in \mathcal{S}$, such that other peers can validate any new proposals using the BMachine contract $\mathcal{B}$. One possible encoding for proposals is $(n, \sigma_i, (q', \sigma_s')) \in (\mathcal{N} \times \Sigma_i \times \Omega)$. The field n names the transition invoked by the proposal, while σ_i and (q', σ_s') are for validation. A proposal is deemed valid by the primitive contract if (i) the transition n is defined for the current state q of the BMachine instance $(\mathcal{B}, (q, \sigma_s))$, (ii) the guard with arguments (σ_s, σ_i) yields *true*, (iii) the effect with arguments (σ_s, σ_i) returns σ_s' and (iv) the target state of the transition corresponds to q'. Once the consensus protocol approves the proposal, the latter is assigned as the new value $v' \in \mathcal{V}$ of the register and the *retrieve* method can be used to decode the new value $\omega' \equiv (q', \sigma_s')$ from the BMachine instance. The sequence for proposing a new value by the integration logic is shown in Figure 1.

Example. Developers using the SCEW middleware only have to implement BMachines and integration logic, being oblivious to the BFT machinery provided by the middleware. Listing 1 shows a stylized implementation of the *offer* transition, a transition which initiates the exchange of tools for the sharing economy contract of Figure 2. From a developer's perspective, the guard first checks if the caller is the owner of the tool, after which the effect updates the BMachine instance value $\sigma_s \in \Sigma_s$ with new ownership information. If both calls succeed, the state of the BMachine is updated from *Ready* to *Offered*, completing the transition. The integration logic for calling the transition and retrieve the resulting value is shown in Listing 2.

Listing 1. Implementation of the offer transition, types from the *Transition* interface have been expanded for clarity.

```
1  type Offer={borrower: ID};
2  type ToolOffer={
3    tool: ID; offerer: ID; offeree: ID;
4  };
5  type State={tool: ID;owner: ID};
6  type Ctx={caller: ID};
7  const offerTransition: Transition = {
8    name:'offer',from:'Ready',to:'Offered',
9    guard:(s: State,o: Offer,ctx: Ctx)=>
10     s.owner===ctx.caller,
11   effect:(s:State,o:Offer,ctx:Ctx)=>({
12     tool: s.tool,
13     offerer: s.owner,
14     offerree: o.borrower,
15   } as ToolOffer),
16 };
```

Listing 2. Integration logic for calling the *offer* transition on a register containing the *tool* and retrieve the result.

```
1  await tool.propose({
2      transition: 'offer',
3      args: {borrower: borrowerID },
4  });
5  const value = await tool.get();
```

5 Evaluation

This section presents and discusses the evaluation of SCEW. To evaluate SCEW, we implemented a research prototype of both the middleware and a tool sharing application. The contract used by the application is shown in Figure 2. The atomic registers are implemented using a quorum based BFT-protocol. We first show the experimental setup, followed by a discussion of the results.

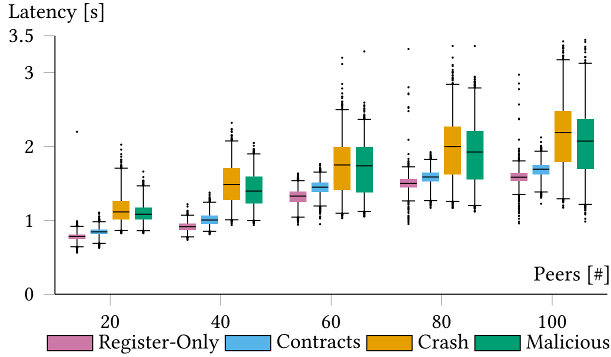

Figure 3. Distribution of latency versus network size for each scenario. Whiskers indicate the 1th and 99th percentile.

Setup. The experiments aim to measure the performance and scalability of SCEW in terms of latency for varying network sizes in different scenarios. The first two scenarios, *register-only* and *contract*, aim to establish a baseline for both the performance overhead caused by the contracts as well as the performance of a network with no faults. The third scenario *crash* investigates the impact of crashes, and the last scenario *malicious* considers malicious peers which are actively injecting faults by violating the contract.

The experiments were conducted on the Azure public cloud. The P2P network is emulated by 4 to 20 *standard F8s v2* virtual machines with 8 vCPUs and 16GB RAM. Each VM runs 5 containerized instances of the tool sharing application in the chromium web browser. The P2P overlay is structured as a flat overlay where each browser is connected with at least 5 other peers. We modelled 4G mobile network conditions with the Linux traffic control tool *tc* [8], increasing the network delay to 60ms [5]. Users are emulated by exchanging tools for 5 minutes at a fixed transaction rate of 1 tx/s, scaling down proportionally as peers leave the network. This transaction rate was chosen as an over approximation for any real-world interactions. Each experiment was executed ten times to increase confidence in our results.

Results and discussion. Results are shown in Figure 3. Comparing the first two scenarios *register-only* and *contracts*, it is clear that adding smart contracts only introduces very limited overhead. Both scenarios show latencies below 2 seconds, even for larger networks, which is sufficient for the interactive performance required by both use cases. The increase in latency for larger networks can mostly be attributed to an increase in overhead by the underlying consensus protocol, as more peers need to vote to reach the quorum.

For both the *crash* and *malicious* scenarios 30% of all peers are affected by faults. The 99th percentile latency increases in both scenarios for larger networks, but stays below 3.2 seconds. This relative increase in latency compared to earlier scenarios can be explained by the way in which the network handles the aforementioned faults. In the case of crashes, peers stop actively partaking in the application, while the remaining peers try to heal the overlay network. The scenario with malicious peers behaves similarly, as peers which violate the contract will be ignored by any honest peers that detect malicious behavior. In both cases the quorum required to reach consensus remains unaltered while the active portion of the network decreases, meaning that a smaller number of peers must collect the same amount of votes to confirm a proposal. This increases the contribution of slower peers to the critical path, increasing latency.

Overall, our evaluation shows that SCEW is suitable for developing in client-centric P2P web apps that require lightweight BFT and interactive collaboration. Keeping latencies below 3.2 seconds in 99% of all transactions, even in the case of failures in networks with 100 peers.

6 Related Work

This section discusses related work not mentioned in Section 3. We first discuss some examples of frameworks for collaborative P2P web apps, followed by C-CRDTs. We wrap up with a discussion of FSolidM, a state machine representation of smart contracts for the Ethereum blockchain.

Automerge [11], Legion [27] and Yjs [21, 22] are frameworks for developing P2P web apps in the browser. All three frameworks make use of CRDT technology for synchronization such as operation-based Commutative Replicated Data Types [25] and Δ-CRDTs [28], making them resilient against crashes. However, none of there frameworks considers Byzantine faults.

Computational CRDTs or C-CRDTs [17, 19, 20] are CRDTs that perform collective computations, using CRDT semantics to merge results from local computations into the final result. Examples include the distributed computation of a sum [20] and top-K leaderboards [19]. This stands in contrast with smart contracts, which all perform the same computation and reach a consensus on the result of a single invocation.

FSolidM [16] is a tool for building and verifying Ethereum smart contracts as finite state machines. It does so by storing the state of the contract as a single object that persists across state transitions. However, this does not work well with a state-based approach where the entire object, including superfluous fields, needs to be sent over the network during synchronization. We instead used a representation that reuses the transition's results as the state of the register.

7 Conclusion

This work presented SCEW, a programming framework for lightweight programmable BFT-consensus in client-centric P2P web apps that require interactive collaboration. SCEW manages shared assets and their life-cycle using a combination of smart contracts based on state machines and CvRDTs with atomic register semantics, implementing a BFT-consensus

protocol. Assets are managed individually rather than collectively, allowing for efficient synchronization at the cost of lacking support for transactions involving multiple assets. However, the absence of cross-asset transactions poses no problems for the use cases considered in this work. The evaluation shows that SCEW is suited to implement systems that scale up to 100 peers, keeping the latency below 2 seconds in scenarios with no failures. Even in the case of Byzantine failures, performance remains acceptable with latencies under 3.2 seconds.

References

[1] A. Alabbas and J. Bell. 2018. *Indexed Database API*. Recommendation. W3C. https://www.w3.org/TR/2018/REC-IndexedDB-2-20180130/

[2] 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. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 30, 15 pages.

[3] A. Bergkvist, D. C. Burnett, C. Jennings, A. Narayanan, B. Aboba, T. Brandstetter, J. Bruaroey, and H. Boström. 2021. *WebRTC 1.0: Real-time Communication Between Browsers*. W3C. https://www.w3.org/TR/2021/REC-webrtc-20210126/

[4] M. Castro and B. Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, USA, 173–186.

[5] K. Fitchard. 2019. *USA Mobile Network Experience Report January 2019*. Opensignal. https://www.opensignal.com/reports/2019/01/usa/mobile-network-experience

[6] K. Frenken and J. Schor. 2017. Putting the sharing economy into perspective. *Environmental Innovation and Societal Transitions* 23 (2017), 3 – 10. Sustainability Perspectives on the Sharing Economy.

[7] I. Hickson. 2015. *Web Workers*. Working Draft. W3C. http://www.w3.org/TR/2015/WD-workers-20150924/

[8] B. Hubert. 2021. *tc(8)*. Linux man-pages project. https://man7.org/linux/man-pages/man8/tc.8.html

[9] K. Jannes, B. Lagaisse, and W. Joosen. 2019. 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* (Dresden, Germany) *(EdgeSys '19)*. ACM, New York, NY, USA, 7–11.

[10] K. Jannes, B. Lagaisse, and W. Joosen. 2019. 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* (Davis, CA, USA) *(SERIAL '19)*. ACM, New York, NY, USA, 3–8.

[11] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan. 2019. Local-First Software: You Own Your Data, in Spite of the Cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. ACM, New York, NY, USA, 154–178.

[12] L. Lamport. 1986. On interprocess communication. *Distributed Computing* 1, 2 (01 Jun 1986), 86–101.

[13] L. Lamport. 1986. On interprocess communication. *Distributed Computing* 1, 2 (01 Jun 1986), 77–85.

[14] L. Lamport, R. Shostak, and M. Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 382–401.

[15] A. Madhusudan, I. Symeonidis, M. Mustafa, R. Zhang, and B. Preneel. 2019. 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, Prague, Czech Republic, 163–171.

[16] A. Mavridou and A. Laszka. 2018. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In *Financial Cryptography and Data Security*, S. Meiklejohn and K. Sako (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 523–540.

[17] C. Meiklejohn and P. Van Roy. 2015. Lasp: A Language for Distributed, Coordination-Free Programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming* (Siena, Italy) *(PPDP '15)*. ACM, New York, NY, USA, 184–195.

[18] S. Nakamoto. 2008. *Bitcoin: A Peer-to-Peer Electronic Cash System*. https://bitcoin.org/bitcoin.pdf

[19] D. Navalho, S. Duarte, and N. Preguiça. 2015. A Study of CRDTs That Do Computations. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data* (Bordeaux, France) *(PaPoC '15)*. ACM, New York, NY, USA, Article 1, 4 pages.

[20] D. Navalho, S. Duarte, N. Preguiça, and M. Shapiro. 2013. Incremental Stream Processing Using Computational Conflict-Free Replicated Data Types. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms* (Prague, Czech Republic) *(CloudDP '13)*. ACM, New York, NY, USA, 31–36.

[21] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma. 2015. Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types. In *Proceedings of the 15th International Conference on Engineering the Web in the Big Data Era - Volume 9114* (Rotterdam, The Netherlands) *(ICWE 2015)*. Springer-Verlag, Berlin, Heidelberg, 675–678.

[22] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma. 2016. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. In *Proceedings of the 19th International Conference on Supporting Group Work* (Sanibel Island, Florida, USA) *(GROUP '16)*. ACM, New York, NY, USA, 39–49.

[23] J. Nielsen. 2010. *Website Response Times*. Nielsen Norman Group. https://www.nngroup.com/articles/website-response-times/

[24] F. B. Schneider. 1984. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Trans. Comput. Syst.* 2, 2 (May 1984), 145–154.

[25] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. 2011. *Conflict-free Replicated Data Types*. Research Report RR-7687. INRIA. 18 pages. https://hal.inria.fr/inria-00609399

[26] N. Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday* 2, 9 (Sept. 1997).

[27] 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. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 283–292.

[28] A. van der Linde, J. Leitão, and N. Preguiça. 2016. Δ-CRDTs: Making δ-CRDTs Delta-Based. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data* (London, United Kingdom) *(PaPoC '16)*. ACM, New York, NY, USA, Article 12, 4 pages.

[29] M. Vukolić. 2016. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In *Open Problems in Network Security*, J. Camenisch and D. Kesdoğan (Eds.). Springer International Publishing, Cham, 112–125.

[30] G. Wood. 2014. *Ethereum: A secure decentralised generalised transaction ledger* (eip-150 revision ed.). http://gavwood.com/paper.pdf

[31] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, and P. Rimba. 2017. A Taxonomy of Blockchain-Based Systems for Architecture Design. In *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, Lyon, France, 243–252.