OrderlessChain: A CRDT-based BFT Coordination-free Blockchain Without Global Order of Transactions

Pezhman Nasirifard
Technical University of Munich
Germany
p.nasirifard@tum.de

Ruben Mayer
University of Bayreuth
Germany
ruben.mayer@uni-bayreuth.de

Hans-Arno Jacobsen
University of Toronto
Canada
jacobsen@eecg.toronto.edu

ABSTRACT

Existing permissioned blockchains often rely on coordination-based consensus protocols to ensure the safe execution of applications in a Byzantine environment. Furthermore, the protocols serialize the transactions by ordering them in a global order. The serializability preserves the correctness of the application's state stored on the blockchain. However, coordination-based protocols limit the throughput and scalability and induce high latency. In contrast, application-level correctness requirements exist that are not dependent on the order of transactions, known as *invariant-confluence (I-confluence).* The I-confluent applications can execute transactions in a coordination-free manner, benefiting from the improved scalability compared to the coordination-based approaches. The safety and liveness of I-confluent applications are studied in non-Byzantine environments, but the correct execution of such applications remains a challenge in Byzantine coordination-free environments. We introduce OrderlessChain, a novel permissioned blockchain based on a novel BFT coordination-free protocol for the safe and live execution of I-confluent applications in a Byzantine environment. We implemented a prototype of our system, and our evaluation results show that our coordination-free approach performs significantly better than coordination-based blockchains.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**.

KEYWORDS

Permissioned Blockchain, CRDT, I-confluence, Byzantine Fault Tolerance, Coordination-free

ACM Reference Format:
Pezhman Nasirifard, Ruben Mayer, and Hans-Arno Jacobsen. 2023. OrderlessChain: A CRDT-based BFT Coordination-free Blockchain Without Global Order of Transactions. In *24th International Middleware Conference (Middleware '23), December 11–15, 2023, Bologna, Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3590140.3629111

1 INTRODUCTION

The main property contributing to blockchains' popularity is the trusted execution of transactions in a trustless, decentralized environment. To offer trust and prevent Byzantine behavior, blockchains use consensus protocols, such as the *Proof-of-Work-based (PoW)* protocol used in Bitcoin [53]. Another essential property of consensus protocols is to enable the system to agree on the total global order of transactions for a serialized execution. The serializability is required to preserve the correctness of the application's state stored on the blockchain. For example, serialization prevents a user's negative account balance in the case of Bitcoin, as every node sequentially executes the transactions in the same order. However, the consensus protocols in several blockchains are severe bottlenecks to their throughput and latency [35, 68].

In contrast to public blockchains, permissioned blockchains are only accessible by authenticated and authorized participants [11, 68]. Although the participants' identity is known, they do not trust each other. Permissioned blockchains, such as *Hyperledger Fabric (Fabric)* [2], take advantage of their permissioned property to implement more efficient coordination-based consensus protocols. However, the coordination-based nature of these protocols remains a bottleneck [14, 15, 71].

Decreasing coordination plays a vital role in improving the scalability of any distributed system [4]. A coordination-free blockchain could enable the concurrent execution of transactions, leading to improved throughput and latency. However, simply eliminating the coordination may jeopardize the application's correctness. For example, a payment processing application may require rejecting transactions that result in negative account balances. A coordination-free blockchain cannot preserve this [4, 38].

In contrast, there exist application-level correctness requirements that *can* be preserved in a coordination-free distributed system, which are known as *Invariant-Confluent (I-confluent)* invariant conditions [4]. For example, transactions that only deposit funds to an account can be executed without coordination. In other words, the I-confluent transactions can be processed in any order while preserving application-level correctness, and the final state of the application is independent of the order of the transactions. One technique that can create I-confluent transactions is *Conflict-free Replicated Data Types (CRDTs)* [70]. CRDTs are abstract data types that converge to the same state in a coordination-free environment.

Bailis et al. [4] demonstrated that unordered transactions preserve the I-confluent invariants of applications in non-Byzantine and eventually consistent environments. In other words, applications with I-confluent invariants are safe and live in non-Byzantine coordination-free environments. The authors also showed coordination-free approaches' improved scalability, throughput, and latency. However,

Middleware '23, December 11–15, 2023, Bologna, Italy

ACM ISBN 979-8-4007-0177-1/23/12…$15.00
https://doi.org/10.1145/3590140.3629111

preserving the safety and liveness of applications in a Byzantine environment depends on paying a high coordination cost in other systems and is challenging without coordination [14, 25, 35, 68, 71, 83]. By providing a BFT coordination-free environment where I-confluent applications remain safe and live, we benefit from improved performance and scalability while ensuring trust in a trustless environment. In this work, we present OrderlessChain, a coordination-free permissioned blockchain without a total global order of transactions. OrderlessChain uses the properties of permissioned blockchains and CRDTs to offer an innovative BFT coordination-free two-phase *execute-commit* protocol for creating safe and live applications. We also built five applications on OrderlessChain to show its practicability.

In summary, we offer the following contributions in this paper:

(1) We introduce a novel BFT coordination-free protocol without requiring the nodes to coordinate to reach a consensus. We also offer proof of its BFT property.
(2) We present OrderlessChain, a novel permissioned blockchain based on our BFT protocol, capable of executing safe and live applications. Our system eliminates the coordination overhead and significantly improves the throughput and scalability over coordination-based blockchains.
(3) We present a novel approach for creating Turing-complete blockchain applications based on CRDTs, which preserves the I-confluent invariants of applications in a coordination-free Byzantine environment and is more scalable than the existing CRDT-enabled blockchain.
(4) We implement a prototype of OrderlessChain and demonstrate our approach's improved throughput and latency for I-confluent applications compared to coordination-based permissioned blockchains.

The remainder of the paper is organized as follows. First, we provide a background on I-confluence and CRDTs in Section 2 followed by our system model in Section 3. Then, we explain our protocol in Section 4. We discuss the applications of OrderlessChain and its implementation in Sections 5 and 6. We also explain our approach for preserving application-level correctness requirements in Section 7, and the effects of Byzantine participants in Section 8. We present evaluations in Section 9 and review related work in Section 10.

2 BACKGROUND

Invariant Conditions and Invariant Confluence – Different applications have different correctness requirements. For example, a banking application may be required to prevent the customers' account balances from dropping below zero. Developers specify the correctness of an application by defining a set of invariant conditions $\{I_1,...,I_s\}$ on the application's state. Each I_j represents a requirement that nodes must preserve during the application's lifecycle. Preserving invariants in a distributed system with globally serialized transactions is relatively straightforward. Provided that each transaction preserves the invariants, serialization enables the nodes to apply the transactions in a sequentially isolated manner and preserve the invariants. However, serialization comes at a high coordination cost. In a coordination-free distributed system, the nodes may receive the transactions in different orders. Hence, preserving invariants is challenging. For example, a node that stores the account balance of a customer with an account balance of $\{Balance: 100\}$ can accept only one of the withdrawal transactions of $Withdraw(50)$ and $Withdraw(60)$. Applying both transactions would result in a negative account balance and violates the application's invariants. Without coordination, the nodes cannot agree to accept one of the two transactions.

Bailis et al. [4] studied preserving invariants in a non-Byzantine coordination-free distributed system and introduced the notion of *Invariant Confluence (I-confluence)*. A set of transactions $\{TS_1,...,TS_m\}$ is I-confluent with regard to an invariant condition I_j, if the transactions can be applied in different orders on different nodes while preserving I_j. Consider the mentioned withdrawal transactions as an example of a non-I-confluent transaction set. However, two deposit transactions $Deposit(50)$ and $Deposit(60)$ are I-confluent, as applying these transactions in any order on different nodes does not violate the non-negative balance invariant condition. Hence, the I-confluent transactions must have these two properties: (1) *Commutativity:* The transactions can be applied in any order. (2) *Convergence:* The final state is independent on the order of transactions. Bailis et al. proved that only I-confluent transactions could be executed on a coordination-free distributed system, and non-I-confluent transactions require coordination among the system's nodes [4].

Conflict-free Replicated Data Types – One available technique that provides commutative and convergent transactions as I-confluence requires is Conflict-free Replicated Data Types (CRDTs). CRDTs represent abstract data types that converge to the same state in the presence of concurrent transactions in a coordination-free distributed system [70]. These data types encapsulate common data structures such as maps and provide APIs for reading and modifying their values. Since concurrent transactions can result in conflicting values, CRDTs use built-in mechanisms to resolve conflicts without coordination. Shapiro et al. [70] formalized CRDTs and proved their strong eventual consistency property (SEC) in an eventually consistent system. An SEC system has two requirements: (1) *Eventual delivery of transactions:* If a transaction is delivered to one correct node, then all correct nodes will eventually receive the transaction. (2) *Strong convergence of nodes:* If the same set of transactions is applied on every correct node, then the nodes' state immediately converges to the same state [70].

CRDTs synchronize among different nodes through propagating commutative transactions [44]. When extending common data structures with CRDT features, the transactions may inherently be commutative or not. For example, a counter is easily modeled as a CRDT since increment transactions are intrinsically commutative. However, modifications for several other data types are not commutative. For instance, assigning a value to a single-value register is not inherently commutative. For converting a register to a CRDT, the register needs to be extended with metadata, defining its behavior in the presence of concurrent modifications. This is achieved with the help of the *happened-before* relation [70] that defines the causal order between two events based on *logical clocks* [41]. The theoretical foundation for defining the requirements of several CRDTs has been studied thoroughly [37, 64].

3 SYSTEM MODEL

System Model – OrderlessChain is a strongly eventually consistent, asynchronous permissioned blockchain. An OrderlessChain network consists of a set of organizations $\{O_1,...,O_n\}$ and a set of clients $\{C_1,...,C_r\}$. Organizations can communicate with other non-failed organizations by sending and receiving messages. A unique identifier is assigned to each organization and client. The identity of each organization is known to every other organization and client in the network. An organization represents entities that range from large corporations to small businesses or even individuals. The purpose of organizations is to define trust boundaries in the system. Although the organizations' identity is known to each other, the organizations do not necessarily trust each other.

Running Example – To better convey our system model and design, we create a voting application to which we refer throughout the paper. Each voter $Voter_i$ can vote for one party among the candidate parties in $\{P_1,...,P_n\}$. The network consists of n organizations, each representing one distinct party. Each organization receives and stores votes from voters. We consider the application correct if each voter votes for at most one party. We chose this use case since voting applications are among popular blockchain use cases [30]. Also, studies have shown that coordination in such highly concurrent use cases is a bottleneck [71]. For example, on Fabric, up to 90% of transactions in a voting application may fail [14].

Application's World State – Each organization stores a replica of the application's state as a set of key-value pairs represented by ST_{O_i}, which represents the application state at organization O_i. Since OrderlessChain is an SEC system, the replicated application states $ST_{O_1},...,ST_{O_n}$ at organizations $O_1,...,O_n$ may diverge from each other, but will eventually converge to the same state. At any given point in time, we define the application's world state ST_{App} as $ST_{App} = \cup_{i=1}^{n} ST_{O_i}$, that is as the union of the application state at all organizations where the values of identical keys are merged based on the techniques discussed in this paper.

Invariant Conditions – An application's correctness is imposed by the developer by defining a set of invariant conditions $\{I_1,...,I_s\}$ on ST_{App}. Each invariant I_j specifies a constraint over ST_{App}. We define the application correctness as follows:

Definition 3.1. **$\mathbf{ST_{App}}$ *Correctness.*** *Let ST_{App} be the application's world state that does not violate the invariant conditions $\{I_1,...,I_s\}$. Let the transaction set $\{TS_1,...,TS_m\}$ be I-confluent with regard to $\{I_1,...,I_s\}$. Then, committing the transactions $\{TS_1,...,TS_m\}$ does not violate any invariant conditions $\{I_1,...,I_s\}$ over ST_{App}.*

Application's Endorsement Policy – The developers specify the *endorsement policy* for the application. The endorsement policy specifies which organizations must sign and commit the transactions. The process of obtaining the signature is called *endorsing*. The application's endorsement policy has the format $EP:\{q\ of\ n\}$, where n is the number of organizations in the system, and q is the minimum number of organizations required for endorsing as well as committing a transaction. In other words, the endorsement policy determines the trust requirements of the application and enables the developer to adjust the amount of trust required.

In the context of our voting example, consider an election with four participating parties P_1,P_2,P_3,P_4 where each party is represented by a corresponding organization $O_{P_1},O_{P_2},O_{P_3},O_{P_4}$. Consider the following two possible endorsement policies: $EP_1:\{2\ of\ 4\}$ and $EP_2:\{4\ of\ 4\}$. EP_1 requires that votes are endorsed and committed by at least two of the four organizations. EP_2 indicates that all four organizations must endorse and commit the voter's vote. Furthermore, we identify one invariant condition: *maximally one vote per voter*. The application is correct if the *maximally one vote per voter* invariant is preserved over ST_{App} and committing transactions do not violate this invariant.

Transaction Model – A transaction is valid as follows:

Definition 3.2. ***Transaction Validity.*** *Let the application's endorsement policy be $EP:\{q\ of\ n\}$. Let ST_{App} be correct concerning the invariant conditions. Let the transaction TS_i be I-confluent concerning the invariant conditions. Then, TS_i is valid if and only if it satisfies these two requirements: (1) Signature validity: TS_i is endorsed by at least q organizations and the client signed the transaction. (2) Invariant conditions validity: Applying TS_i does not violate any invariants.*

We define the transaction TS_i to be committed as follows:

Definition 3.3. ***Committed Transaction.*** *Let the application's endorsement policy be $EP:\{q\ of\ n\}$. Let the transaction TS_i be valid. Then, TS_i is successfully committed if and only if at least q organizations individually process and commit the transaction successfully.*

For the voting example with $EP_1:\{2\ of\ 4\}$, a transaction is valid if it is signed by the client and is endorsed by at least two organizations. Additionally, the valid transaction must not violate the *maximally one vote per voter* invariant. Also, at least two organizations must commit a valid transaction.

Failure Model – We consider the organizations and clients potentially Byzantine. Byzantine organizations or clients can fail arbitrarily. We consider an organization non-faulty if and only if the organization processes every transaction according to the OrderlessChain's protocol. The transactions can be delivered in any order differing from the sent order; they may also be duplicated, lost, or corrupted during transmission. The safety and liveness properties of applications running on OrderlessChain are defined as follows:

Definition 3.4. ***Safety.*** *Only valid transactions are successfully committed.*

Definition 3.5. ***Liveness.*** *Every valid transaction is eventually successfully committed.*

We have two kinds of failures: (1) *Signature failure:* When a transaction does not receive the required endorsements based on the endorsement policy, or the client's signature is invalid. (2) *Organization failure:* Any Byzantine failures of the organizations, including crash and omission failures and the organizations' arbitrary behavior, such as intentionally jeopardizing the system through tampering with messages, forging signatures, or software bugs.

Intuitively speaking, consider the two possible endorsement policies for our voting example. EP_1 requires the endorsement and committing of at least two organizations. Therefore, at most, one of the four organizations can be Byzantine, so the other non-faulty organizations can prevent committing invalid transactions and keep the application safe. With more than one Byzantine organization, the client may collude with the Byzantine organizations and collect the two required endorsements and commits for the invalid transactions, and the non-faulty organizations cannot prevent it. However, the voting application with EP_2 is safe for up to three Byzantine organizations, as the remaining one non-faulty organization can prevent the successful commit of invalid transactions. For liveness with EP_1, the client must communicate with at least two organizations. As there are four organizations, liveness can tolerate two Byzantine failures. However, the liveness of EP_2 cannot tolerate any Byzantine failures, as any faulty organization can hinder the transaction from being endorsed or committed by all four organizations.

Formally speaking, for an application with the endorsement policy $EP: \{q\ of\ n\}$ and with up to f Byzantine organizations, the application is safe if $q \geq f+1$. The application is also live if $n-q \geq f$. We provide proof of the safety and liveness of OrderlessChain in Section 8. The safety and liveness condition of OrderlessChain in a Byzantine environment differs from the conventional $3f+1$ requirement, as we do not require the organizations to coordinate to reach a consensus. Instead, we use the permissioned property of the system and the organizations' known identity to endorse the transactions, where consequently, the non-faulty organizations prevent endorsing and committing invalid transactions.

In the case of a network partition, an application with the endorsement policy of $EP: \{q\ of\ n\}$ can remain available if the number of organizations in every partition satisfies the safety and liveness requirements. Hence, OrderlessChain is available under network partitions according to the CAP theorem [23], if in every partition there exist at least q organizations, and once the network partition is resolved, the state of partitions can be merged based on the techniques discussed in this paper.

4 ARCHITECTURE AND PROTOCOL

OrderlessChain Architecture – Organizations are responsible for hosting smart contracts, receiving and executing transactions, and managing a replica of the application's ledger. Every application running on OrderlessChain makes use of an isolated ledger, which contains the application state ST_{O_i}. The application's ledger on every organization consists of two components: (1) an append-only hash-chain log and (2) a database. The hash-chain log contains all transactions the organization has received since the beginning of time in a hash-chain data structure, ensuring the integrity of transactions. If a Byzantine organization tampers with one transaction, the signature on the log and all succeeding transactions in the hash-chain log will be invalid. By sequentially executing every transaction in the hash-chain log, we reach the application state ST_{O_i}. For a more efficient approach and to avoid executing every transaction each time ST_{O_i} is required, an organization applies each transaction to its database when appended to the log. Therefore, the database represents the current application state ST_{O_i}.

The messages are authenticated using digital signatures based on a standard Public Key Infrastructure (PKI) [49]. Organizations and clients use PKI to authenticate and sign transactions and verify the integrity of the messages.

Developers create smart contracts, which are programs containing the application's logic. The system supports executing Turing-complete logic. Each smart contract can contain several functions that encapsulate the logic of the application's tasks.

Protocol and Transaction Lifecycle – OrderlessChain follows a two-phase *execute-commit* protocol. Clients first submit transaction proposals to be executed by organizations. If the first phase succeeds, clients send the transactions to the organizations to be committed. Figure 1 demonstrates the complete transaction lifecycle for an application with endorsement policy $EP: \{q\ of\ n\}$.

Phase 1 / Execution Phase – The client prepares a transaction proposal TP_i containing the client's identification, the smart contract's identifier, the function to be invoked, and the input parameters. The client broadcasts the proposal to at least q organizations according to the endorsement policy (EP) (Step 1 in Figure 1). Organizations receive the proposal and execute the smart contract with the provided parameters. The execution result is a set of I-confluent operations for modifying the application's state, created based on the CRDT methodology. These I-confluent operations preserve the application's invariant conditions, which we explain in detail in the following sections. The operations are added to a write-set. Then, the organization hashes and signs the write-set with its private key and creates a signature. Finally, the organization delivers the write-set with the created signature as a response (endorsement) to the client (Step 2 in Figure 1). This signature ensures that the client or other organizations cannot tamper with the operations in the endorsement's write-set, as tampering makes the signature invalid.

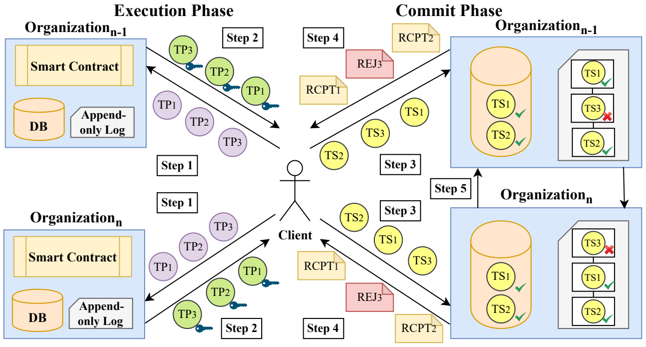


Figure 1: Transaction lifecycle on OrderlessChain.

Phase 2 / Commit Phase – The client waits until it receives the minimum number of endorsements required by the EP. If the write-sets of all endorsements contain identical operations, the client assembles a transaction TS_i. The identical operations in the endorsements show that organizations followed the same protocol for executing the smart contract. Suppose some Byzantine organizations do not execute the smart contract defined by the developer or based on the provided input parameters. In that case, the operations will not match those created by non-Byzantine organizations and will cause the transaction to fail. The client adds the endorsement's write-set to the TS_i's write-set. The client hashes and signs the transaction's write-set with its private key to create a signature to ensure its integrity and includes it in the transaction. The client also includes the received endorsements in the transaction. The client sends back the transactions to at least q organizations as specified by the EP (Step 3). These organizations could be different from those who initially endorsed the proposal. If an organization has yet to commit the transaction, it validates and commits each received transaction according to the definitions above. Before committing a transaction, organizations verify whether the transaction's endorsements and the client's signature are valid (*signature validation*) and whether endorsements satisfy the EP to offer BFT. For verifying the validity of endorsements and the client's signature, the organization hashes the transaction's write-set and uses the public keys of endorsing organizations and the client to verify their signatures. This verification shows that the endorsing organizations created identical write-sets, and the client did not tamper with them. If the transaction passes the signature validation, it is marked as valid and otherwise is invalid.

The organizations update their database with the write-set of valid transactions, whereas all valid and invalid transactions are appended to the hash-chain log. The invalid transactions are added to the ledger for bookkeeping purposes. Since Byzantine clients can create invalid transactions for *Distributed Denial-of-Service (DDoS)* attacks, we discuss countermeasures of such behaviors in Section 8. For appending the transaction to the log, the organization creates a block $Block_h{:}{<}\,TS_i{,}Hash(Block_{h-1})\,{>}$, which contains the transaction and the hash of the last block $Block_{h-1}$ in the log. Then, the organization appends the created block to the log. For valid transactions, a receipt $RCPT_i : HashAndSign(\,Block_h{,}Valid)$, including the signed hash of the block containing the transaction, is sent to the client (Step 4). If the transaction is invalid, the organization sends a rejection $REJ_i : HashAndSign(Block_h{,}Invalid)$ to the client. As the receipt contains the hash of the block, which is dependent on the hash of previous blocks in the log, the organization cannot modify the content of the transaction without destroying and invalidating $RCPT_i$ of TS_i and other transactions. The client awaits receiving the minimum number of receipts the EP requires. The client can archive the transaction's receipts for bookkeeping purposes.

After sending the client's receipt, the organization periodically gossips the transactions to other organizations to ensure every organization receives the client's transactions (Step 5). Upon receiving a transaction from another organization, the organization checks the ledger to determine if the transaction has already been received from other organizations or clients. If the transaction has already been processed, the organization ignores it and avoids committing it again; otherwise, it is committed following the above-explained procedure. If a client sends a transaction that the organization has received from other organizations or a duplicate transaction from the client itself, it does not commit it again. Instead, a receipt or rejection is sent to the client.

5 ORDERLESSCHAIN APPLICATIONS

By discussing two use cases, we explain the possible use cases of OrderlessChain and the system's internal approach for creating CRDT-based I-confluent applications.

Application Modeling – To implement a use case in a smart contract, we need to model the application as data structures that match the use case's description and contain the application's data. We discuss modeling two use cases:

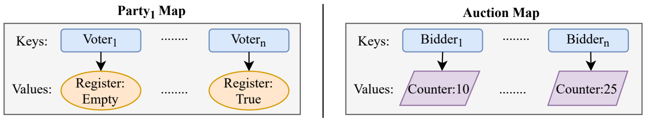
Voting Application – One possible solution for modeling our running voting example in a smart contract is shown in Figure 2(a): For every party participating in the election, we require a map containing key-value pairs. The key is the voter's identification, and the value is a register that stores a Boolean value for the vote sent by the voter for this party.

Auction Application – Auction applications are among the common use cases of blockchains [22]. An auction is a highly concurrent use case that can benefit from a coordination-free approach. Consider an auction and a set of bidders $\{Bidder_1, ..., Bidder_n\}$. The bidder $Bidder_i$ submits bids. Each bid contains the amount it wishes to add to its previous bid. The bidder must be able only to increase its last bid. Based on this description, we realize one invariant condition: *increase-only bids*.

Table 1: Modification and read APIs of supported CRDTs.

CRDT	Modification APIs	Read API
G-Counter	*AddValue*(*value*,*clock*)	*Read*()
CRDT Map	*InsertValue*(*key*,*value*,*clock*)	*Read*(*key*)
MV-Register	*AssignValue*(*value*,*clock*)	*Read*()

One possible design is as shown in Figure 2(b): Each auction is modeled as a map containing key-value pairs. The key is the bidder's identification, and the value is a counter. The counter stores the cumulative bids of the bidders. The counter's value can only be increased, and the value is increased with every new bid sent by the bidder.



(a) Data structure of a participating party. (b) Data structure of an auction.

Figure 2: Application modeling for the voting and auction.

CRDT Abstractions – CRDTs provide a solution for creating commutative convergent operations, and we use CRDTs in smart contracts. OrderlessChain's protocol is independent of CRDTs used in smart contracts. CRDTs are also replaceable with alternative techniques that provide commutative operations, such as *Operational Transformation* [73]. However, many CRDTs exist for various data types whose specifications must be supported by the smart contract execution environment. In the current implementation, OrderlessChain supports the specifications of grow-only counters (G-Counter) [70], CRDT Maps [37], and multi-value registers (MV-Register) [37]. We chose these three CRDTs as they satisfy the requirements of the voting and auction applications. Other use cases may require further CRDTs. For enabling the support for other CRDTs, their design requirements, based on the available literature, must be added to the system [64, 70].

The three CRDTs represent the following data structures: (1) *G-Counter:* It is a monotonically increasing numeric variable. (2) *CRDT Map:* This CRDT is built upon a map structure containing key-value pairs. The key is an identifier, and the value can be any object. (3) *MV-Register:* This is a shared variable capable of containing multiple values simultaneously. Every CRDT provides read APIs and modification APIs for incrementing the G-Counter, inserting a key-value pair to the CRDT Map, and assigning a value to the MV-Register as shown in Table 1. Using the read APIs in the smart contracts causes no side effects and requires no CRDT operation. The developers create operations in the smart contract containing the modification API calls. The value must be `null` for deleting a value. The modification APIs contain a logical clock used to infer the happened-before relations. For creating more complex data structures, maps can be nested, where the value of the key-value pairs can be either a new CRDT Map, G-Counter, or MV-Register.

These CRDTs are used for voting and auction applications as follows. *Voting application:* As previously shown in Figure 2(a), each party is modeled as a map, and the voter's votes are modeled as key-value pairs in the party's map where the values are registers. Therefore, we use a CRDT Map to model the party's map and the MV-Register as the votes' register. *Auction application:* As shown in Figure 2(b), we use a map for modeling the auction and increase-only counters for bids. Hence, we use a CRDT Map to model the auction's map and G-Counters to model the bids.

To evaluate an operation's effects, the operation must be applied to the CRDT, which may cause conflicts. The CRDTs must provide a built-in mechanism for resolving conflicts. We identify the conflicting operations of the three CRDTs and offer a conflict resolution accordingly. (1) *G-Counter:* As the operations increase the counter's value, the modification operations are inherently commutative and cause no conflict. (2) *CRDT Map:* The modification operations that modify different keys in the map are commutative and non-conflicting and can be applied concurrently. However, the operations that modify identical keys are conflicting. The conflict is resolved based on the happened-before relations among operations. If the happened-before relation can be inferred, the operations are applied based on the relation; however, if the happened-before relation cannot be inferred, a new map is created, and the conflicting values are added to the new map as new key-value pairs, as shown in Figure 3. (3) *MV-Register:* On MV-Register, every modification operation is conflicting, and the value of the register is determined based on the happened-before relation among clocks. If the happened-before relation cannot be inferred from the clocks, the register stores all values, as shown in Figure 4.

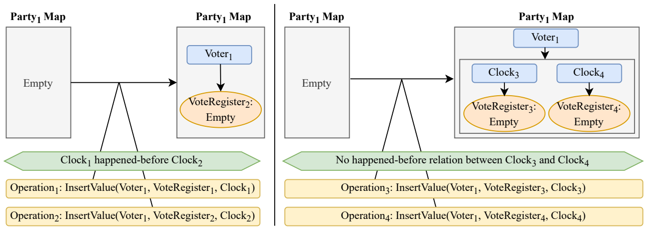

Party$_1$ Map
Empty
Party$_1$ Map
Voter$_1$
VoteRegister$_2$: Empty
Clock$_1$ happened-before Clock$_2$
Operation$_1$: InsertValue(Voter$_1$, VoteRegister$_1$, Clock$_1$)
Operation$_2$: InsertValue(Voter$_1$, VoteRegister$_2$, Clock$_2$)
Party$_1$ Map
Empty
Party$_1$ Map
Voter$_1$
Clock$_3$
Clock$_4$
VoteRegister$_3$: Empty
VoteRegister$_4$: Empty
No happened-before relation between Clock$_3$ and Clock$_4$
Operation$_3$: InsertValue(Voter$_1$, VoteRegister$_3$, Clock$_3$)
Operation$_4$: InsertValue(Voter$_1$, VoteRegister$_4$, Clock$_4$)

Figure 3: Applying CRDT Map modification operations.

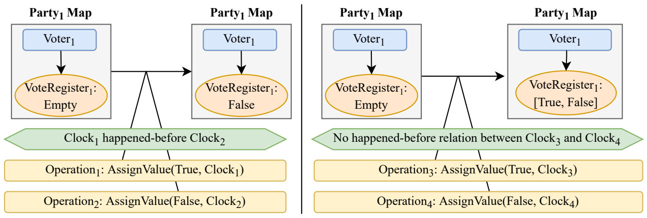

Party$_1$ Map
Voter$_1$
VoteRegister$_1$: Empty
Party$_1$ Map
Voter$_1$
VoteRegister$_1$: False
Clock$_1$ happened-before Clock$_2$
Operation$_1$: AssignValue(True, Clock$_1$)
Operation$_2$: AssignValue(False, Clock$_2$)
Party$_1$ Map
Voter$_1$
VoteRegister$_1$: Empty
Party$_1$ Map
Voter$_1$
VoteRegister$_1$: [True, False]
No happened-before relation between Clock$_3$ and Clock$_4$
Operation$_3$: AssignValue(True, Clock$_3$)
Operation$_4$: AssignValue(False, Clock$_4$)

Figure 4: Applying MV-Register modification operations.

6 IMPLEMENTATION

We implemented a prototype of OrderlessChain with the Go language [3] and gRPC [20]. We open-sourced the code and the smart contracts discussed in this paper [1].

Smart Contracts – Developers use our *Smart Contract Library (SCL)* for developing smart contracts and defining the logic of applications. The smart contract includes functions that encapsulate different functionalities of the application. To enable developers to interact with data stored on the ledger, SCL offers interfaces for defining operations called CRDT APIs. Each client keeps track of a Lamport clock [41], passed into the smart contract with proposals. The client increments the clock with every submitted proposal. Each client's Lamport clock is independent of the clock of other clients. Furthermore, each CRDT object has a unique identification on the ledger. The read API does not require creating any operation, and SCL only requires the identification of the CRDT object to retrieve it. For modifications, in addition to the identification of the CRDT object, each operation includes four components: (1) *Operation identifier:* The identification of the operation is unique per CRDT object and is a combination of the client's identification and the client's Lamport clock. (2) *Modification value and type:* The value that the operation modifies and the type of CRDT. (3) *Client's clock:* The client's Lamport clock. (4) *Operation path:* Developers can create nested CRDT structures for creating more complex data structures. The path specifies the location of the modification, starting from the root of the CRDT object. For example, in the voting application with four parties, the function in the smart contract creates four operations for voting for party P_1. One operation sets the voter's MV-Register on party P_1 to *true*, and the other three operations set the voter's MV-Register on the other three parties to *false*. These four operations are included in the write-set of proposals for vote submission.

Applying Transactions – Developers can implement functions in smart contracts for invoking read APIs and retrieving the values of CRDT objects. Subsequently, clients can submit proposals to an organization O_i for reading the values. In our voting example, the developer can implement a function to read the number of votes submitted to a party. As OrderlessChain is an SEC system, the application state ST_{O_i} may diverge from the application states on other organizations. Therefore, reading the values at O_i only reflects the modifications applied at O_i.

To compute the CRDT object's value in response to read API calls, the organization should retrieve and apply every operation in the ledger submitted for the CRDT object. As the number of operations increases, the time required for applying operations also increases. This increasing overhead is a well-known problem of CRDTs [8, 39]. Hence, we implemented an optimization to address this issue. Section 4 explains that the ledger contains a database besides the hash-chain log. The database is updated with every valid transaction. It consists of a conventional key-value database, namely *LevelDB [24]*, and an in-memory cache. Upon the transaction commit, the operations are inserted into LevelDB. We do so as retrieving the operations from LevelDB is more efficient than retrieving them from the log during a cache miss. The value of the CRDT object in the cache is updated with the transaction's operations according to Algorithm 1. In response to read API calls, the organizations return the value of the CRDT object from the cache. This approach offers *read-your-writes consistency* from the client's point of view [60].

Algorithm 1 demonstrates our approach for applying each operation to the CRDT object. For every operation, before applying it, the CRDT object is traversed from its root until it reaches the location defined by the operation's path (Line 3). As the object can be a nested

[1] https://github.com/orderlesschain/orderlesschain

structure, parts of the path might not have been added to the object yet. Therefore, the missing parts are created and added. Additionally, the location contains the clocks of the previously applied operations. Once the location for modification is reached (Line 4), the changes are applied (Line 5). For applying the changes, as we explained in the CRDT abstractions, the built-in conflict resolution is applied depending on the type of object and the clocks of previously applied operations. Additionally, the operation's clock is appended to the location's clocks. The time and space complexity of Algorithm 1 is $O(n)$, where n is the number of operations being applied.

Algorithm 1: Applying operations to the CRDT.

```
1 ApplyOperations (CRDTObj,Operations)
      input  : CRDTObj, a reference to the CRDT object.
      input  : Operations, the modification operations.
2     foreach $Op_i$ in Operations do
3         $CRDTObj.Create(Op_i.OpPath)$
4         $Location = CRDTObj.GetModifyLoc(Op_i.OpPath)$
5         $CRDTObj.Apply(Location, Op_i.Val, Op_i.ValType, Op_i.Clock)$
```

In Section 8, we prove the SEC property. However, first, we demonstrate that the application state ST_{O_i} is independent of the order of transactions. We formulate the following lemma:

Lemma 6.1. *Independent of the processing order of transactions in the transaction set $\{TS_1,...,TS_m\}$ in organization O_i, application state ST_{O_i} converges to the same state for all i.*

Proof. The write-set of every transaction in $\{TS_1,...,TS_m\}$ only contains CRDT modification operations. As CRDTs are provided with a built-in conflict resolution mechanism, applying the operations in the write-set of operations using Algorithm 1 ensures that transactions can be processed in any order while converging to the same state. Hence, the convergence of ST_{O_i} is independent of the order of transactions. □

7 PRESERVING INVARIANT CONDITIONS

As explained in Section 2, organizations can commit a set of I-confluent transactions in a coordination-free manner without additional validations while preserving the invariants. Since the CRDT operations in the write-set of transactions modify the application's state, the operations must be I-confluent. Developers who define the logic for creating operations in a smart contract must implement the identified invariants as I-confluent operations.

In the case of our voting application, we realized the *maximally one vote per voter* invariant. To determine that the invariant can be preserved by creating I-confluent operations, we reason as follows: Consider an election with two participating parties. As explained in Section 6, every transaction TS_{Vote} that submits a vote has two operations in the write-set. One operation sets the voter's MV-Register in the elected party's map to *true*. The other operation sets the voter's MV-Register for the non-elected parties to *false*.

As there is no coordination among organizations, the voter can submit several votes. However, the *maximally one vote per voter* invariant requires that we only count one of the votes. Consider the following transaction set $\{TS^{Voter1}_{Vote1}, TS^{Voter1}_{Vote2}\}$, submitted by *Voter₁*, as shown in Figure 5. Each transaction contains two operations. *Voter₁* submitted two votes for two different parties, where there exists a happened-before relation between operations in TS^{Voter1}_{Vote1} and TS^{Voter1}_{Vote2}. Therefore, independent of the order they are processed, based on the CRDT's conflict resolution mechanism, operations in TS^{Voter1}_{Vote2} overwrite the effects of operations in TS^{Voter1}_{Vote1}. Hence, we count only one of the votes submitted by the *Voter₁*. The *maximally one vote per voter* invariant is preserved, and the transactions are I-confluent concerning the invariant.

We can similarly reason that the auction application is I-confluent concerning the *increase-only bids* invariant.

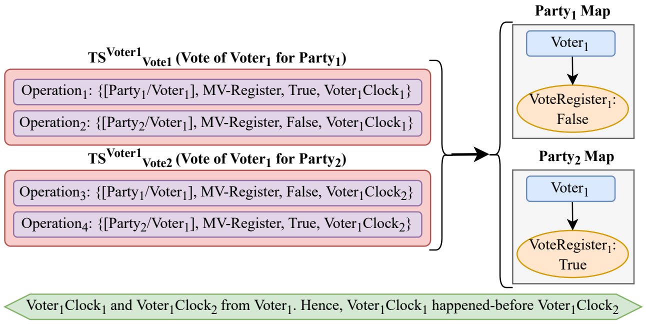

Party₁ Map
TS^Voter1_Vote1 (Vote of Voter₁ for Party₁)
Operation₁: {[Party₁/Voter₁], MV-Register, True, Voter₁Clock₁}
Operation₂: {[Party₂/Voter₁], MV-Register, False, Voter₁Clock₁}
Voter₁
VoteRegister₁: False
TS^Voter1_Vote2 (Vote of Voter₁ for Party₂)
Party₂ Map
Operation₃: {[Party₁/Voter₁], MV-Register, False, Voter₁Clock₂}
Operation₄: {[Party₂/Voter₁], MV-Register, True, Voter₁Clock₂}
Voter₁
VoteRegister₁: True
Voter₁Clock₁ and Voter₁Clock₂ from Voter₁. Hence, Voter₁Clock₁ happened-before Voter₁Clock₂

Figure 5: Preserving the invariant for the voting application.

8 BYZANTINE ACTORS

Organizations or clients are potentially Byzantine. We identify four types of Byzantine faults by clients: (1) A Byzantine client may send proposals to the organizations without sending the transaction to be committed. This does not leave any lasting side effects. However, it can be used for *Distributed Denial-of-Service (DDoS)* attacks. As only authenticated clients can communicate with the organizations, OrderlessChain can employ existing DDoS attack detection mechanisms [16] to revoke Byzantine clients' permissions. (2) A Byzantine client may only send transactions to a subset of organizations during the commit phase. As the organizations gossip the transactions to other organizations after committing the transaction, all organizations eventually receive the transactions. (3) Byzantine clients may send different logical timestamps to different organizations for a proposal. In this case, the operations in the endorsements do not match, which prevents the creation of a valid transaction. (4) If the client does not increment the clock with every proposal, the organizations cannot infer happened-before relations between operations during commit. As explained in Section 5, the proposed CRDT approach can resolve the conflict of such operations without affecting other clients' operations. Therefore, Byzantine clients cannot jeopardize the system.

To discuss the safety and liveness concerning Byzantine organizations, we introduce the following theorem:

Theorem 8.1. *Let the endorsement policy for an application be $EP:\{q\ of\ n\}$ with $n \geq q > 0$. Then, for up to f Byzantine organizations, the application is safe if and only if $q \geq f+1$. Furthermore, the application is live if and only if $n-q \geq f$.*

Proof. According to our definition of safety and liveness, the safe and live OrderlessChain must prevent committing invalid transactions and eventually commit valid transactions. We identify two types of Byzantine faults by organizations. Byzantine organizations may attempt to jeopardize the system by either responding with wrong messages or avoiding responding altogether. Wrong messages include forged signatures from organizations and clients, transactions with tampered or corrupted write-set operations, incorrectly executed smart contracts, or duplicated or lost messages. As the integrity of messages sent by organizations and clients can be examined, the signatures cannot be forged, and the organizations can independently prove the validity of organizations' and clients' signatures. As the system commits every transaction only once, and multiple executions of proposals do not leave any lasting side effects, duplication of messages has no effect. If the messages are suspected to be lost, they can be resent. Additionally, if a client's transaction fails due to the Byzantine organizations' wrong messages, the client can resubmit the proposals to another set of organizations and resend the transaction. On OrderlessChain, the developers identify and define the application logic for creating I-confluent update operations. Therefore, the invariants are preserved as long as the write-set operations are not tampered with and the smart contract is executed as defined by the developer. Since the write-set of every endorsement must include identical operations, as long as there exists at least one non-faulty organization among the q endorsing organizations, which creates the write-set operations that can be differentiated from the tampered operations or the incorrectly executed smart contract, creating a valid transaction is impossible, and the application is safe. Hence, the application is safe if and only if $q \geq f+1$.

Byzantine organizations may not respond to clients. For the application to be live, the client must endorse and commit the transaction on q among n organizations. Therefore, the transaction can reach at least q organizations if and only if $n-q \geq f$. Therefore, the application is live if and only if $n-q \geq f$. □

We demonstrated that liveness and safety depend on the application's endorsement policy. In other words, the safety and liveness can be tailored to the application's requirements. For example, for the voting application with four parties, the regulation of a fair election may dictate that all parties endorse every vote. Therefore, we need $EP:\{4\ of\ 4\}$. If the regulations demand the endorsement of at most two parties, we can have an $EP:\{2\ of\ 4\}$. Furthermore, since the Byzantine behavior of organizations can be observed, and the identity of organizations is known to each other, the organizations have the incentive to behave honestly, as otherwise, they may face the consequences. For example, a Byzantine party jeopardizing the election may face legal consequences.

The following theorem demonstrates that ST_{App} is SEC.

Theorem 8.2. *Let the application be safe and live. Then, the application's world state ST_{App} is SEC.*

Proof. According to the definition of SEC in Section 2, an SEC system must satisfy two requirements of *eventual delivery of transactions* and *strong convergence of nodes*. In Theorem 8.1, we demonstrated that every valid transaction is committed for a safe and live application. Additionally, non-faulty organizations gossip the transaction to other non-faulty organizations. Therefore, provided that the application is safe and live, every non-faulty organization eventually receives a valid transaction. Hence, *eventual delivery of transactions* is satisfied.

In Lemma 6.1, we proved that independent of the order of transactions in the transaction set $\{TS_1,...,TS_m\}$, the application state ST_{O_i} at organization O_i converges to the same state for all i. Since the *eventual delivery of transactions* requirement for the safe and live application is satisfied, when the transaction set $\{TS_1,...,TS_m\}$ is delivered to the non-faulty organization O_i, the same set is delivered to every other non-faulty organization. Therefore, according to Lemma 6.1, all ST_{O_i} converges to the same state, and the requirement *strong convergence of nodes* is satisfied. Hence, the application's world state ST_{App} of a safe and live application on OrderlessChain is SEC. □

9 EVALUATION

We first evaluate OrderlessChain. Then, we compare it to *Fabric* [2], *FabricCRDT* [54], *BIDL* [66] and *Sync HotStuff* [1]. Fabric is a permissioned blockchain capable of executing Turing-complete applications. FabricCRDT (built as an extension on top of Fabric) runs CRDT-enabled applications. BIDL is a permissioned blockchain optimized for data center networks inspired by Fabric. Sync HotStuff introduces a synchronous BFT consensus protocol based on the HotStuff protocol [80]. Fabric, FabricCRDT, and BIDL's network comprise organizations. We adjusted Sync HotStuff to employ the concept of organizations. On Fabric and FabricCRDT, the clients send the transactions to an ordering service for consensus and to create a global order by batching transactions into blocks. Before the transaction commits, Fabric's organizations perform a *multi-version concurrency control validation (MVCC validation)* to ensure that the application's invariants are preserved. FabricCRDT does not perform an MVCC validation and only merges the transaction values using JSON CRDT techniques [37]. BIDL uses a central *sequencer* for sequencing transactions. Afterward, it executes the transactions and performs coordination-based consensus in parallel. Sync HotStuff uses coordination- and leader-based consensus for ordering and executing transactions.

We compare OrderlessChain to Fabric, FabricCRDT, BIDL, and Sync HotStuff prototypes. We implemented the prototypes by studying the available source code and following their concepts using Go, gRPC, and LevelDB. We implemented these prototypes because the original Fabric, FabricCRDT, and BIDL offer many security and network-related features we do not implement in OrderlessChain. These features impose performance penalties and would have caused an increased transaction latency. For example, in the case of Fabric, Gorenflo et al. [25] and Chacko et al. [14] offer extensive insights on the performance penalties. We replicated the original implementations for a fair comparison since we intended to compare our coordination-free protocol to their coordination-based protocols independently of the implementation of the rest of the system. Furthermore, the CRDT approach in FabricCRDT does not use the cache we implemented as an optimization. For fairness, we also implemented such a cache in FabricCRDT's CRDT approach.

Experimental Applications – We developed a synthetic application for evaluating OrderlessChain. Based on the examples discussed, we also implemented voting and auction applications for comparing OrderlessChain to the other four systems. Every application consists of one smart contract, and in total, we developed eleven smart contracts (available in the Git repository mentioned in

Section 6). Each smart contract has one *modify-function* for modifying the data on the ledger and one *read-function* for retrieving data from the ledger.

Synthetic Application – For a controlled evaluation of OrderlessChain, we implemented a synthetic application. The application's smart contract includes two functions $\mathit{Modify}(\mathit{ClientId}_i, \mathit{Clock}_i, \mathit{ObjCount}, \mathit{OpsPerObjCount}, \mathit{CRDTType})$ and $\mathit{Read}(\mathit{ObjCount})$. The *Modify* function receives the client identification and clock, the number of CRDT objects and operations per each CRDT object modification, and the CRDT type. The write-set of the transaction includes $\mathit{ObjCount} \times \mathit{OpsPerObjCount}$ operations. The *Read* function reads a specific number of CRDT objects as specified by *ObjCount*.

Voting Application – We developed voting applications for all five evaluated systems. The application's smart contract for OrderlessChain has two functions: $\mathit{Vote}(\mathit{Voter}_i, \mathit{Clock}_i, \mathit{Party}_j, \mathit{Election}_l)$ and $\mathit{ReadVoteCount}(\mathit{Party}_j, \mathit{Election}_l)$. For an election with n parties, the *Vote* function results in n total operations (one operation per object) in the write-set as explained in Section 6. *ReadVoteCount* retrieves the current number of votes of Party_j. The smart contracts of the other four systems also include *Vote* and *ReadVoteCount* functions, which are implemented based on the best practices for developing smart contracts on these systems [14, 54, 66].

Auction Application – The auction application's smart contract of OrderlessChain has two functions: $\mathit{Bid}(\mathit{Bidder}_i, \mathit{Clock}_i, \mathit{BidIncrease}_i, \mathit{Auction}_j)$ and $\mathit{GetHighestBid}(\mathit{Auction}_j)$. The *Bid* function includes one operation in its write-set for increasing the bidder's G-Counter. *GetHighestBid* reads the current highest bid. The smart contracts of the other four systems also include a *Bid* and a *GetHighestBid* function.

Workloads, Control Variables and Metrics – Each experiment is executed on an initially empty ledger. We submit a workload containing transactions invoking the modify- and read-functions in the smart contracts, also referred to as *modify-* and *read-transactions*. The workload includes a specific percentage of modify-transactions and read-transactions, uniformly distributed during the execution of the experiment. Each organization receives a specific percentage of the load on the system. We define the transaction arrival rate in *transactions per second (tps)* of the system as the total number of transactions per second submitted by all clients to the system. The other control variables are the number of organizations, endorsement policies, the Byzantine failures, and the number of organizations to which each organization gossips the transaction, which we refer to as the *Gossip Ratio*. The gossips are propagated at one-second intervals. For the endorsement policies of $\mathit{EP}: \{q \mathit{ of } n\}$, the clients send the proposals and transactions to exactly q organizations. Each organization has one node on Fabric, FabricCRDT, BIDL, and Sync HotStuff. Each experiment is executed for 180 seconds. Fabric, FabricCRDT, and BIDL use the *Solo* ordering service [2].

For the synthetic application, we used 1000 clients. *ObjCount*, *OpsPerObjCount*, and *CRDTType* are control variables. We defined 1000 voters, eight elections, and eight parties per election for the voting application. We defined 1000 bidders, eight auctions, and a gradually growing number of bids for the auction application. We chose these values according to the scalability evaluation of Fabric done by other authors [14]. The input parameters for modify- and read-transactions are randomly selected from these predefined values based on a uniform distribution during the experiment.

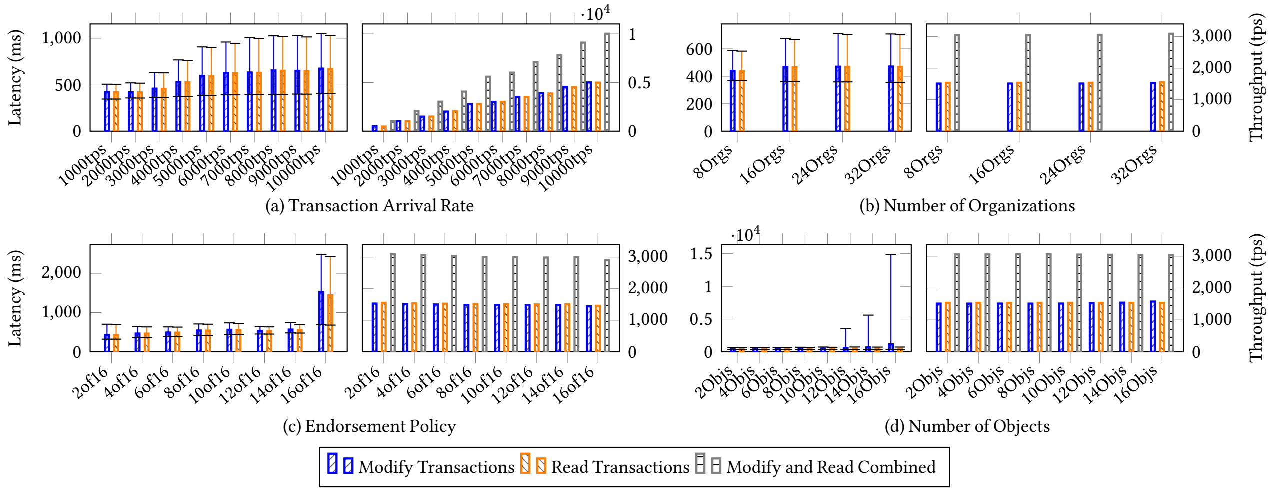
Each experiment is executed at least three times, and the results are averaged. At the end of each experiment, the performance metrics are collected. We measure the *transaction throughput*, the *average transaction latency*, the *1st percentile transaction latency*, and the *99th percentile transaction latency*. The transaction throughput is the total number of successfully committed transactions divided by the total time taken to commit these transactions. The transaction latency is the response time per transaction from sending the proposal until receiving the commit receipts from organizations, according to the endorsement policy.

Experimental Setup – Each organization of the five studied systems runs on an individual KVM-based Ubuntu 20.04 virtual machine (VM), and different organizations do not share VM resources. Each VM uses 9.8 GB of RAM and four vCPUs. Since the VMs are located within a single cluster and are connected via LAN, we used Ubuntu's *NetEm (network emulation)* and *tc (traffic control)* facilities for adding 100 ms ping delay, 4 ms jitter, and 100 Mb rate control to all links for emulating a WAN. We chose these values by observing the delays and bandwidth between two Ubuntu servers in two cities in Europe and North America, provided by two cloud providers. Based on studies that also use emulated WANs [14] and our observations, this setting fairly accurately emulates a realistic WAN. The ordering service of Fabric, FabricCRDT, and BIDL, the sequencer of BIDL, and the leader of Sync HotStuff runs on separate VMs. We also developed a benchmarking tool that orchestrates a distributed deployment of clients, generates and submits transactions, and collects performance metrics. The benchmarking tool is inspired by *Hyperledger Caliper* [65], and its code is published with the system's code.

Table 2: Control variables of synthetic application.

Control Variable	Default	Executed Configuration
(1) TS Arrival Rate	3000 tps	{1000 tps, ..., 10,000 tps}
(2) Number of Orgs	16 Orgs	{8 Orgs, ..., 32 Orgs}
(3) Endorsement Policy	$\{\mathit{4\ of\ 16}\}$	$\{\{\mathit{2\ of\ 16}\}, ..., \{\mathit{16\ of\ 16}\}\}$
(4) Number of Obj	1 Obj	{2 Objs, ..., 16 Objs}
(5) Operations per Obj	1 Op	{2 Ops, ..., 16 Ops}
(6) CRDT Type	G-Counter	{G-Counter, MV-Register, Map}
(7) Workload (Read/Mdfy)	R50M50	{R10M90, ..., R90M10}
(8) Workload per Org	Uniform	{Uniform, Normal Distribution}
(9) Gossip Ratio	1 Org	{1 Org, ..., 15 Orgs}
(10) Byzantine Orgs	0 Failure	{1 , 2 , 3 } Failures
(11) Byzantine Clients	0% Failure	{50%, 75%, 100% } Failures
(12) Byzantine Orgs/Clients	0/0% Failure	{3/50%, 3/75%, 3/100% } Failures

Experimental Results for Synthetic Application on OrderlessChain – Table 2 displays the control variables, their default values, and the executed experimental configurations for the synthetic application on OrderlessChain. One of the control variables is set to the executed configurations, and the other control variables are set to the default value. As shown in Figure 6(a), the throughput increases with an increasing transaction arrival rate, but the latency rises. We studied the effect of increasing the number of organizations on throughput and latency, as shown in Figure 6(b). We set the endorsement policy for each experiment to $\mathit{EP}: \{\mathit{4\ of\ NumberOfOrgs}\}$. We observe that the system scales for increasing organizations without affecting the throughput and latency. As shown in Figure 7, we also compared the average latency to throughput for an increasing number of organizations and arrival rates and observed that OrderlessChain scales. With an increasing number of organizations required



(a) Transaction Arrival Rate

(b) Number of Organizations

(c) Endorsement Policy

(d) Number of Objects

Figure 6: Throughput, average, 1st, and 99th percentiles transaction latencies for executed configurations of synthetic application.

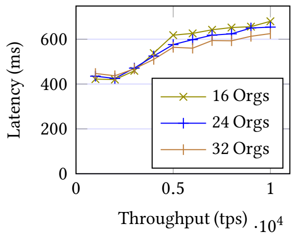
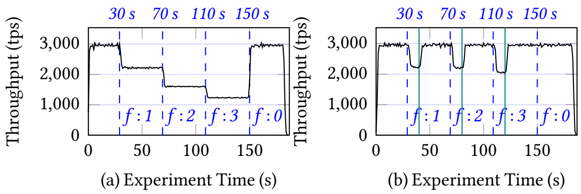


Figure 7: Average latency to throughput.

by the endorsement policy, we observe that the latency increases as the load on the organization increases, as shown in Figure 6(c). We observe in Figure 6(d) that the latency increases for a larger number of objects in the transaction due to the locking mechanism used in the cache to avoid concurrent reads and writes. The results of experiments with configurations 5 to 9 are explained in the following and are not shown in the figures due to space limitations. We observe that throughput and latency are unaffected by the increasing number of operations and are independent of CRDT types. We gradually decreased the modify-transactions in the workload from 90 percent to 10 percent, and we observed that the latency and throughput were unaffected. We also changed the distributed workload per organization from a uniform to a normal distribution, where some organizations received a higher percentage of the workload. We did not observe a significant difference except for the slight increase in latency for the higher-loaded organizations. We did not observe a significant change in latency and throughput for an increasing gossip ratio either.



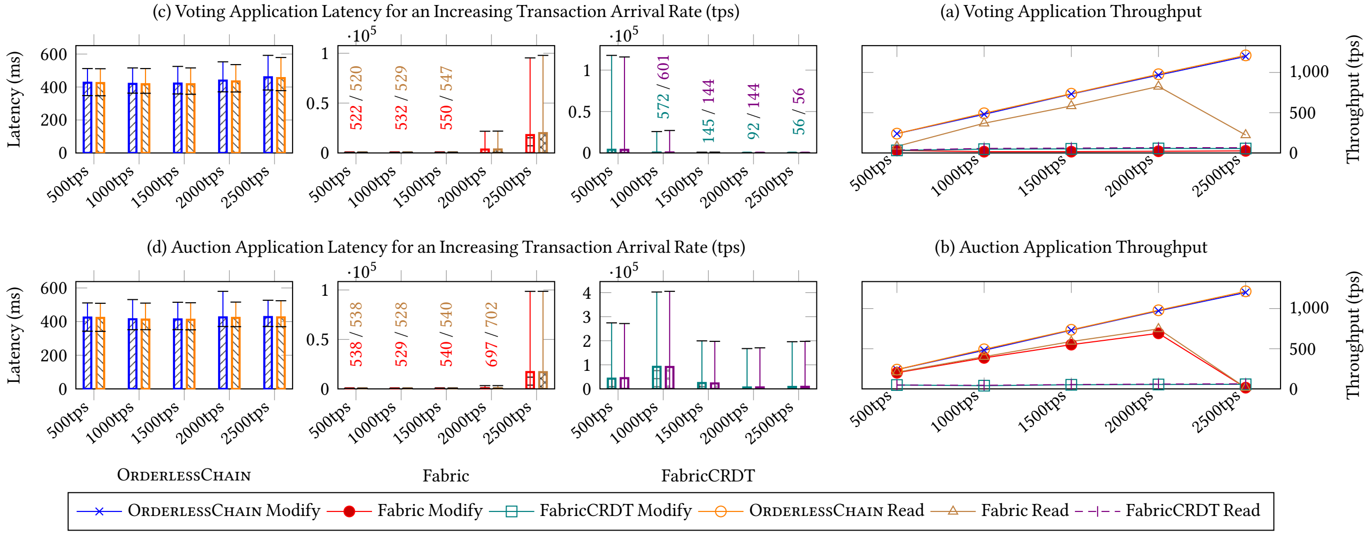
(a) Experiment Time (s)

(b) Experiment Time (s)

Figure 8: Experiments with Byzantine organizations.

We studied the effects of Byzantine failures. First, as shown in Figure 8(a), three randomly selected organizations behave arbitrarily for a specific period while all clients are non-faulty. The Byzantine organizations either randomly avoid responding to clients or endorse the proposals incorrectly. The Byzantine organizations also randomly avoid forwarding the transactions to other organizations. We included three Byzantine organizations as, based on the $EP:\{4\ of\ 16\}$, the safety and liveness of the application can tolerate up to three Byzantine failures, and this is the worst-case scenario for organizations. We observed that the throughput decreases with every Byzantine failure. However, the latency is not affected (not shown in the figures). The decreasing throughput is due to clients being unable to collect the minimally required valid endorsements. Since clients can observe organizations that wrongly endorse or do not respond while others respond with lower latency, they can avoid Byzantine organizations. To demonstrate this, we ran experiments where clients randomly selected another organization. As shown in Figure 8(b), the throughput returns to its pre-failure value immediately after clients avoid the Byzantine organizations, as shown by the solid green lines. We also ran experiments where all organizations were non-faulty with an increasing percentage of Byzantine clients, randomly either not sending the transactions for commit after the execution phase or tampering with the transaction's write-set. We observed that all faulty transactions are rejected while the latency is unaffected, showing the system stays safe and live (results are not plotted). Finally, we executed experiments with three Byzantine organizations with an increasing percentage of Byzantine clients. Similar to previous Byzantine experiments, we observed the decreased throughput without affecting latency, and the system remained safe and live with no extra cost due to Byzantine failures.

Vote and Auction Applications – We compared OrderlessChain with Fabric and FabricCRDT with 8 organizations for each system and the $EP:\{4\ of\ 8\}$. Then, we compared it with BIDL and Sync HotStuff with 16 organizations for each of the three systems and the $EP:\{4\ of\ 16\}$. We did so as the configuration with 16 organizations for Fabric and FabricCRDT caused the failure of a significant portion of transactions due to their coordination-based approach limitations, which prevented us from providing meaningful insights. Also, for FabricCRDT and Sync HotStuff, we observed that latency


(c) Voting Application Latency for an Increasing Transaction Arrival Rate (tps)

(a) Voting Application Throughput

(d) Auction Application Latency for an Increasing Transaction Arrival Rate (tps)

(b) Auction Application Throughput

Figure 9: Experiments with voting and auction applications on OrderlessChain, Fabric, and FabricCRDT.

significantly increases for a higher transaction arrival rate due to FabricCRDT's CRDT implementation and Sync HotStuff's leader-based approach, so we limited the transaction latency for them to 240 seconds, after which they are timed out and not considered for throughput and latency evaluation.

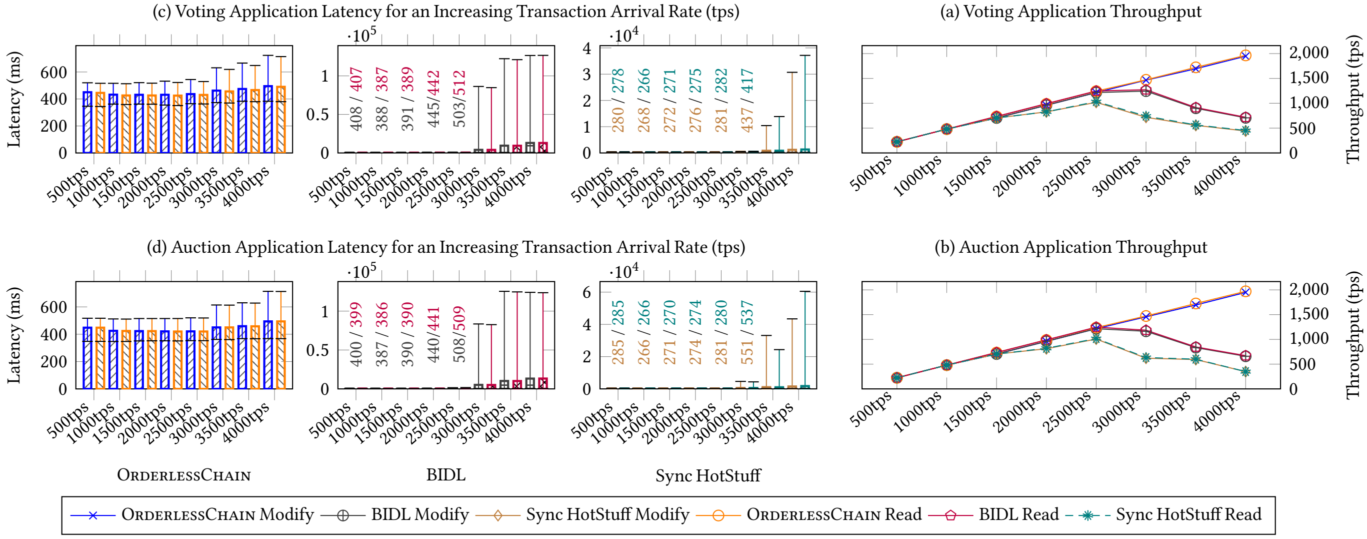
As shown in Figures 9(a) and (b), we observe that OrderlessChain demonstrates a higher throughput for both applications. On Fabric, the failed transactions due to the MVCC validation, explain its low throughput. Although we used caching for the CRDT approach in FabricCRDT, the CRDT approach still is a bottleneck. As shown in Figures 9(c) and (d) (for the lower values, the average latencies are written on the plots), OrderlessChain's latency remains constant under increasing arrival rates. Fabric's latency significantly increases for higher arrival rates. The reason is that Fabric's central ordering service for consensus is a bottleneck, as shown in Table 3 for the 2500 tps of the voting application (the results do not include the added WAN network latency on the client side). The increased latency causes more transactions to fail due to MVCC validation. FabricCRDT demonstrates irregular latency patterns as timed-out transactions are not considered. As demonstrated in Figures 10(a) and (b), we observe that although both BIDL and Sync HotStuff scale better than Fabric and FabricCRDT, OrderlessChain demonstrates a higher throughput for both applications. Furthermore, OrderlessChain's latency stays constant as the latency of BIDL and Sync HotStuff significantly increases for higher arrival rates. As the design of BIDL is highly optimized for data center networks with high bandwidth and low network latency, their proposed coordination-based approach for consensus and BIDL's central sequencer, becomes a bottleneck in a WAN setup with limited bandwidth and higher network latency, as shown in Table 3 for the 4000 tps of the voting application. These results corroborate the findings in the BIDL paper. For Sync HotStuff, the main bottleneck is the leader component in their coordination-based approach.

We observe that for identical configurations, the organizations of all five systems utilize the same amount of memory on average. For example, each organization of OrderlessChain and Fabric consumes on average 400 Mb of Heap for the 2500 tps of the voting application. However, the CPU utilization of OrderlessChain is higher than the utilization of other systems. For example, the Fabric organization's CPU utilization for the 2500 tps of the voting application is, on average, at 30%, whereas the OrderlessChain organization is at 50%. This higher utilization is attributed to applying the CRDT operations to the cache. However, as applying the modifications to the cache is done sequentially due to the employed locking mechanism, the higher CPU utilization for cache operations is bounded. The main limitation of OrderlessChain is the cache's locking mechanism to avoid concurrent reads and writes due to Go language constraints, which can be addressed using other technologies that offer lock-free data structures.

Table 3: Breakdown average transaction processing time.

OrderlessChain (ms)	Fabric (ms)	BIDL (ms)	Sync HotStuff (ms)
P1/Execution: 64	P1/Endorse: 59	P1/Sequence: **346**	P1/Consensus: **5532**
P2/Commit: 110	P2/Consensus: **17270**	P2/Consensus: **6803**	P2/Commit: 6
	P3/Commit: 11	P3/Execution: 54	
		P4/Commit: 1	

Discussion – We do not require coordination for preserving I-confluent invariants. However, coordination is required for applications with non-I-confluent invariants. Suppose we require an invariant to specify a deadline for the end of an election, after which the votes are rejected. This is a non-I-confluent invariant and requires coordination. One approach for enabling OrderlessChain to preserve such invariants is extending it with coordination-based protocols of Fabric and enabling this protocol when required. For example, given that the end of an election specifies only a short time of the whole time this event runs, which can be up to a few hours or days, the coordination-based protocol can be enabled only when we


(c) Voting Application Latency for an Increasing Transaction Arrival Rate (tps)

(a) Voting Application Throughput

(d) Auction Application Latency for an Increasing Transaction Arrival Rate (tps)

(b) Auction Application Throughput

Figure 10: Experiments with voting and auction applications on OrderlessChain, BIDL, and Sync HotStuff.

are near the end. Otherwise, we use our scalable coordination-free protocol.

There exists an extensive range of I-confluent CRDT-based use cases [10, 13, 17, 32, 33, 39, 47, 50, 52, 58, 69, 74–77, 81, 82], from key-value stores to collaborative environments, which can be implemented on OrderlessChain. Also, CRDT-based and I-confluent development tools such as *Automerge* [36], *Katara* [40] and *Lucy* [78] for modeling and expressing various applications can be adapted to OrderlessChain to offer BFT. Katara offers a solution for automatically creating CRDTs from sequential non-CRDT implementations. Lucy provides an environment for determining whether invariant conditions are I-confluent. We developed other applications [55–57] (not evaluated here) as proof of concept. We implemented an IoT-based supply chain use case to monitor the health of temperature-sensitive products during transit. We also implemented a trusted distributed file storage system and a private *Federated Learning* system by extending OrderlessChain with customized CRDTs. The development of these applications on OrderlessChain was straightforward.

10 RELATED WORK

The low scalability of PoW-based protocols makes them infeasible for permissioned blockchains such as *MultiChain* [26], *R3 Corda* [28], *Quorum* [51], and Fabric [2], which use various non-PoW-based *coordination-based* protocols. Although these protocols improve performance, the required coordination among nodes negatively affects performance. Also, many transactions fail due to Fabric's optimistic coordination-based protocol [14, 71]. Also, the *Raft*-based [59] ordering service of Fabric is not BFT. Other studies propose BFT orderers [7, 9].

Reducing coordination to improve scalability while preserving invariants has been an active field of research. Several studies propose solutions in non-Byzantine systems [5, 6, 42–46, 61, 62]. However, they do not consider the added complexity of Byzantine failures for preserving invariants. Some works reduce coordination while offering BFT and preserving invariants [27, 38, 48, 63, 67]. However, they do not eliminate the coordination or have limited use cases.

Bailis et al. [4] introduced I-confluence, which shares similarities with *Left Commuting Operations* [21] for identifying the possible order of operations to persevere the application's serializability. It also shares similarities with the *CALM theorem* [29], demonstrating that monotonic transactions can be processed coordination-free.

Studies propose coordination-based BFT approaches for executing CRDT applications [18, 19, 72, 83]. However, only some works study CRDTs in blockchains. *Vegvisir* [34] study integrating CRDTs with a *Directed Acyclic Graph-structured* blockchain without support for executing smart contracts. *RAMBLE* [31] proposes a blockchain-based Twitter-like messaging protocol based on CRDT sets. *MEChain* [79] proposes a CRDT-enabled blockchain-based system for storing electronic health records. *Setchain* [12] decreases coordination in blockchains by only partially ordering transactions. However, their solution is only limited to grow-only sets and still requires some round of coordination. FabricCRDT [54] uses coordination-based JSON CRDT techniques. The difference between FabricCRDT and OrderlessChain, besides our system enabling BFT CRDTs in a coordinate-free environment, is FabricCRDT's state-based CRDT approach. For every modification on FabricCRDT, the entire object stored on the ledger must be retrieved and modified and then sent to organizations to be merged with the existing objects. On FabricCRDT, the objects gradually become large, negatively affecting the performance, as observed here.

11 CONCLUSIONS

We presented OrderlessChain, a BFT coordination-free permissioned blockchain capable of hosting and executing an extensive range of safe and live CRDT-based I-confluent applications. Our evaluation shows that a coordination-free permissioned blockchain performs significantly better than coordination-based approaches for applications with I-confluent invariant conditions.

REFERENCES

[1] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin. 2020. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *2020 IEEE Symposium on Security and Privacy*. IEEE, 106–118. https://doi.org/10.1109/SP40000.2020.00044

[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*. ACM, 30:1–30:15. https://doi.org/10.1145/3190508.3190538

[3] The Go Authors. 2023. Golang, Go Programming Language. https://golang.org/ Accessed: 2023-09-12.

[4] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* (2014), 185–196. https://doi.org/10.14778/2735508.2735509

[5] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. 2015. Putting Consistency Back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM. https://doi.org/10.1145/2741948.2741972

[6] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguiça. 2015. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *2015 IEEE 34th Symposium on Reliable Distributed Systems*. IEEE, 31–36. https://doi.org/10.1109/SRDS.2015.32

[7] A. Barger, Y. Manevich, H. Meir, and Y. Tock. 2021. A Byzantine Fault-Tolerant Consensus Library for Hyperledger Fabric. In *2021 IEEE International Conference on Blockchain and Cryptocurrency*. IEEE, 1–9. https://doi.org/10.1109/ICBC51069.2021.9461099

[8] J. Bauwens and E. Gonzalez Boix. 2019. Memory Efficient CRDTs in Dynamic Environments. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. ACM, 48–57. https://doi.org/10.1145/3358504.3361231

[9] A. Bessani, J. Sousa, and M. Vukolić. 2017. A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. ACM. https://doi.org/10.1145/3152824.3152830

[10] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. 2014. Riak DT Map: A Composable, Convergent Replicated Dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*. ACM, 1–1. https://doi.org/10.1145/2596631.2596633

[11] C. Cachin and M. Vukolic. 2017. Blockchain Consensus Protocols in the Wild. *CoRR* (2017). arXiv:1707.01873

[12] M. Capretto, M. Ceresa, A. F. Anta, A. Russo, and C. Sánchez. 2022. Setchain: Improving Blockchain Scalability with Byzantine Distributed Sets and Barriers. arXiv:2206.11845

[13] S. J. Castiñeira and A. Bieniusa. 2015. Collaborative Offline Web Applications Using Conflict-free Replicated Data Types. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. ACM. https://doi.org/10.1145/2745947.2745952

[14] J. A. Chacko, R. Mayer, and H-A. Jacobsen. 2021. *Why Do My Blockchain Transactions Fail? A Study of Hyperledger Fabric*. ACM, 221–234. https://doi.org/10.1145/3448016.3452823

[15] J. A. Chacko, R. Mayer, and H.-A. Jacobsen. 2023. How To Optimize My Blockchain? A Multi-Level Recommendation Approach. *Proc. ACM Manag. Data* (2023). https://doi.org/10.1145/3588704

[16] R. Chaganti, B. Bhushan, and V. Ravi. 2022. The Role of Blockchain in DDoS Attacks Mitigation: Techniques, Open Challenges and Future Directions. arXiv:2202.03617

[17] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD*. ACM, 275–290. https://doi.org/10.1145/3183713.3196898

[18] G. A. Di Luna, E. Anceaume, and L. Querzoni. 2020. Byzantine Generalized Lattice Agreement. In *IEEE IPDPS*.

[19] S. Duan, M. K. Reiter, and H. Zhang. 2017. Secure Causal Atomic Broadcast, Revisited. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 61–72. https://doi.org/10.1109/DSN.2017.64

[20] Cloud Native Computing Foundation. 2023. gRPC, a High Performance, Open-Source Universal RPC Framework. https://grpc.io/ Accessed: 2023-10-10.

[21] R. Friedman and K. Birman. 1996. *Trading Consistency for Availability in Distributed Systems*. Technical Report.

[22] H. S. Galal and A. M. Youssef. 2019. Verifiable Sealed-Bid Auction on the Ethereum Blockchain. In *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 265–278.

[23] S. Gilbert and N. Lynch. 2012. Perspectives on the CAP Theorem. *Computer* 45 (2012), 30–36. https://doi.org/10.1109/MC.2011.389

[24] Google. 2021. LevelDB. https://github.com/google/leveldb Accessed: 2023-10-08.

[25] C. Gorenflo, S. Lee, L. Golab, and S. Keshav. 2019. FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second. (2019), 455–463. https://doi.org/10.1109/BLOC.2019.8751452

[26] G. Greenspan. 2015. Multichain Private Blockchainm White Paper. , 57–60 pages. http://www.multichain.com/download/MultiChain-White-Paper.pdf Accessed: 2023-08-28.

[27] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovič, and D.-A. Seredinschi. 2019. The Consensus Number of a Cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. ACM, 307–316. https://doi.org/10.1145/3293611.3331589

[28] M. Hearn and R. G. Brown. 2016. Corda: A Distributed Ledger. *Corda Technical White Paper* 2016 (2016).

[29] J. M. Hellerstein. 2010. The Declarative Imperative: Experiences and Conjectures in Distributed Logic. *SIGMOD Rec.* (2010), 5–19. https://doi.org/10.1145/1860702.1860704

[30] J. Huang, D. He, M. S. Obaidat, P. Vijayakumar, M. Luo, and Kim-Kwang R. Choo. 2021. The Application of the Blockchain Technology in Voting Systems: A Review. *ACM Comput. Surv.* (2021). https://doi.org/10.1145/3439725

[31] M. Imam, S. Takiar, and J. Wang. 2017. RAMBLE: Reliable Asynchronous Messaging for Byzantine Linked Entities. (2017).

[32] K. Jannes, B. Lagaisse, and W Joosen. 2021. OWebSync: Seamless Synchronization of Distributed Web Clients. *IEEE Transactions on Parallel and Distributed Systems* (2021), 2338–2351. https://doi.org/10.1109/TPDS.2021.3066276

[33] T. Jungnickel and L. Oldenburg. 2017. Pluto: The CRDT-Driven IMAP Server. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*. ACM. https://doi.org/10.1145/3064889.3064891

[34] K. Karlsson, W. Jiang, S. Wicker, D. Adams, E. Ma, R. van Renesse, and H. Weatherspoon. 2018. Vegvisir: A Partition-Tolerant Blockchain for the Internet-of-Things. In *2018 IEEE 38th International Conference on Distributed Computing Systems*. IEEE, 1150–1158. https://doi.org/10.1109/ICDCS.2018.00114

[35] S. Kim, Y. Kwon, and S. Cho. 2018. A Survey of Scalability Solutions on Blockchain. In *2018 International Conference on Information and Communication Technology Convergence*. 1204–1207. https://doi.org/10.1109/ICTC.2018.8539529

[36] M. Kleppmann. 2020. Automerge, A JSON-like CRDT. https://github.com/automerge/automerge Accessed: 2023-10-11.

[37] M. Kleppmann and A. R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* (2017), 2733–2746. https://doi.org/10.1109/TPDS.2017.2697382

[38] M. Kleppmann and H. Howard. 2020. Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases. *CoRR* (2020). arXiv:2012.00472

[39] 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*. ACM, 154–178. https://doi.org/10.1145/3359591.3359737

[40] S. Laddad, C. Power, M. Milano, A. Cheung, and J. M. Hellerstein. 2022. Katara: Synthesizing CRDTs with Verified Lifting. *Proc. ACM Program. Lang.* (2022). https://doi.org/10.1145/3563336

[41] L. Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

[42] L. Lamport. 2005. Generalized Consensus and Paxos. (2005).

[43] L. Lamport. 2006. Lower Bounds for Asynchronous Consensus. *Distributed Computing* (2006), 104–125. https://doi.org/10.1007/s00446-006-0155-x

[44] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary. In *OSDI*. USENIX Association, 265–278.

[45] J. Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. 2014. Warranties for Faster Strong Consistency. In *USENIX NSDI*. USENIX Association, 503–517.

[46] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 401–416. https://doi.org/10.1145/2043556.2043593

[47] Y. Mao, Z. Liu, and H.-A. Jacobsen. 2022. Reversible Conflict-Free Replicated Data Types. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*. ACM, 295–307. https://doi.org/10.1145/3528535.3565252

[48] J.-P. Martin and L. Alvisi. 2006. Fast Byzantine Consensus. (2006), 402–411. https://doi.org/10.1109/DSN.2005.48

[49] U. Maurer. 1996. Modelling a Public-Key Infrastructure. In *European Symposium on Research in Computer Security*. Springer, 325–350.

[50] D. Mealha, N. Preguiça, M. C. Gomes, and J. Leitão. 2019. Data Replication on the Cloud/Edge. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*. ACM. https://doi.org/10.1145/3301419.3323973

[51] J. P. Morgan Chase. 2018. A Permissioned Implementation of Ethereum. https://github.com/ConsenSys/quorum Accessed: 2023-10-08.

[52] M. Najafzadeh, M. Shapiro, and P. Eugster. 2018. Co-Design and Verification of an Available File System. In *Verification, Model Checking, and Abstract Interpretation*. Springer International Publishing, 358–381.

[53] S. Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System.

[54] P. Nasirifard, R. Mayer, and H.-A. Jacobsen. 2019. FabricCRDT: A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains. In *Proceedings of the 20th International Middleware Conference*. ACM, 110–122. https://doi.org/10.

1145/3361525.3361540

[55] P. Nasirifard, R. Mayer, and H.-A. Jacobsen. 2022. OrderlessChain: A CRDT-Enabled Blockchain without Total Global Order of Transactions: Poster Abstract. In *Proceedings of the 23rd International Middleware Conference Demos and Posters*. ACM, 5–6. https://doi.org/10.1145/3565386.3565486

[56] P. Nasirifard, R. Mayer, and H.-A. Jacobsen. 2022. OrderlessFile: A CRDT-Enabled Permissioned Blockchain for File Storage: Poster Abstract. In *Proceedings of the 23rd International Middleware Conference Demos and Posters*. ACM, 15–16. https://doi.org/10.1145/3565386.3565491

[57] P. Nasirifard, R. Mayer, and H.-A. Jacobsen. 2022. OrderlessFL: A CRDT-Enabled Permissioned Blockchain for Federated Learning: Poster Abstract. In *Proceedings of the 23rd International Middleware Conference Demos and Posters*. ACM, 7–8. https://doi.org/10.1145/3565386.3565487

[58] 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 2016 ACM International Conference on Supporting Group Work*. ACM, 39–49. https://doi.org/10.1145/2957276.2957310

[59] D. Ongaro and J. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference*. USENIX Association, 305–319.

[60] Oracle. 2021. Read-Your-Writes Consistency. https://bit.ly/3dIAXOp Accessed: 2023-09-20.

[61] P. E. O'Neil. 1986. The Escrow Transactional Method. *ACM Trans. Database Syst.* (1986), 405–430. https://doi.org/10.1145/7239.7265

[62] F. Pedone and A. Schiper. 1999. Generic Broadcast. In *Distributed Computing*. Springer-Verlag, 94–108.

[63] M. Pires, S. Ravi, and R. Rodrigues. 2017. Generalized Paxos Made Byzantine (and Less Complex). In *Stabilization, Safety, and Security of Distributed Systems*. Springer International Publishing, 203–218.

[64] N. Preguiça. 2018. Conflict-free Replicated Data Types: An Overview. arXiv:1806.10254

[65] Hyperledger Project. 2023. Hyperledger Caliper. https://hyperledger.github.io/caliper/ Accessed: 2023-10-02.

[66] J. Qi, X. Chen, Y. Jiang, J. Jiang, T. Shen, S. Zhao, S. Wang, G. Zhang, L. Chen, M. H. Au, and H. Cui. 2021. BIDL: A High-Throughput, Low-Latency Permissioned Blockchain Framework for Datacenter Networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. ACM, 18–34. https://doi.org/10.1145/3477132.3483574

[67] P. Raykov, N. Schiper, and F. Pedone. 2011. Byzantine Fault-Tolerance with Commutative Commands. In *Principles of Distributed Systems*. 329–342.

[68] L. S. Sankar, M. Sindhu, and M. Sethumadhavan. 2017. Survey of Consensus Protocols on Blockchain Applications. In *2017 4th International Conference on Advanced Computing and Communication Systems*. 1–5. https://doi.org/10.1109/ICACCS.2017.8014672

[69] M. Shapiro, A. Bieniusa, N. M. Preguiça, V. Balegas, and C. Meiklejohn. 2018. Just-Right Consistency: Reconciling Availability and Safety. *CoRR* (2018). arXiv:1801.06340

[70] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. 2011. Conflict-Free Replicated Data Types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[71] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich. 2019. Blurring the Lines Between Blockchains and Database Systems: The Case of Hyperledger Fabric. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 105–122. https://doi.org/10.1145/3299869.3319883

[72] A. Shoker, H. Yactine, and C. Baquero. 2017. As Secure as Possible Eventual Consistency: Work in Progress. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*. ACM. https://doi.org/10.1145/3064889.3064895

[73] D. Sun, S. Xia, C. Sun, and D. Chen. 2004. Operational Transformation for Collaborative Word Processing. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*. ACM, 437–446. https://doi.org/10.1145/1031607.1031681

[74] V. Tao, M. Shapiro, and V. Rancurel. 2015. Merging Semantics for Conflict Updates in Geo-Distributed File Systems. In *Proceedings of the 8th ACM International Systems and Storage Conference*. ACM. https://doi.org/10.1145/2757667.2757683

[75] 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*. ACM. https://doi.org/10.1145/3038912.3052673

[76] P. van Hardenberg and M. Kleppmann. 2020. PushPin: Towards Production-Quality Peer-to-Peer Collaboration. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. ACM. https://doi.org/10.1145/3380787.3393683

[77] S. Weiss, P. Urso, and P. Molli. 2009. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 404–412. https://doi.org/10.1109/ICDCS.2009.75

[78] M. Whittaker and J. M. Hellerstein. 2020. Checking Invariant Confluence, In Whole or In Parts. *SIGMOD* (2020), 7–14. https://doi.org/10.1145/3422648.3422651

[79] H. Y. Wu, L. Jie Li, H.-Y. Paik, and S. S. Kanhere. 2021. MEChain: A Multi-layer Blockchain Structure with Hierarchical Consensus for Secure EHR System. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 976–987. https://doi.org/10.1109/TrustCom53373.2021.00136

[80] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. ACM, 347–356. https://doi.org/10.1145/3293611.3331591

[81] G. Younes, A. Shoker, P. S. Almeida, and C. Baquero. 2016. Integration Challenges of Pure Operation-based CRDTs in Redis. In *First Workshop on Programming Models and Languages for Distributed Computing*. ACM, 7:1–7:4. https://doi.org/10.1145/2957319.2957375

[82] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. 2015. Write Fast, Read in the Past: Causal Consistency for Client-Side Applications. In *Proceedings of the 16th Annual Middleware Conference*. ACM, 75–87. https://doi.org/10.1145/2814576.2814733

[83] W. Zhao, M. Babi, W. Yang, X. Luo, Y. Zhu, J. Yang, C. Luo, and Mary Y. 2016. Byzantine Fault Tolerance for Collaborative Editing with Commutative Operations. In *2016 IEEE International Conference on Electro Information Technology*. IEEE, 246–251. https://doi.org/10.1109/EIT.2016.7535248