Pezhman Nasirifard

Reducing Coordination in Permissioned Blockchains with Conflict-free Replicated Data Types

Technische
Universität
München

Technische Universität München

TUM School of Computation, Information and Technology

Reducing Coordination in Permissioned Blockchains with Conflict-free Replicated Data Types

Pezhman Nasirifard

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technische Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:	Prof. Dr.-Ing. Pramod Bhatotia
Prüfer*innen der Dissertation:	1. Prof. Dr. Hans-Arno Jacobsen
	2. Prof. Dr. Kaiwen Zhang

Die Dissertation wurde am 22.12.2022 bei der Technische Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 12.05.2023 angenommen.

Abstract

Since the introduction of *Bitcoin*, numerous permissionless and permissioned blockchains have been proposed with the prospect of disrupting various industries. The main property contributing to their popularity is offering decentralized trust and providing a safe and live environment for the execution of decentralized applications in Byzantine distributed systems. Blockchains often rely on coordination-based consensus protocols to offer trust and *Byzantine Fault Tolerance (BFT)*. Furthermore, these coordination-based protocols serialize the transactions into a global order to preserve the correctness of the application's state stored on the blockchain. However, the required coordination to reach consensus has been a bottleneck for scalability and transaction throughput and has induced high transaction latency. These problems hinder the widespread adoption of blockchains despite their significant potential. To improve the scalability, throughput, and transaction latency of permissioned blockchains, we proposed two *Conflict-free Replicated Data Types-based (CRDTs)* approaches to reduce the coordination in the system.

We first provide a CRDT-based scalability solution for Hyperledger Fabric (Fabric), materialized in a new system called FabricCRDT. Fabric is one of the most prominent permissioned blockchains and provides a complete ecosystem for developing production-grade decentralized applications. However, Fabric's coordination-based protocol induces high latency, which results in a high percentage of concurrent and conflicting transactions. Consequently, Fabric's optimistic concurrency control mechanism causes the failure of these conflicting transactions. Our proposed solution in FabricCRDT detects the conflicting concurrent transactions automatically and employs CRDT-based techniques to resolve the values of conflicting transactions and merges them without data loss or corruption. Furthermore, to facilitate the adoption of permissioned blockchains, we maintain FabricCRDT backward compatible with existing Fabric applications. The evaluation of FabricCRDT and its comparison to Fabric demonstrate the significantly increased throughput of successful transactions on FabricCRDT compared to Fabric. Although FabricCRDT's CRDT-based approach does not require coordination for the correct execution of applications, it uses Fabric's coordination-based protocol for backward compatibility purposes.

Second, we propose OrderlessChain, a novel permissioned blockchain based on a novel BFT coordination-free protocol without serializing transactions into a global order.

Although serializability is required to preserve the correctness of many applications, 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 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. By using the properties of permissioned blockchains and CRDTs, OrderlessChain provides a solution for the safe and live execution of I-confluent applications in a Byzantine environment. We extensively evaluated OrderlessChain and compared its coordination-free protocol to the coordination-based protocols of Fabric and FabricCRDT. Our evaluation confirms the significant potential of our BFT coordination-free approach as a scalable alternative over coordination-based permissioned blockchains for I-confluent applications. Furthermore, to demonstrate the applicability of OrderlessChain to other domains, we adapted OrderlessChain with novel CRDTs to offer a blockchain-based secure *federated learning* solution and a private decentralized file storage system.

Zusammenfassung

Seit der Einführung von Bitcoin wurden zahlreiche Permissionless und Permissioned Blockchains vorgeschlagen, die die Aussicht haben, verschiedene Branchen zu verändern. Die Haupteigenschaft, die zu ihrer Beliebtheit beiträgt, ist das Angebot von dezentralem Vertrauen und die Bereitstellung einer *Safe* und *Live* Umgebung für die Ausführung von dezentralen Anwendungen in byzantinischen verteilten Systemen. Blockchains basieren häufig auf koordinationsbasierten Konsensprotokollen, um Vertrauen und *BFT (Byzantine Fault Tolerance)* zu bieten. Darüber hinaus serialisieren diese koordinationsbasierten Protokolle die Transaktionen in einer global gültigen Sortierung, um die Korrektheit des auf der Blockchain gespeicherten Anwendungsdaten zu bewahren. Die für die Konsensfindung erforderliche Koordinierung hat sich jedoch als Engpass für die Skalierbarkeit und den Transaktionsdurchsatz erwiesen und zu einer hohen Transaktionslatenz geführt. Diese Probleme behindern die breite Einführung von Blockchains trotz ihres großen Potenzials. Um die Skalierbarkeit, den Durchsatz und die Transaktionslatenz von Permissioned Blockchains zu verbessern, in dieser Arbeit werden zwei auf *Conflict-free Replicated Data Types (CRDTs)* basierende Ansätze vorgeschlagen, um die Koordination im System zu reduzieren.

Zunächst wird eine CRDT-basierte Skalierbarkeitslösung für Hyperledger Fabric (Fabric) geboten, die in einem neuen System namens FabricCRDT realisiert ist. Fabric ist eine der bekanntesten Permissioned Blockchains und bietet ein komplettes Ökosystem für die Entwicklung produktionsreifer dezentraler Anwendungen. Das koordinationsbasierte Protokoll von Fabric führt jedoch zu einer hohen Latenzzeit, was einen hohen Prozentsatz an gleichzeitigen und konfliktreichen Transaktionen zur Folge hat. Folglich führt der optimistische Gleichzeitigkeitssteuerungsmechanismus von Fabric zum Scheitern dieser widersprüchlichen Transaktionen. Die von uns vorgeschlagene Lösung in FabricCRDT erkennt sich gegenseitig beeinträchtigenden Transaktionen automatisch und setzt CRDT-basierte Techniken ein, um die Konflikte aufzulösen und sie ohne Datenverlust oder Beschädigung zusammenzuführen. Um die Einführung von Permissioned Blockchains zu erleichtern, ist FabricCRDT rückwärtskompatibel mit bestehenden Fabric-Anwendungen. Die Evaluierung von FabricCRDT und der Vergleich mit Fabric zeigen, dass der Durchsatz erfolgreicher Transaktionen auf FabricCRDT im Vergleich zu Fabric signifikant höher ist. Obwohl der CRDT-basierte Ansatz von FabricCRDT keine Koordination für die korrekte Ausführung von

Anwendungen erfordert, verwendet er das koordinationsbasierte Protokoll von FABRIC, um Abwärtskompatibilität zu gewährleisten.

Zweitens wird ORDERLESSCHAIN vorgeschlagen, eine neuartige Permissioned Blockchain, die auf einem neuartigen koordinationsfreien BFT-Protokoll basiert, ohne Transaktionen in eine globale Reihenfolge zu bringen. Obwohl die Serialisierbarkeit erforderlich ist, um die Korrektheit vieler Anwendungen zu bewahren, gibt es Korrektheitsanforderungen auf Anwendungsebene, die nicht von der Reihenfolge der Transaktionen abhängen, bekannt als *Invariant Confluence (I-confluence)*. Die I-confluence-Anwendungen können koordinationsfrei ausgeführt werden und profitieren von der besseren Skalierbarkeit im Vergleich zu koordinationsbasierten Ansätzen. Die *Safety* und *Liveness* von I-confluence-Anwendungen wurden in nicht-byzantinischen Umgebungen untersucht, aber die korrekte Ausführung solcher Anwendungen bleibt eine Herausforderung in byzantinischen koordinationsfreien Umgebungen. Durch die Nutzung der Eigenschaften von Permissioned Blockchains und CRDTs bietet ORDERLESSCHAIN eine Lösung für die Safe und Live Ausführung von I-confluence-Anwendungen in einer byzantinischen Umgebung. ORDERLESSCHAIN wurde evaluiert und sein koordinationsfreies Protokoll mit den koordinationsbasierten Protokollen von FABRIC und FABRICCRDT verglichen. Die Evaluierung bestätigt das erhebliche Potenzial unseres koordinationsfreien BFT-Ansatzes als skalierbare Alternative zu koordinationsbasierten Permissioned Blockchains für I-confluence-Anwendungen. Um die Anwendbarkeit von ORDERLESSCHAIN auf andere Domänen zu demonstrieren, wurde ORDERLESSCHAIN mit neuartigen CRDTs angepasst, um eine Blockchain-basierte sichere *Federated Learning*-Lösung und ein privates dezentrales Dateispeichersystem anzubieten.

Acknowledgments

This doctoral dissertation and the included research were carried out at the Department of Computer Science at the Technical University of Munich under the supervision of Prof. Hans-Arno Jacobsen.

I would like to express my sincere gratitude to Prof. Hans-Arno Jacobsen for his motivation, continuous support, and guidance he provided over the years. He offered me a great deal of freedom during my academic journey and encouraged me to go in different exciting directions. I also sincerely thank Prof. Ruben Mayer for advising me, continuously providing valuable feedback on my work, and being a reliable supporter. I would like to thank Prof. Viktor Leis for supporting me at the end of my Ph.D.

I would like to thank the rest of my thesis committee: Prof. Kaiwen Zhang from École de technologie supérieure for agreeing to be my second examiner and Prof. Pramod Bhatotia for accepting to chair the committee.

Many thanks go to all my former and present colleagues. The pandemic showed me the significance of great colleagues in making a Ph.D. more enjoyable. I would like to especially thank Dr. Jose Rivera and Dr. Martin Jergler for providing the stepping stone to our chair and Dr. Christoph Doblander, Alexander Isenko, and Herbert Woisetschläger for investing countless hours in maintaining our infrastructure and offering valuable help and insights. I sincerely thank Jeeta Ann Chacko for being an awesome officemate.

I want to sincerely thank all my students I was honored to supervise and lecture. Their work provided me with valuable input and made the collaboration on various publications besides the works discussed in this dissertation possible.

Outside the academic world, I greatly thank my dearest parents, Motahareh and Nejatali, for making everything possible and providing me with all the opportunities I have had in life. I sincerely thank all my dear friends for supporting me and being interested in and asking about my work. Last but not least, my infinite gratitude goes to Prof. Christian Strobel for standing by my side on my good and several bad days, emotionally and mentally supporting me with all he could give, and making my every day less challenging. I am very proud of you. You Rock!

Contents

1 Introduction

The emergence of blockchain technologies offered the prospect of disrupting a wide range of domains and industries, from e-government to financial and healthcare sectors [1, 2, 3]. The main property of blockchains that contributed to their explosive popularity is the decentralized trust that enables individuals and organizations to execute decentralized applications safely and transactions in Byzantine distributed systems without requiring centralized trust [4, 5].

Despite the significant potential and the initial enormous hype surrounding blockchains, its various open problems hinder the wide adoption of blockchain technologies [6, 7]. The problems such as low scalability, limited transaction throughput, high transaction latency, and several more have made the realization of several use cases challenging [8, 9]. The low scalability and throughput problems are rooted in the employed coordination-based protocols, which require coordination among nodes to reach consensus in order to ensure decentralized trust and execute applications correctly [8, 10, 11].

This work proposes solutions for reducing the coordination and improving the scalability and adoption of existing and novel permissioned blockchains. First, we propose a *Conflict-free Replicated Data Type-based (CRDT)* [12] approach to significantly improve the throughput of the prominent permissioned blockchain of HYPERLEDGER FABRIC [13] realized in a new system called FABRICCRDT. Second, we use properties of CRDTs and permissioned blockchains to offer a novel scalable and coordination-free permissioned blockchain named OrderlessChain for the safe and trusted execution of decentralized applications in Byzantine environments.

1.1 Motivation

Since the introduction of *Bitcoin* in 2008 by Satoshi Nakamoto [14], the popularity of blockchain technologies has increased rapidly [1, 3, 15]. The primary property contributing to blockchain's fast growth is the decentralized trust that enables the safe execution of applications in trustless Byzantine environments. Blockchains offer decentralized trust by employing various *Byzantine Fault Tolerant (BFT)* coordination-based protocols [10, 16, 17]. Although BFT protocols have existed long before the introduction of Bitcoin and have been studied comprehensively in academia [18, 19], many blockchain systems provide a complete solution for hosting decentralized applications, processing transactions, and storing the applications' state in immutable ledgers. Since the applicability of Bitcoin is limited mainly to cryptocurrency-related applications, several permissionless and permissioned blockchains have been proposed to realize a wide range of use cases in numerous fields and industries, including finance, e-government, IoT, healthcare, and supply-chain management [1, 2, 3, 20, 21, 22].

Despite the potential of blockchains for developing a wide range of decentralized applications, and even with the introduction of many blockchains that provide clients with novel ways of processing transactions and storing data in trustless environments [13, 23, 24, 25, 26, 27], the limited scalability of several blockchains due to their coordination-based protocols hinders their widespread adoptions [15, 26, 28]. The scalability and throughput of several existing blockchains fall significantly behind existing non-BFT distributed systems. Their significantly limited performance makes realizing blockchain-based use cases unrealistic and prohibitively inefficient [4, 15].

The low scalability is severe in the case of popular permissionless blockchains such as Bitcoin and *Ethereum* [23], which process merely tens of transactions per second, inducing low throughput, very high transaction latency, and very volatile transaction processing fees [7, 9]. The primary reason for the poor performance is the employed coordination-based BFT *Proof-of-Work-based (PoW)* protocols [6]. Furthermore, the PoW-based protocols are extremely energy intensive and contribute significantly to CO_2 emissions and intensifying climate change [29, 30]. Although several scalability solutions have been proposed to address the scalability of permissionless blockchains ranging from optimizing the internal components of PoW-based systems [31] to various novel coordination-based BFT consensus protocols such as the *Proof-of-Stake* protocols [32], the coordination required to reach consensus remains a bottleneck [10, 16, 17].

In contrast to permissionless blockchains, where participants can freely join the network and process transactions, permissioned blockchains exist where the identity of participants is known, and only authenticated authorized participants can join the network [5, 27]. Permissioned blockchains such as Hyperledger Fabric (Fabric) [13] use this permissioned property to offer significantly more scalable coordination-based protocols with higher transaction throughput and lower latency than their permissionless blockchain counterparts. Despite the improved scalability of permissioned blockchains, their employed BFT and non-BFT protocols, such as *Raft* [33], *Practical-BFT* [34], or *BFT-SMaRt* [35], require coordination to reach a consensus which also causes scalability bottlenecks [8, 36].

According to the *Scalability Trilemma* [37, 38], scaling blockchains without limiting security and decentralization stays challenging. Decreasing coordination is critical for improving the scalability of any distributed systems, and in order to materialize the true potentials of permissioned blockchains and enable their widespread adoption, improving the scalability of coordination-based protocols through reducing coordination and offering high transaction throughput with low latency plays a vital role [39, 40].

1.2 Problem Statement

We aim to reduce coordination in permissioned blockchains, and consequently improve their scalability, facilitate their adoptions, and enable realizing a wide range of use cases across different domains. We realize two primary research objectives in this dissertation:

1. *FABRIC is currently among the most prominent permissioned blockchain for various enterprise use cases [8, 36]. However, its coordination-based protocol is a scalability bottleneck. We seek to provide a scalability solution for FABRIC to improve its throughput.*
2. *We aim to reduce coordination in BFT permissioned blockchains to improve their scalability while offering a safe system for the correct execution of a wide range of decentralized applications in a trustless Byzantine environment.*

In both objectives, reducing coordination is an essential requirement. However, besides facilitating decentralized trust, coordination to reach consensus enables the blockchain to agree on the total global order of transactions for a serialized execution. Serializability is required to preserve the correctness of the application's state stored on the blockchain [39]. For example, in the case of a decentralized banking application, the application's correctness definition may require the system to prevent a client's negative account balance. As every node in the blockchain sequentially executes the transactions in the same order, preserving this correctness requirement is relatively straightforward. However, the coordination-based serialization for preserving the application's correctness is a scalability bottleneck [25, 26].

For use cases where the identity of participants is known, permissioned blockchains, such as FABRIC, constitute a viable solution [5, 25]. In permissioned blockchains, despite the known identity of participants, they do not necessarily trust each other. FABRIC takes advantage of its permissioned property to implement a coordination-based protocol for a trusted execution of transactions and for preserving the application's correctness. FABRIC uses an optimistic three-phase *execute-order-validate (EOV)* protocol, where the transactions are initially concurrently executed without changing the application's state on the ledger. Then, the transactions are serialized into a total global order and are validated and committed. The independent concurrent transactions, which read and write individual parts of the application's state, are valid and committed successfully. However, the concurrent transactions that read and write the identical parts of the application's state may conflict and are invalidated and fail to commit. This technique is inspired by the concurrency control mechanisms implemented in several distributed databases, which preserve the application's correctness while enabling the concurrent execution of transactions for improved scalability, throughput, and latecy [41, 42, 43].

However, the latency between the start (execution) and end (commit) of a transaction in Fabric is hundreds of milliseconds to seconds due to additional steps required to ensure the trusted execution of transactions. The added latency is significantly higher than the latency for processing transactions in distributed databases that use a similar technique [8]. Consequently, the added latency increases the probability of the arrival of dependent concurrent transactions, causing their failures. Several studies show that realistic enterprise use cases, from digital voting applications to medical record management systems, contain highly concurrent and dependent transactions, and executing such applications on Fabric results in up to 90% transaction failures [8, 36, 44]. Such a high transaction failure rate is a significant bottleneck to Fabric. Furthermore, once a transaction fails, the only option for clients is to create a new transaction and resubmit, which adds to the complexity of Fabric's application development.

Several works propose various approaches for decreasing the transaction conflict rate and increasing the throughput of successful transactions on Fabric [36, 44, 45, 46, 47]. Some studies propose various approaches to reordering transactions in Fabric to decrease the dependencies among concurrent transactions and reduce transaction failure rates or early abort the conflicting trasanctions [36, 44, 45]. Also, some studies propose different optimization approaches to decrease Fabric's internal latency for processing transactions and thus increase its throughput [46, 47, 48, 49]. However, non of the existing works offer a solution to eliminating the transaction failure of concurrent transactions. Therefore, we aim to propose a solution that enables Fabric to manage the conflicting transactions internally without rejecting the transactions or coordinating with other nodes in the network. The proposed solution can significantly improve the throughput of Fabric and simplify the application development process.

Although we aim to eliminate the failures of concurrent transactions, reducing coordination offers the most significant contribution to improving the performance and scalability of permissioned blockcains [39]. A coordination-free permissioned blockchain enables the concurrent execution of transactions, leading to higher throughput and lower latency. However, simply eliminating the coordination may jeopardize the correctness depending on the application's requirements. For example, in the case of the banking application with the non-negative account balance requirement, a coordination-free permissioned blockchain cannot preserve this requirement [39, 40].

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 [39]. For example, transactions that only deposit funds to an account can execute 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. Bailis et al. [39] 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 the improved throughput and latency of taking advantage of coordination-free approaches.

However, preserving the safety and liveness of applications in a Byzantine environment depends on paying a high coordination cost in other systems, and existing works rely on coordination rounds to prevent the Byzantine participants from intentionally violating the application's correctness [50, 51, 52, 53]. We seek to offer a solution to a BFT coordination-free environment where I-confluent applications remain safe and live. Therefore, we benefit from improved scalability while ensuring trust in a Byzantine environment.

1.3 Approach

In summary, the objectives of this dissertation are twofold. We first aim to provide a scalability solution for Fabric to significantly increase its successful transaction throughput. We achieve this goal by offering a CRDT-based [12] approach for internally resolving transaction conflicts and preventing the failure of concurrent transactions. Second, we seek to offer a coordination-free permissioned blockchain for the safe and live execution of decentralized applications without paying the high coordination cost. We offer a BFT coordination-free protocol that uses the permissioned property of the system and CRDTs for the correct execution of transactions without coordination. The following section briefly explains our approaches to realizing these objectives.

1.3.1 A CRDT-based Approach to Fabric

The high transaction latency induced by Fabric's three-phase protocol and the relatively high dependency among transactions in real-world use cases causes the failure of a high number of concurrent dependent transactions by Fabric's optimistic concurrency control mechanism [8, 36]. To address the high rate of conflicting transaction failures, we propose an approach to internally resolving the conflicts of concurrent transactions and preventing their consequent failures. Thereby we eliminate the failures caused by Fabric's concurrency control mechanism and successfully commit every valid transaction while preserving the application's correctness.

Our approach to resolving the conflicts of concurrent transactions uses CRDTs, which are abstract data types that converge to the same state on distributed nodes without coordination among nodes to reach a consensus. As we aim to improve the scalability of Fabric and increase its adoption, we realize the significance of maintaining our approach backward-compatible with the available applications developed for Fabric. Hence, we extend Fabric with CRDT-enabled functionalities into a new system called FabricCRDT, which offers the same functionalities as Fabric regarding hosting and executing decentralized applications while offering a novel environment for executing CRDT-enabled applications.

FabricCRDT follows the same three-phase protocol as Fabric for executing and ordering transactions. However, FabricCRDT bypasses Fabric's optimistic concurrency control mechanism and employs CRDT-based techniques for detecting conflicting transactions and merging their values without causing data loss or corruption. A plethora of CRDTs, from *CRDT Counters* to *Maps* and *Sets* [12, 54], can be realized on FabricCRDT. Although our proposed solution is CRDT-agnostic, the specification of CRDTs must be supported in the application execution environment of FabricCRDT. In order to offer the potential of developing a wide range of use cases, we enable a *JSON CRDT-based* [54] approach that encapsulates JSON objects. Since JSON is a standard data structure used in various applications and systems, enabling FabricCRDT with JSON CRDT functionalities provides a solution for realizing these applications on FabricCRDT, benefiting from its decentralized trust.

To understand the potentials and limitations of FabricCRDT over Fabric, we perform extensive evaluations of both systems by developing IoT-based applications. Our evaluations demonstrate that our approach can eliminate conflicting transaction failures and significantly improve successful transaction throughput over Fabric. Also, by avoiding the failures of conflicting transactions, FabricCRDT simplifies the complexity of developing decentralized CRDT-compatible applications.

1.3.2 A Coordination-free Permissioned Blockchain

As explained, reducing coordination is critical for improving the scalability of permissioned blockchains. However, decreasing and eliminating coordination in Byzantine distributed systems is challenging without paying the high coordination cost while providing a safe and live environment for executing applications and preserving their invariant conditions. We identify that the invariant conditions of applications can be classified into I-confluent and non-I-confluent invariant conditions, where I-confluent invariant conditions can be preserved in coordination-free non-Byzantine environments [55].

To address the problem of preserving I-confluent invariant conditions in Byzantine environments, we propose an approach for a BFT coordination-free two-phase *execution-commit* protocol for the safe and live execution of I-confluent applications. Our proposed protocol is materialized in OrderlessChain, a novel, strongly eventually consistent, asynchronous permissioned blockchain. OrderlessChain's network consists of several organizations and clients, where the organizations host and execute applications and store a replica of the applications' state in immutable ledgers. Clients create and submit transactions for executing the applications and interact with the application's state. For processing transactions according to our protocol, clients first submit *proposals* to be executed and signed by organizations. After the successful conclusion of the first phase, clients create and send signed transactions based on the proposals to the organizations to be validated and committed. The application developers assign an *endorsement policy* to every application, which specifies which organizations must execute and sign the proposals for the transaction to be valid. Endorsement policies also establish which organizations must validate and commit transactions. In other words, they specify the trust requirements of the applications. Furthermore, endorsement policies enable OrderlessChain to prevent the Byzantine behavior of malicious clients and organizations without coordination. Honest organizations detect Byzantine behavior by verifying the signature of the organizations and clients according to the endorsement policy.

Since OrderlessChain's protocol uses a coordination-free approach and the transactions are not serialized into a total global order, different organizations validate and commit transactions in different orders. In order to preserve the I-confluent invariant conditions of applications, we must be able to process the transactions in any order while converging to the same state. In order words, the transactions must be *commutative* and *convergent*. Since CRDTs offer such properties, OrderlessChain uses a CRDT-based approach to modeling and implementing decentralized applications using *CRDT Maps*, *Grow-only Counters*, and *Multi-Value Registers* [12, 54] to create commutative and convergent transactions.

To demonstrate the applicability of OrderlessChain to various domains, we developed a few decentralized applications and systems, including a digital voting application, an auction application, a secure and decentralized file storage system, and a private *federated learning (FL)* environment [56]. We extensively evaluate the performance of OrderlessChain and compare the scalability of our coordination-free approach to the coordination-based protocols of FabricCRDT and Fabric. Our evaluations demonstrate the significant scalability gain, highly improved transaction throughput, and decreased latency for hosting and executing I-confluent applications using OrderlessChain's coordination-free approach over coordination-based permissioned blockchains.

1.4 Contributions

The main contributions of FabricCRDT as a scalability solution of Fabric are:

i. We study the applicability of CRDTS to permissioned blockchains since, despite their significant contribution to the scalability of non-Byzantine production-grade systems, their potential for improving the scalability of permissioned blockchains has received limited research attention. We propose a novel approach for enabling CRDTs on the permissioned blockchains that automatically resolve transaction conflicts without data loss, resulting in significantly improved throughput of transactions.

ii. We offer a scalability solution for Fabric to increase the adoption of permissioned blockchains to various fields and industries by extending Fabric with CRDT-enabled functionalities into a new system called FabricCRDT. It remains backward compatible with existing Fabric applications while enabling the realization of new scalable CRDT-based applications, incentivizing the adoption of permissioned blockchains.

iii. We reduce the complexity of developing decentralized applications by eliminating the failures of concurrent and conflicting transactions. Therefore, developers are discharged from the additional effort required to implement mechanisms to handle scenarios containing such failures.

iv. We offer a simplified CRDT-based programming model for developing CRDT-based applications without the typical complexity of CRDT-enabled systems. This programming model requires a minimal learning curve for developers familiar with Fabric. We also provide insights into the appropriate use cases for CRDT-enabled permissioned blockchains.

v. We implement a prototype of FabricCRDT and evaluate and demonstrate our system's improved scalability and throughput over Fabric as a CRDT-enabled permissioned blockchain.

The main contributions of the coordination-free approach of OrderlessChain are:

i. We introduce a novel coordination-free protocol for offering Byzantine fault tolerance without requiring the nodes to coordinate to reach a consensus. By reducing coordination, we significantly improve scalability and transaction throughput and decrease the latency. We also offer proof of the protocol's BFT property.

ii. We demonstrate OrderlessChain, a novel, strongly eventually consistent, and asynchronous permissioned blockchain based on our proposed BFT coordination-free protocol, capable of executing safe and live applications. Our system eliminates the coordination's overhead and significantly improves the throughput and scalability over coordination-based permissioned blockchains.

iii. We present a novel approach for creating Turing-complete decentralized applications based on CRDTs. Our approach preserves the I-confluent invariant conditions of applications in a coordination-free Byzantine environment and ensures their correct executions. We demonstrate that our approach is more scalable than the existing CRDT-enabled permissioned blockchains, and we offer the potential for realizing new use cases.

iv. We implement a complete prototype of OrderlessChain and demonstrate that our system improves throughput and latency for I-confluent applications compared to coordination-based permissioned blockchains. Furthermore, we open-sourced the system code for the public [57].

v. To demonstrate the applicability of OrderlessChain to various domains and industries, we introduce OrderlessFL, an OrderlessChain-based FL system. OrderlessFL uses a novel CRDT for concurrent and asynchronous aggregation of FL models. OrderlessFL offers a safe and private environment for training *Machine Learning* models, where malicious participants cannot tamper with model updates. OrderlessFL is also open-sourced [58].

vi. We also introduce OrderlessFile, an OrderlessChain-based private and distributed file storage system, to demonstrate our system's applicability as an alternative to non-transparent and centralized cloud-based storage systems. OrderlessFile uses a novel CRDT for splitting files into shards which are safely and privately replicated and stored where Byzantine participants cannot tamper with data and violate its integrity. OrderlessFile's code is also open-sourced [59].

Parts of the contents of this dissertation are published or under submission to the following venues:

- P. Nasirifard, R. Mayer, and H.-A. Jacobsen. "FabricCRDT: A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains." In: Proceedings of the

20th International Middleware Conference. Middleware '19. Davis, CA, USA: ACM, 2019 [55].

- P. Nasirifard, R. Mayer, and H.-A. Jacobsen. "OrderlessChain: A CRDT-based Coordination-free Blockchain Without Global Order of Transactions." In: Proceedings of the 24th International Middleware Conference. Middleware '23. Bologna, Italy: ACM, 2023 [60].
- P. Nasirifard, R. Mayer, and H.-A. Jacobsen. "OrderlessChain: A CRDT-Enabled Blockchain without Total Global Order of Transactions." In: Proceedings of the 23rd International Middleware Conference: Demos and Posters. Middleware '22. Quebec, Quebec City, Canada: ACM, 2022 [61].
- P. Nasirifard, R. Mayer, and H.-A. Jacobsen. "OrderlessFL: A CRDT-Enabled Permissioned Blockchain for Federated Learning." In: Proceedings of the 23rd International Middleware Conference: Demos and Posters. Middleware '22. Quebec, Quebec City, Canada: ACM, 2022 [62].
- P. Nasirifard, R. Mayer, and H.-A. Jacobsen. "OrderlessFile: A CRDT-Enabled Permissioned Blockchain for File Storage." In: Proceedings of the 23rd International Middleware Conference: Demos and Posters. Middleware '22. Quebec, Quebec City, Canada: ACM, 2022 [63].

1.5 Organization

The remainder of the dissertation is organized as follows. First, in Chapter 2, we provide the background material on FABRIC and its three-phase protocol for processing transactions. We also introduce CRDTs and discuss their internal mechanisms for resolving conflicts without coordination, and describe the I-confluence concepts for preserving the application's invariant conditions in coordination-free distributed systems.

Chapter 3 introduces the related work of the contributions of this dissertation, including the existing scalability solutions for FABRIC and existing works for executing CRDT-enabled applications in Byzantine and non-Byzantine environments. The chapter also elaborates on the previous studies for reducing coordination in distributed systems for enhancing scalability. We also present state-of-the-art of blockchain-based FL and file storage systems.

Chapter 4 presents our approach to offering the CRDT-enabled permissioned blockchain of FabricCRDT. We first provide the exact cause of transaction failures due to Fabric's optimistic concurrency control mechanism, which we address with FabricCRDT. We introduce our CRDT-based approach to eliminating such failures and elaborate on modeling and creating CRDT-based applications on FabricCRDT. Our evaluation demonstrates the improved scalability and throughput of FabricCRDT over Fabric.

Chapter 5 presents OrderlessChain, our coordination-free approach to executing safe and live I-confluent applications in Byzantine distributed environments. We provide a detailed explanation of OrderlessChain's architecture and protocols and explain our method for realizing several decentralized CRDT-based applications. The significant improvement to the throughput and latency of our coordination-free approach over coordination-based protocol is demonstrated through our evaluations. To establish the applicability of OrderlessChain to other domains, we introduce an OrderlessChain-bassed FL system and file storage system in Chapter 6.

Finally, Chapter 7 covers the concluding remarks on FabricCRDT for eliminating the failure of concurrent and conflicting transactions and on our BFT coordination-free solution materialized in OrderlessChain.

2

Background

In the following chapter, we introduce the required background materials for this dissertation. In Chapter 4, we present FabricCRDT, a scalability solution for Fabric using *Conflict-free Replicated Data Types (CRDTs).* Therefore, in Section 2.1, we provide a detailed explanation of Fabric's components and its protocol for executing, ordering, and committing transactions. We continue in Section 2.2 by giving an overview of CRDTs.

In Chapter 5, we continue to offer a more scalable blockchain system named OrderlessChain. OrderlessChain takes advantage of CRDTs and permissioned properties of permissioned blockchains, such as Fabric, to provide a coordination-free approach for the safe execution of *Invariant Confluent* applications. Hence, we explain the concept of Invariant Confluence and its independence from coordination in a distributed environment in Section 2.3.

2.1 Hyperledger Fabric Permissioned Blockchain

Hyperledger Fabric (Fabric) is an open-source permissioned blockchain initiated by the *Linux Foundation* [13]. Here, we explain the specifications and structure of Fabric v1.4 since we used this long-term supported version in our work.

2.1.1 Architecture

In contrast to permissionless blockchains such as Bitcoin and Ethereum [14, 23], where every entity can freely join the network and process transactions, joining a permissioned blockchain, such as FABRIC, is limited to authenticated and authorized participants.

FABRIC provides a complete ecosystem for hosting decentralized applications and processing transactions for executing them and interacting with their state. FABRIC's ecosystem offers a wide range of features and services, including decentralized trust, storing the application state on the ledger, an isolated transaction execution environment, private communication channels, and sophisticated identity and membership management. Blockchain developers can use general-purpose programming languages like *Go Language* [64] or *JavaScript* to implement *smart contracts* known as *chaincodes*. The chaincode consists of several functions that encapsulate the application's logic. The clients interact with chaincodes by creating and submitting transactions. For modifying the data stored on the ledger, developers use the *chaincode shim* in the chaincode. The shim is a language-specific library provided by FABRIC's ecosystem and provides APIs to read and write data from and to the ledger.

The two main components of FABRIC's network are *peers* and *orderers*. Each peer belongs to an *organization*, which defines the trust boundary of the system. Each organization may consist of several peers. However, every peer belongs to only one organization. The peers within one organization trust each other. However, although the identity of a peer from one organization is known to the peer of other organizations, the peers of different organizations do not trust each other. The peers also use private communication channels to communicate with other peers and orderers.

An *Endorsement Policy* is assigned to each application hosted on FABRIC. The endorsement policy specifies which peers from which organizations must execute and commit the transactions. In other words, the application's endorsement policy defines the application's decentralized trust requirement.

Peers are responsible for hosting the chaincodes, executing transactions, and storing the data on their local copy of the ledger. The peer's ledger consists of two components: (1)

an append-only hash-chain log and (2) a *world state* key-value database such as *LevelDB* or *CouchDB* [42, 65]. The hash-chain log contains every failed or successful transaction the peer has received since the beginning of time. The database represents the current state of the application's data stored on the ledger. By sequentially executing all *valid* transactions in the hash-chain log starting from the genesis block, we reach the current state of the application stored in the database. FABRIC uses the database as an optimization step since sequentially executing the transactions in the has-chain log each time to reach the application's current state is prohibitively inefficient and computationally expensive.

The orderers receive transactions from clients in the network and serialize and batch the transactions to global order. Furthermore, the batches of the ordered transactions into blocks are sent to peers to be processed and committed to the ledger. A simple ordering service may consist of one orderer component. However, an ordering service consisting of only one orderer is a single point of failure. In order to offer fault tolerance, FABRIC may employ a network of several orderers, which use a coordination-based protocol such as *Paxos* [66] or *Raft* [33] to reach a consensus on the global order of transactions. Although these protocols may offer *Crash fault tolerance*, FABRIC's ordering service is not BFT.

2.1.2 FABRIC's Coordination-based Protocol

FABRIC follows a coordination-based three-phase *execute-order-validate (EOV)* transaction lifecycle. A complete workflow for executing and committing a transaction is depicted in Figure 2.1.1 and follows these steps:

Phase 1 / Execute – The client creates a *transaction proposal* TP_i, which contains the name of the chaincode and the chaincode's function, the input parameters, and the endorsement policy of the application. The client submits the proposal to the peers specified by the endorsement policy in parallel (Step 1 in Figure 2.1.1). Each peer executes the chaincode against the local copy of the world state database, signs the execution results, and sends the results back to the client (Step 2). The results are read-write-sets, where read sets contain the keys read during execution with their version numbers, and write sets contain the key-value pairs to be written to the ledger. Peers do not modify

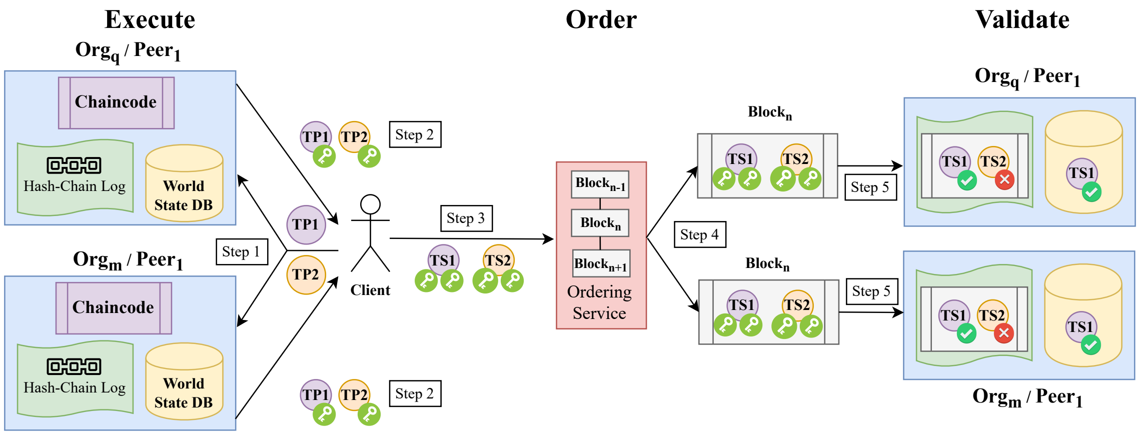


Figure 2.1.1: Transaction lifecycle on FABRIC.

their local copy of the ledger during this phase and only execute the proposal in an isolated manner, also known as peers simulate the proposal.

Phase 2 / Order – Once the client has received an adequate number of endorsements that satisfy the endorsement policy, it creates a transaction TS_i containing the proposal's payload, endorsements, and other metadata. Finally, the client sends the transaction to the ordering service (Step 3). The ordering service receives the transactions from every client in the network, serializes transactions into a total global order, and batches them into new blocks (Step 4). The ordering service is configured to create blocks based on three criteria: (1) a maximum number of transactions in the block, (2) a maximum allowed size, and (3) a timeout period. Once a block is created, the ordering service broadcasts the new block to the peers (Step 5).

Phase 3 / Validate – Peers perform two validations on the transactions of the incoming blocks and then commit the transactions: (1) For the first validation, a peer in parallel verifies whether the transactions are endorsed correctly based on the endorsement policy. This validation ensures the decentralized trust on FABRIC and is also known as *Validation System Chaincode (VSCC) validation.* (2) Second, a peer sequentially compares the version of the keys in the transaction's read-sets with its local copy of the world state database to ensure that the records that were read during the endorsement phase have not changed concurrently in the database. This validation ensures data consistency and is known as *Multiversion Concurrency Control (MVCC) validation.* The transactions that successfully pass both validations are considered valid. Finally, the peer appends every valid and invalid transaction to the hash-chain log and updates the world state database with the write-set of the valid transactions.

2.2 Conflict-free Replicated Data Types

Conflict-free Replicated Data Types (CRDTs) are abstract data types that can be replicated on several replicas with the guarantee to eventually converge to the same state without requiring a coordination-based consensus protocol [12, 67]. In other words, CRDTs converge to the same state in the presence of concurrent transactions in a coordination-free distributed system. CRDTs provide well-defined interfaces, representing various general-purpose data structures such as counters, sets, lists, maps, and JSON objects [54, 67, 68, 69, 70, 71, 72, 73].

CRDTs offer the properties of the general-purpose data structures by extending them with some metadata, making concurrent and conflicting transactions on these data types *commutative* and *convergent*. Commutative and convergent transactions can be applied in different orders on replicas, resulting in the same state independent of the order of the applied transactions, provided no transactions are lost or duplicated. Since concurrent transactions can result in conflicting values, CRDTs use built-in mechanisms to resolve conflicts without coordination. Shapiro et al. [12] formalized CRDTs and proved their *Strong Eventual Consistency (SEC)* property in an eventually consistent distributed system. An SEC system has two requirements:

1. **Eventual Delivery of Transactions:** If a transaction is delivered to one correct replica, then all correct replicas will eventually receive the transaction.
2. **Strong Convergence of Replicas:** If the same set of transactions is applied on every correct replica, then the replica's state immediately converges to the same state.

CRDTs are generally divided into two main categories: *State-based CRDTs* and *Operation-based CRDTs*. State-based CRDTs exchange the whole or delta state of the data type and merge the local state with the received state. For operation-based CRDTs, replicas propagate the state by sending modification operations to other replicas. For example, a counter data type that only increments a value by one can be converted to a grow-only CRDT counter by defining an *increment operation* that increments the value of the counter by one. The grow-only CRDT counter is relatively easy to create since the increment operation is inherently commutative, although not *idempotent*. Therefore, the grow-only CRDT counter converges to the same state independent of the order of applied operations, yet the same operation cannot be applied more than once. For interacting with this counter over the network, replicas send increment operations. Provided that an asynchronous distributed system where the delivery of messages eventually succeeds without message loss or duplication, the counter eventually converges to the same value on all replicas.

However, modification operations 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 [12] that defines the causal order between two events based on *Logical Clocks* [74]. An event E_1 happened before event E_2, if and only if (1) E_1 happened before E_2 in one replica, or (2) E_1 is the event of sending a message M from the sending replica and E_2 is the event of receiving message M on the receiving replica, or (3) there exists an event E such that E_1 happened before E and E happened before E_2. To transform a single-value register to a CRDT, the update operations should include the logical clocks established in which order the operations have happened. Suppose the happened-before relation based on the logical clock cannot be explained. In that case, the register acts as a multi-value register and stores the conflicting values until the final value is decided based on the application's logic.

The theoretical foundation for defining the requirements of several CRDTs has been studied thoroughly [54, 67, 68, 69, 70, 71, 72, 73]. Among existing CRDTs, a JSON CRDT represents a complex general-purpose data structure [54]. A JSON object is a tree structure consisting of other structures like maps and lists. In JSON, a map is a dictionary of key-value pairs where keys are string constants and values are either primitive values like strings or numbers or complex structures like other maps and lists. In this work, we assume that maps are unordered structures and that the values in the maps are either a string, a map, or a list. In JSON, lists are ordered arrays of objects, a combination of primitive values or complex structures, like strings, numbers, maps, or lists.

2.3 Invariant Conditions and Invariant Confluence

Different applications have different correctness requirements. For example, a distributed 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 replicas must preserve during the application's lifecycle. Preserving invariant conditions in a distributed system with globally serialized transactions using techniques such *Total Order Broadcast* [75] to offer *State Machine Replication* [76, 77] is relatively straightforward. Since each transaction preserves the invariant conditions, serialization enables the replicas to apply the transactions sequentially in isolation and preserve the invariant conditions.

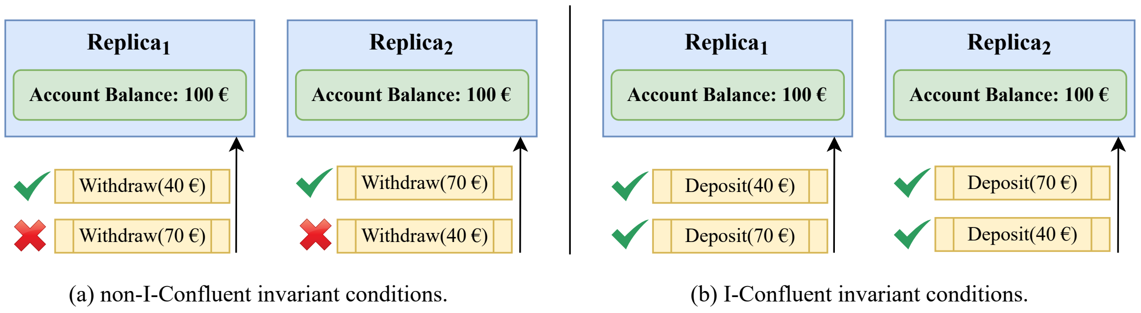


Figure 2.3.1: Non-I-Confluent and I-Confluent invariant conditions.

However, serialization comes at a high coordination cost. In a coordination-free distributed system, the replicas may receive the transactions in different orders. Hence, preserving invariant conditions is challenging. For example, consider a replica that stores the account balance of a customer with an account balance of $\{Balance : 100euros\}$, as shown in Figure 2.3.1(a). The replica can accept only one of the withdrawal transactions of $Withdraw(40euros)$ and $Withdraw(70euros)$. Applying both transactions results in a negative account balance and violates the application's invariant conditions. With coordination, the replicas can agree to accept one of the two transactions.

Bailis et al. [39] studied preserving invariant conditions in a non-Byzantine coordination-free distributed system and introduced the notion of *Invariant Confluence (I-confluence).* A set of transactions $\{TS_1, ..., TS_m\}$ are I-confluent concerning an invariant condition I_j if the transactions can be applied in different orders on different replicas while preserving I_j.

Consider the mentioned withdrawal transactions as an example of a non-I-confluent transaction set. However, two deposit transactions $Deposit(40euros)$ and $Deposit(70euros)$ are I-confluent, as applying these transactions in any order on different replicas does not violate the non-negative invariant condition, as shown in Figure 2.3.1(b).

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 of 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 replicas [39]. Furthermore, as we discussed the commutative and convergent properties of CRDTs in the previous section, CRDTs are a viable method to implement applications with I-confluent invariant conditions.

In this dissertation, for the sake of simplicity, we interchangeably use the phrases, *I-confluent applications*, *I-confluent invariants*, *I-confluent transactions.* All these phrases imply that the set of transactions interacting with the application's state is I-confluent concerning the application's invariant conditions. We also interchangeably refer to *invariant conditions* by using only the term *invariants.*

3 Related Work

The following chapter presents the collective state-of-the-art, discussed in this dissertation. In the upcoming chapters, we first introduce FabricCRDT, a CRDT-based scalability solution of Fabric, to eliminate transaction failures. We explain the existing works on scalability solutions of Fabric in Section 3.1.

We continue by introducing OrderlessChain, a coordination-free BFT permissioned blockchain, which offers a safe environment for the execution of decentralized and distributed applications with I-confluent invariant conditions. OrderlessChain achieves this by using properties of CRDTs and permissioned blockchains. In Section 3.2, we review the related works on CRDTs in Byzantine and non-Byzantine distributed environments. We also discuss the previous works on preserving the invariant conditions of distributed applications in Section 3.3.

To demonstrate the applicability and contributions of OrderlessChain to other domains and industries, we introduce two new systems of OrderlessFL and OrderlessFile. OrderlessFL is built upon OrderlessChain to offer a safe blockchain-based *federated learning (FL)* system using novel CRDTs. We introduce the state-of-the-art of such FL systems in Section 3.4. Finally, OrderlessFile is an extension of OrderlessChain for providing secure and private blockchain-based distributed file storage. In Section 3.5, we review the existing blockchain-based file storage systems.

3.1 Scalability Solutions of Fabric

The high computational overhead and the low throughput of *Proof-of-Work-based (PoW)* consensus protocols of permissionless blockchains, such as Bitcoin and Ethereum [14, 23], are significant bottlenecks hindering the widespread adoption of blockchains in enterprise and industry [6, 26]. In addition to low scalability, the high financial cost of processing transactions on permissionless blockchains and their prohibitively high and environmentally damaging energy consumption make PoW-based BFT protocols infeasible for permissioned blockchains [7, 30, 78].

Permissioned blockchains such as *MultiChain* [79], *R3 Corda* [27], *Quorum* [80], and Fabric [13] make use of various non-PoW-based coordination-based consensus protocols. MultiChain uses a modified version of *Practical BFT* protocol [34]. R3 Corda uses a combination of *Single Notaries*, *Raft* [33] and *BFT-SMaRt* [35]. Quorum, as a permissioned implementation of Ethereum, takes advantage of Raft and *Istanbul BFT* [81]. Fabric currently uses a *Paxos* [66] or Raft-based coordination-based consensus protocol. Furthermore, the Paxos or Raft-based ordering service of Fabric is not BFT. A malicious orderer may jeopardize the system by tampering with transactions or avoiding batching them into the blocks. However, other studies propose BFT ordering services for Fabric, based on BFT-SMaRt [82, 83]. The coordination-based protocols used in permissioned blockchains are more performant and scalable regarding throughput and latency than their permissionless counterparts. However, the required coordination among nodes to reach consensus is yet a scalability bottleneck [5, 36, 84, 85]. Specifically, in the case of Fabric, many transactions fail due to Fabric's optimistic coordination-based protocol [8, 41, 86].

Several works propose various approaches for improving the scalability and throughput of Fabric [36, 44, 45, 87, 88]. Ankur et al. [36] propose transaction reordering techniques inspired by transaction processing of databases [89] to improve Fabric's throughput by early aborting conflicting transactions. The authors decrease the number of conflicting transactions by reordering the transactions in the ordering service according to the transaction's dependency graph. Although the authors demonstrate the practicality of reordering for decreasing transaction failures, they do not aim for the total elimination of failures. Similarly, Pingcheng et al. [44] introduce a solution for improving the throughput of Fabric by creating conflict graphs of transactions and reordering the transactions to decrease the conflicted and failed transactions. Seep et al. [45] propose an approach for ordering and prioritizing transactions based on a weighted fair queueing mechanism to offer clients different and improved *Quality of Service*. However, they do not eliminate transaction failures. Xu et al. [87] also present a transactions reordering method that assigns higher priority to read-only transactions and filters out stale transactions to decrease the transaction abort and failure rates. Sun et al. [90] also offer a transaction reordering approach for decreasing transaction failure rates. To improve Fabric's throughput, Gorenflo et al. [88] introduce *XOX Fabric* that offers an approach for re-executing and committing the conflicted failed transactions. Furthermore, their approach prevents unintentional *Distributed Denial-of-Service (DDoS)* attacks, as the system re-executes the failed transactions without requiring the clients to resubmit them.

Several works focus on determining different bottlenecks of Fabric and offering solutions for mitigating the issues [46, 47, 48, 49]. Zsolt et al. [48] also identify that the coordination-based ordering service of Fabric is a bottleneck. The authors introduce *StreamChain* and improve the performance of the ordering service by replacing Fabric's block processing mechanism with stream processing approaches. They use virtual blocks to decrease the end-to-end latency required for committing transactions and increase the throughput. The authors of *FastFabric* [49] offer extensive analysis and re-architecting guidelines of Fabric to improve several bottlenecks, including the consensus mechanism, I/O, and computational overhead for ordering and validating transactions and repeated validation of certificates for endorsement policies. They implement improvements such as decoupling and parallelizing several internal processes of Fabric and integrate caching to reduce the I/O latency. *xFabLedger* [46] is an extension to Fabric, which stores the ledger data on a remote database to increase the storage scalability of the peers. Thakkar et al. [47] increase the throughput of Fabric by introducing a new component named *Sparse Peer*, which only validates and commits a subset of transactions. They identify that the high latency of validation and commit phases is a bottleneck for Fabric. Therefore, the authors implement a new pipeline to validate and commit transactions in parallel to significantly increase valid transactions' throughput.

The various explained studies provide valuable insights into different approaches to improving the performance of Fabric. These works offer solutions for improving the scalability of Fabric by decreasing the internal latency of Fabric and increasing the throughput of valid transactions. However, they do not eliminate the transaction failure of valid transactions or necessarily improve the performance of the coordination-based protocol of Fabric. In contrast, FabricCRDT offers CRDT-based solutions to eliminate valid transaction failures. OrderlessChain also offers a coordination-free BFT protocol with significantly higher scalability, throughput, and lower latency than the coordination-based protocol of Fabric.

3.2 CRDTs in non-Byzantine and Byzantine Systems

CRDTs have been an impactful technique to improve the scalability and performance of various production-grade non-Byzantine applications, including distributed databases, distributed file systems, and collaborative editing tools [72, 91, 92, 93, 94, 95]. *Riak* [72] is a key-value store that uses several CRDTs, including *flags*, *registers*, *counters*, *sets*, and *maps*, to enable highly available concurrent reads and writes. *Concordant* [96] is an edge-first distributed database that uses delta-based CRDTs to offer *Just-Right Consistency* [94]. *AntidoteDB* [97] is a geo-replicated database that uses several CRDTs, such as the *Last-Writer-Wins Register* [98], *Multi-Value Register*, maps, and sets, to create a highly available transactional database. *Dynomite* [99], a distributed data storage built upon *Redis* [93], and *Memcached* [100] uses CRDTs to create a self-healing system. Similarly, *SoundCloud* [101] implemented *Roshi* [91], which uses *Last-Write-Wins-Element Set* [94] to store large-scale timestamped events. Several applications, such as the note-taking application on *Apple's iOS* [102] and the GPS navigation system from *TomTom* [103] have reported using CRDTs to synchronize the client's data across several mobile devices. Although these applications and systems demonstrate the applicability of CRDTs to several domains and can be implemented on FabricCRDT and OrderlessChain, they do not consider the effect of potential Byzantine participants on the correct execution of their systems. In contrast, FabricCRDT and OrderlessChain provide a safe environment for the execution of CRDT-enabled applications in a Byzantine environment based on permissioned blockchains.

Despite the established benefits of CRDTs for improving the scalability of non-Byzantine distributed applications through decreasing the coordination and resolving conflicting writes automatically, the applicability of CRDTs to Byzantine environments, especially in blockchains, has received less research attention.

A few studies propose approaches for secure CRDT-enabled distributed databases in a Byzantine environment without using conventional BFT coordination-based protocols [40, 104, 105, 106, 107]. Barbosa et al. [104] propose a secure CRDT-enabled storage system based on AntidoteDB using *Homomorphic Encryption* [108]. Kleppmann [105] proposes a generalized solution for offering BFT CRDTs by extending operation-based CRDTs. The author models the operations into a *Directed Acyclic Graph (DAG)* and ensures the eventual delivery of operations in the network. Kleppmann and Howard [40] also present an approach for processing I-confluent transactions on Byzantine distributed databases. The authors introduce a BFT and eventually consistent replicated database and propose an approach for creating a DAG-based dependency graph of transactions. Non-faulty nodes periodically retrieve the missing dependent transactions from other non-faulty nodes. However, their work focuses on peer-to-peer databases and does not offer an environment for the trusted execution of decentralized applications. Similarly, *Matrix* [106], a decentralized publish/subscribe-based middleware, offers a DAG-based CRDT approach to instant messaging in Byzantine environments. Auvolat et al. [107] offer state-based CRDT sets and maps using *Merkle Trees*, which is safe for any number of Byzantine nodes in open networks, where nodes can join and leave freely. However, they only support CRDT sets and maps. In contrast to these coordination-free systems, which offer solutions for specific CRDT applications and systems, FabricCRDT and OrderlessChain offer a general-purpose BFT environment for executing a wide range of Turing complete CRDT applications.

Furthermore, some studies propose coordination-based BFT approaches for executing CRDT applications [50, 51, 109, 110]. Zhao et al. [50] propose a BFT collaborative editing environment based on commutative operations of CRDTs. However, for safe and secure execution, this coordination-based approach requires $3f + 1$ nodes where at most, f nodes can be Byzantine. Shoker et al. [51] propose a BFT and partition-tolerant system for executing CRDT applications that use BFT-SMaRt. Hence, this approach requires $3f + 1$ nodes for safe execution. Cholvi et al. [109] propose a BFT *Distributed Grow-only Set*, which stores a set of immutable records using a collection of BFT atomic operations. However, this work is limited to grow-only sets with restricted applicability and requires $3f + 1$ nodes for safe execution. *SCEW* [110] introduces a BFT client-centric peer-to-peer environment for creating web applications using coordination-based Practical BFT. As these approaches use coordination-based protocols to offer BFT, similar to the coordination-based protocols used in blockchains, the required coordination to reach consensus is a scalability bottleneck. Here, OrderlessChain offers a coordination-free approach to offer trust in a trustless Byzantine environment which can tolerate any number of Byzantine nodes.

Very few works study the applicability of CRDTs to permissionless and permissioned blockchains [52, 111, 112, 113]. *Vegvisir* [111] introduces a DAG-structured blockchain for CRDT-enabled applications. Vegvisir offers a power-efficient blockchain for IoT devices that tolerates network partitioning. However, it does not support executing smart contracts. Fabric developers have introduced a proposal to enhance Fabric's concurrency control by using built-in plugins for parallel execution of basic updates such as incrementing or decrementing counters [114]. However, the implementation of this proposal has not been released, and the available information on the proposal is limited and lacks technical details. *RAMBLE* [112] proposes a blockchain-based BFT censorship-resistant asynchronous distributed messaging protocol with similar functionalities to *Twitter* [115] based on gossip-based epidemic broadcasts and CRDT sets. *MEChain* [113] proposes a blockchain-based secure and private *Electronic Health Record* storage system. The authors use operation-based CRDTs for storing multi-layer structured health-related data. *Setchain* [52] offers a solution for decreasing coordination in BFT blockchains by only partially ordering transactions. Setchain uses epochs as synchronization barriers where transactions that belong to the same epoch are not ordered, while transactions from different epochs are ordered. However, their solution is only limited to grow-only sets and still requires some round of coordination. Contrary to these blockchains, FabricCRDT and OrderlessChain offer techniques not limited to specific CRDTs and provide a safe environment for executing a wide range of CRDT-based use cases. Furthermore, OrderlessChain offers a scalable approach to offering BFT without requiring communication-heavy coordination among nodes to reach a consensus.

3.3 Invariant Conditions of Distributed Applications

Reducing coordination plays a vital role in improving the scalability of any distributed system [39], which has been an active field of research. Over the past few decades, several studies have proposed approaches to decrease the high coordination costs, increasing the throughput and decreasing latency in non-Byzantine distributed environments while preserving the application's invariant conditions [116, 117, 118, 119, 120]. Pedone et al. [116] define the *Generic Broadcast* problem in non-Byzantine *Crash Fault Tolerant* environments, where the messages in the network are only globally ordered provided that the semantics of messages are potentially conflicting. They demonstrate the improved latency over the coordination-based *Atomic Broadcast* protocols [121], where all messages are globally ordered. Lamport [117] improves the efficiency of the coordination-based Paxos protocol and offers *Generalized Paxos* in a non-Byzantine system by adapting the *State Machine Replication* [76, 77] defined in Paxos for partially ordering of non-conflicting messages and enabling their concurrent execution. Lamport [118] enhances the proposed approach further to the lower bounds of coordination for partially ordering messages. Li et al. [119] propose preserving invariant conditions by offering global coordination only when the application requires strong consistency. Otherwise, their proposed solution uses a faster, eventually consistent method with less coordination but offers weaker consistency. Lloyd et al. [120] propose *COPS*, a geo-replicated key-value storage. This system replicates key-value pairs across nodes in a coordination-free manner while preserving the causal relations of pairs.

In contrast to the studies for reducing coordination in non-Byzantine systems, some studies offer solutions to preserve invariant conditions without coordination in non-Byzantine environments [122, 123, 124, 125]. O'Neil [122] introduces a transactional escrow method for dividing the available resources among nodes, where the resources can be consumed independently on the node without coordinating with other nodes. Balegas et al. [123] use the escrow methods to define *Bounded CRDT Counter*, which preserves the invariant condition of not turning negative. They perform this by specifying a fixed number of decrement operations each node can perform so the total number of decrement operations does not exceed the counter's value. Once the limit of decrement operations is reached on one node, the node coordinates with other nodes to query and retrieve other nodes' non-consumed decrement operations. Balegas et al. [124] extend their method further for preserving invariant conditions on other data types. Furthermore, Liu et al. [125] introduce time-limited warranties, during which distributed objects preserve and lock specific invariant conditions. Therefore, the clients acquiring such warranties do not need to communicate with nodes to validate and preserve the invariant conditions.

For minimizing and eliminating coordination among nodes, Bailis et al. [39] propose I-confluence, a formal framework for analyzing the application-level invariant conditions over the node's state to determine whether coordination among nodes is necessary. Since identifying I-confluent invariant conditions can be cumbersome for application developers, Whittaker et al. [126] propose *Lucy*, which automatically checks whether the invariant conditions of an application are I-confluent using an interactive I-confluent decision-making procedure. I-confluence shares similarities with *Left Commuting Operations* introduced by Friedmann and Birman [127]. However, the I-confluence identifies the invariants that can be preserved without coordination. The left commuting method identifies the possible order of operations to preserve the application's serializability definition. Furthermore, I-confluent shares similarities with the *Consistency And Logical Monotonicity (CALM)* theorem [128, 129, 130]. CALM theorem proves that monotonic transactions can be processed in a coordination-free manner.

These works motivate the necessity of reducing coordination to increase scalability and offer valuable and practical solutions for preserving invariant conditions without or with reduced coordination. However, they are designed for non-Byzantine environments, and the increased difficulty of preserving the invariants in a Byzantine environment is not considered, where the malicious participants can intentionally violate the invariant conditions. OrderlessChain offers coordination-free solutions to preserving invariant conditions in Byzantine environments.

In order to address the Byzantine participants, Pires et al. [131] offer a BFT design of Generalized Paxos for partially ordering the transactions in a trustless environment. As a BFT model of Generic Broadcast, Raykov et al. [132] propose a partially ordered environment where only the conflicting transactions are ordered. The authors use the *Recovery Consensus* mechanism to ensure that the same set of non-conflicting messages is concurrently executed and conflicting messages are ordered using coordination-based protocols. Martin et al. [53] propose *FaB Paxos*, a BFT Paxos approach. Although FaB

Paxos offers a low bound of communicated messages required to offer BFT, it still uses a coordination-based approach. Guerraoui et al. [133] offers a theoretical proof that the conventional PoW-based protocol, as used in Bitcoin, is unnecessary to execute payment systems safely and to prevent double-spending attacks [134]. They use a *Reliable Broadcast*-based [135] approach to offer BFT, which is coordination-based. Also, their work is limited to theoretical proof and offers no implementation or quantitative evaluation. Similarly, Collins et al. [136] propose *Astro* based on the reliable broadcast for transferring funds. However, their work is only limited to processing payments.

Although these studies offer BFT while reducing coordination, they do not eliminate the coordination requirement and are restricted to limited use cases. OrderlessChain uses the permissioned property of permissioned blockchains to offer scalable and safe coordination-free approaches for executing several types of applications in Byzantine distributed environments.

3.4 Asynchronous Federated Learning in Byzantine Environments

Machine Learning (ML)-related research has been one of the fastest-growing areas in academia and industry over the past decade [137, 138]. However, as ML's popularity has increased, the concern over ML's potential privacy risks has also increased [139, 140]. A standard conventional ML pipeline requires a tremendous amount of raw data to be processed and trained by a central organization, which introduces serious privacy risks for the owners of the shared raw data [138, 140]. Various privacy-preserving distributed ML approaches have been proposed to address these issues [137, 139, 140]. One of the most prominent solutions to private ML is *federated learning (FL)* [56, 137]. FL offers a privacy-preserving solution by providing a collective approach to train models where the clients train the models locally without sharing the raw data with different organizations [56, 141].

Although a conventional FL system, as initially introduced by McMahan et al. [56], offers more privacy than existing distributed ML systems, their proposed FL solution does not consider the Byzantine behavior of participants. Several works study the potential threats of malicious participants to an FL system, including the *Data Poisoning*, *Inference Attacks*, *Membership Inference Attacks* [142, 143], which result in a significantly reduced privacy.

Several works propose permissionless and permissioned blockchain-based FL systems to offer a trusted and secure training environment in the presence of Byzantine participants [144, 145, 146, 147, 148, 149, 150, 151]. *BlockFlow* [144] proposes an Ethereum-based FL system to hold Byzantine workers accountable who jeopardize the training process. They also employ *Differential Privacy* [152] to offer more privacy. However, the model updates and FL aggregation are performed off-chain due to Ethereum's processing and storage limitations. *BlockFLA* [149] also propose an Ethereum and FABRIC-based FL solution for detecting anomalies caused by malicious behavior and holding the Byzantine participants accountable. *BAFFLE* [145] uses a private Ethereum to offer an FL system without central aggregation, implementing several smart contracts to perform model aggregation locally. Due to the storage limitations of used blockchains, they partition models stored on the system. Zhao et al. [146] propose an *Algorand*-based BFT FL system for IoT devices. Due to the blockchain limitations, similar to BAFFLE, the authors also use an off-chain solution based on *InterPlanetary File System (IPFS)* [153] to store the model updates. Wu et al. [147] introduce *FedBC*, which offer's a FABRIC-based FL system. However, they also make use of third-party IPFS for storage. *GFL* [148] also takes advantage of Ethereum and IPFS to offer a blockchain-based FL system.

These works provide valuable solutions to the BFT FL systems and demonstrate the potential and limitations of blockchains for FL systems. However, these systems rely on coordination-based protocols and, in the worst-case scenarios, use PoW-based blockchains. Hence, the scalability of these systems is limited due to the inherent scalability issues of coordination-based protocols. Furthermore, due to the storage and computational processing limitations on various blockchains, the explained systems often use third-party solutions for storing the model updates and aggregating the updates with the global models, which may be potentially Byzantine. In contrast, in ORDERLESSFL's proposed approach, we offer a BFT solution using a coordination-free approach without the limitations of coordination-based protocols. Also, we offer an on-chain storage solution where Byzantine participants cannot tamper with model updates.

Besides the potential threats of Byzantine participants in FL systems, one other problem in asynchronous FL systems is the concurrent aggregations of the ML models and the *gradient staleness* problem [154, 155]. We introduce FLCRDT with ORDERLESSFL, demonstrating the applicability and contributions of CRDTs in FL systems for concurrent and asynchronous aggregation of FL models and mitigating the gradient staleness problem. However, to the best of our knowledge, no work exists that studies the CRDTs in the FL environment. We use the logical clocks included in the CRDT operations to mitigate the gradient staleness problem. A few studies propose timestamp-based solutions to constrain the gradient staleness [155, 156, 157, 158, 159]. Ho et al. [156] propose a network of distributed *parameter servers* where faster workers aggregate their models more frequently. They enforce a timestamped maximum staleness limit for slower workers. Jiang et al. [157] extend Ho et al.'s approach and propose heterogeneity-aware distributed parameter servers where the learning rate of ML training depends on the model updates' staleness. Li et al. [158] offer an explicit vector clock-based approach for enabling parallel training by distributed parameter servers. Zhang et al. [155] also define a staleness penalty based on the logical clock of model updates for *Asynchronous Stochastic Gradient Descent* in distributed deep learning systems for bounding the gradient staleness. Similarly, Xie et al. [159] propose a logical clock-based approach for mitigating and penalizing stale updates. FLCRDT's approach for mitigating gradient staleness is inspired based on these explained works.

3.5 Blockchain-based File Storage Systems

Cloud storage and *Storage-as-a-Service* have been expanding rapidly in the cloud providers industry [160]. Despite the high availability and low cost of cloud storage, clients must trust the cloud providers to safely and securely store their data [161, 162]. Many blockchain-based file storage systems have been proposed to offer decentralized trust, provide a secure alternative to cloud-based storage solutions, and enable providers to rent out their excess storage [163, 164, 165, 166]. *Stroj* [163] is a blockchain-based cloud storage where providers can rent their excess hardware and bandwidth to the clients. Furthermore, Storj uses a sharding mechanism to split files and store and replicate the shards on the providers. Storj uses a combination of Ethereum-based systems to store the file's metadata and a *Proof of Space* protocol to offer decentralized trusted storage. *Sia* [164] also offers a PoW-based solution for providers to rent out their excess hardware. Sia splits the files into shards and encrypts and stores the shards on different providers. The providers are compensated for the provided resources using Sia's cryptocurrency called *SiaCoin*. *FileCoin* [165] provides blockchain-based off-chain storage using IPFS, where the providers are compensated using FileCoin's native cryptocurrency. FileCoin uses *Proof-of-Spacetime* and *Proof-of-Replication* to ensure clients that the system safely and securely stores and replicates the files on various providers in a decentralized manner for a specific time. However, Guidi et al. [167] demonstrate the lack of decentralization of FileCoin in contrast to its developers' claims. *BlockStore* [166] offers blockchain-based off-chain file storage with an Ethereum-based payment system to enable providers to monetize their excess storage.

A few studies proposed CRDT-based decentralized file storage in non-Byzantine distributed environments [168, 169]. However, the applicability of CRDT-enabled blockchains for BFT file storage has received no research or industry attention. As explained, existing systems rely on coordination-based or PoW-based protocols, which limits the scalability of these systems. Furthermore, they use off-chain third-party solutions due to the storage limitations on blockchains, which may introduce new security threats. In contrast, OrderlessFile, built upon the BFT coordination-free protocol of OrderlessChain, securely and safely replicates files on-chain using a novel CRDT for splitting files into shards.

4

FabricCRDT: A CRDT-enabled Permissioned Blockchain

The significant scalability limitations of PoW-based protocols used in permissionless blockchains and their prohibitively high costs make them infeasible for several real-world use cases in business and industry [4, 11, 26]. For decentralized enterprise use cases where the identity of participants is known, permissioned blockchains constitute a viable alternative. One of the most prominent permissioned blockchains is Fabric, which offers significantly higher throughput and transactional guarantees than Bitcoin and Ethereum while allowing the deployment of Turing complete applications [86].

Although Fabric is significantly more scalable than its permissionless counterparts, Fabric follows an optimistic three-phase protocol to ensure data consistency, which causes the failure of a significant number of concurrent and conflicting transactions in real-world use cases [8, 41]. In this chapter, we introduce FabricCRDT, an extension of Fabric. FabricCRDT takes advantage of CRDTs, to address the failures of concurrent transactions. Our approach uses CRDTs to merge and automatically resolve the conflicts of concurrent transactions instead of causing the failure of transactions. Hence, by eliminating transaction failures, FabricCRDT significantly improves the throughput and scalability of Fabric.

The content of this chapter is based on the paper published on FabricCRDT [55].

The remainder of the chapter is organized as follows. First, we provide a detailed explanation of *Multiversion Concurrency Control* mechanism and the causes for the failure of concurrent transactions in Section 4.1. In Section 4.2, we describe the architecture, design requirements, and the transaction lifecycle of FabricCRDT. In Section 4.3, we introduce our approach for integrating and implementing CRDTs on FabricCRDT. We explain our approach's potential and limitations in comparison to Fabric in Section 4.4. Finally, we evaluate FabricCRDT and demonstrate its improved performance over Fabric in Section 4.5.

4.1 Multiversion Concurrency Control-based Failures

In order to establish the problem better, which FabricCRDT addresses, we discuss Fabric's optimistic concurrency control mechanism and the causes of transaction failures in more detail.

A transaction proposal invokes the chaincode on Fabric, which during the chaincode execution, based on the application's logic, interacts with the stored application's state on the ledger in three ways:

- **Read-Transaction:** Execution of chaincode only results in reading key-value pairs from the ledger.
- **Write-Transaction:** Chaincode only writes key-value pairs to the ledger without reading any pairs during its execution.
- **Read-Write-Transaction:** Chaincode reads key-value pairs from the ledger and writes key-value pairs to the ledger.

The execution of a transaction TS_i results in a *read-set* and a *write-set* included in a *read-write-set* as follows:

$$ReadWriteSet_{TSi} =< ReadSet_{TSi}, WriteSet_{TSi} >$$

The read-set includes a set of keys and the version number of the key's value that a peer retrieved from the ledger during the execution of the chaincode, as follows:

$$ReadSet_{TSi} = \{(K_1^R, VN_1^R), ..., (K_n^R, VN_n^R)\}$$

A read-transaction creates a $ReadWriteSet_{TS_i}$ only with $ReadSet_{TS_i}$ and an empty write-set. Furthermore, read-transactions do not change the ledger's state, and clients may not send the transactions to be ordered and committed.

The write-set contains the key-value pairs created during the execution of the chaincode. The write-set of a transaction TS_i is modeled as follows:

$$WriteSet_{TSi} = \{(K_1^W, V_1^W), ..., (K_m^W, V_m^W)\}$$

The read-write-set of write-transactions only contains a write-set. A read-write-transaction contains both sets in its read-write-set. To commit a successfully validated transaction, the write-set of the transaction is applied to the ledger. The ledger contains the application's world state in the format of key, value, and value's version number tuples, as follows:

$$WorldState = \{(K_1^{WS}, V_1^{WS}, VN_1^{WS}), ..., (K_q^{WS}, V_q^{WS}, VN_q^{WS})\}$$

Applying the write-set to the ledger may cause a conflict and subsequent failure of the transaction known as the *Multiversion Concurrency Control (MVCC) Failure.* Formally speaking, an MVCC conflict occurs if there exists a key $K_l^R \in \{K_1^R, ..., K_n^R\}$ with version number VN_l^R in the TS_i's read-set, and there exist $K_l^{WS} \in \{K_1^{WS}, ..., K_q^{WS}\}$ in the world state, where $K_l^R = K_l^{WS}$ and $VN_l^R \neq VN_l^{WS}$, the version number of K_l in the read-set and world state are unequal [8].

Intuitively speaking, and to illustrate the problem better, imagine that at time T, peer P_1 has the world state:

$$WorldState = \{(K_1^{WS}, V_1^{WS}, VN_1^{WS}), (K_2^{WS}, V_2^{WS}, VN_2^{WS}), (K_3^{WS}, V_3^{WS}, VN_3^{WS})\}$$

P_1 receives a block containing five transactions with corresponding read-write-sets, and the transactions are ordered in the block as follows:

1. $ReadWriteSet_{TS1} =< ReadSet_{TS1} : \{(K_2^R, VN_2^R)\}, WriteSet_{TS1} : \{(K_2^W, V_2^W)\} >$
2. $ReadWriteSet_{TS2} =< ReadSet_{TS2} : \{(K_1^R, VN_1^R), (K_2^R, VN_2^R)\}, WriteSet_{TS2} : \{(K_3^W, V_3^W)\} >$
3. $ReadWriteSet_{TS3} =< ReadSet_{TS3} : \{(K_2^R, VN_2^R)\}, WriteSet_{TS3} : \{(K_3^W, V_3^W)\} >$
4. $ReadWriteSet_{TS4} =< ReadSet_{TS4} : \{(K_3^R, VN_3^R)\}, WriteSet_{TS4} : \{(K_2^W, V_2^W)\} >$
5. $ReadWriteSet_{TS5} =< ReadSet_{TS5} : \{\}, WriteSet_{TS5} : \{(K_3^W, V_3^W)\} >$

Given that all five transactions pass the endorsement policy validation as explained in the background chapter (not explicitly shown here), P_1 sequentially validates the five transactions in the block by comparing the version number of each key in the read-set to the version number in the world state. A transaction is considered valid if both version numbers are equal. If the version numbers are unequal, the peer invalidates the transaction as an MVCC conflict. The key's mismatch results from updates committed by preceding valid transactions. The initial transactions may be included either in the previous blocks or in the same block but preceding the current position of the conflicting transaction. Committing keys in the write-set of the valid transactions causes the version number of keys in the world state database to change. Therefore, P_1 marks TS_1 as valid and TS_2 and TS_3 as invalid because the write-set of TS_1 updates K_2 so that its new version number is $VN_2^{WS} + 1$ and the version number of K_2 in TS_2 and TS_3's read-set does not match ($VN_2^R \neq VN_2^{WS} + 1$). P_1 marks TS_4 as valid, since TS_3 is invalid and version number of K_3 is not updated. Finally, TS_5 is valid due to the empty read-set and independence from the version number of keys in the world-state.

This multiversion concurrency control mechanism is a commonly used optimistic concurrency control model in database systems to increase the throughput and decrease the latency instead of blocking mechanisms such as shared locks [8, 41]. Although this mechanism is necessary for ensuring data consistency and isolation of transactions required for preserving the application's invariant conditions, the relatively high latency between the creation of the read-write-set and the validation of the read-set in Fabric can result in a large number of transactions in a block failing, especially when a small set of frequently accessed keys are included [8, 36, 88]. This high latency consists of the *endorsement latency*, the *ordering latency*, and the *commit latency* [86], as defined below:

1. **Endorsement Latency:** It is the time needed for the client to acquire all the required endorsements, which, depending on the endorsement policy and the complexity of chaincodes, varies significantly for different transactions.
2. **Ordering Latency:** It is the time required for the transaction to be batched in one block and broadcast to the peers. The ordering service batches transactions into a block based on several criteria, including the *maximum number of transactions*, the *maximum total size of transactions in a block*, and a *timeout period* for creating blocks. The ordering service creates a block for higher transaction arrival rates as soon as the maximum size is reached. However, the transaction can be delayed for lower arrival rates until the timeout period is reached. The timeout period is a configurable parameter on the order of seconds.
3. **Commit Latency:** That is the time a peer takes to validate transactions in the block and commit them to the ledger.

These delays are inherent to the design of Fabric and can not be significantly reduced without fundamental changes to the system. The delay may be a few seconds depending on the system configuration of Fabric. Although the MVCC failures cause minor issues for processing read-transactions, realistic use cases consist of a large number of read-write-transactions, where the version number of keys in their read-set depends on the keys in the write-set of preceding transactions [8, 41]. This dependency imposes MVCC conflicts on many transactions containing such hotkeys. According to the *Hotkey Theorem* [88], if the average delay for a transaction containing a hotkey is l, then the maximum throughput for all transactions containing the hotkeys is $1/l$.

Once a transaction fails, the only option for clients is to create a new transaction and resubmit, which adds to the complexity of Fabric application development. Therefore, providing a solution that enables Fabric to manage the conflicting transactions internally without rejecting the transactions can significantly improve the scalability and throughput of Fabric and simplify the application development process.

Without inherently changing the design of Fabric, we proposed using CRDTs for automatically resolving the conflicts and merging the key's values without causing the failures of conflicting keys.

4.2 Architecture and Design

In the following section, we explain the system model. Followed by the design requirements of FabricCRDT that our design must satisfy. We also describe our approach for executing CRDT transactions on FabricCRDT.

4.2.1 System Model

FabricCRDT, as an extension of Fabric, shares the exact system model and failure model with Fabric. FabricCRDT is a strongly eventually consistent permissioned blockchain consisting of a set of peers $\{P_1, ..., P_n\}$ and a set of clients $\{C_1, ..., C_r\}$.

Every peer and client has a unique identifier. The identity of each peer is known to every other peer and client in the network. Each peer belongs to only one organization, representing an entity, which may range from individuals to businesses. The organizations define the trust boundaries in the system. In other words, peers from different organizations do not necessarily trust each other, despite their known identities. However, the peers within one organization trust each other.

Peers and clients can communicate with other non-failed peers by sending and receiving messages. We consider every message and transaction to be delivered eventually, despite arbitrary delays. However, the order of the transactions in a block is not guaranteed to be the same order of transactions when issued by clients or arrived at the ordering service. The ordering service does not guarantee to prevent duplicate transactions. This work assumes that clients do not intentionally submit duplicate transactions. If duplicate transactions are submitted, FabricCRDT also processes duplicate transactions.

4.2.2 Design Requirements

We define four design requirements that FabricCRDT must satisfy:

1. **Backward Compatibility** – We aim to extend Fabric with CRDT-enabled functionalities with minimal changes to the original design of Fabric. This way, we keep the learning curve minimal for developers who have already designed applications for Fabric. Also, the applications developed for Fabric remain compatible with FabricCRDT.

2. **No Transaction Failure** – FabricCRDT should be able to commit all valid CRDT-based transactions successfully. We define *valid transactions* as the transactions submitted by the client which pass the endorsement policy validation successfully.

3. **No Update Loss** – By committing all valid transactions in a block, FabricCRDT eventually converges to the same state on all peers, and all client's updates are preserved while using CRDT techniques to merge conflicting transactions.

4. **Use Case Generality** – To accommodate developers with the possibility of realizing a wide range of use cases, the CRDT approach used for merging conflicting transactions in FabricCRDT should provide developers with a general-purpose data structure to submit data to the ledger.

4.2.3 Transaction Lifecycle

To achieve the discussed design requirements, we deploy a CRDT-based approach for dealing with conflicting transactions internally. As explained, Fabric rejects transactions with an outdated version number of key-value pairs in the read-set and discards these transactions' write-set. Committing key-value pairs of write-set with an outdated version may result in data inconsistencies. To avoid the failure of conflicting transactions and data inconsistencies, FabricCRDT does not reject transactions but instead merges the values of the conflicting transactions using CRDT techniques.

Since we aim to keep Fabric applications compatible with FabricCRDT and to fulfill the *Backward Compatibility* requirement, we define a new type of transaction that encapsulates all CRDT-related functionalities. Figure 4.2.1 displays the transaction lifecycle in FabricCRDT, where CRDT and non-CRDT transactions coexist.

The non-CRDT transactions are standard transactions of Fabric that do not modify CRDT values on the ledger. These transactions follow the three-phase transaction lifecycle of Fabric, as explained in the background chapter, including being ordered into a block, validated for endorsement VSCC, and MVCC validities.

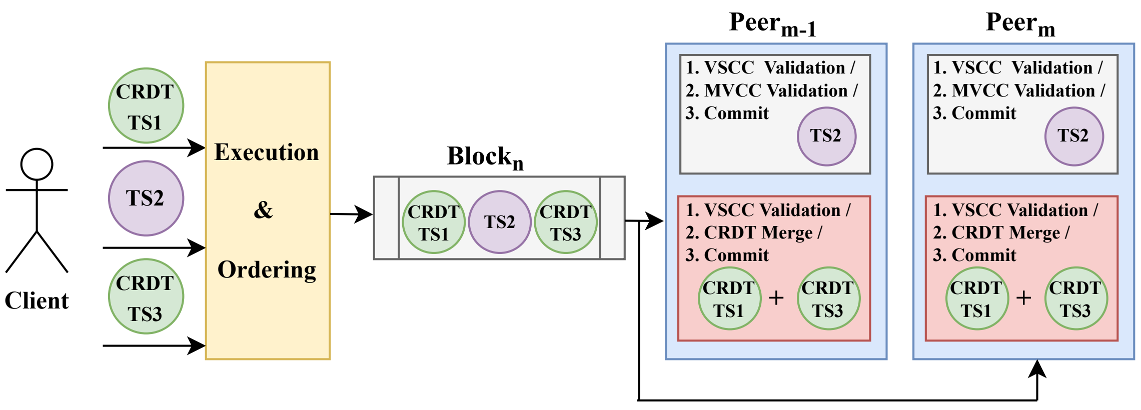


Figure 4.2.1: Transaction lifecycle on FabricCRDT.

Although the CRDT-enabled transactions have a structure similar to standard Fabric transactions, their invocation of chaincode modifies CRDT-encapsulated values on the ledger. The CRDT transactions follow the first two phases of the transaction lifecycle of

Fabric, as demonstrated in Figure 4.2.1. As these phases are explained in the background, we forgo repeating the procedure description.

During the last phase, the CRDT transactions on FabricCRDT are only verified for endorsement VSCC validation. Then, provided the successful endorsement validation, instead of the MVCC validation, the transaction values of conflicting transactions are merged automatically using CRDT techniques before being committed to the ledger. The CRDT procedures used for conflict resolution depend on the type of CRDT object. For example, managing grow-only CRDT counters requires different techniques than merging JSON CRDTs. In our prototype of FabricCRDT, we support merging JSON CRDTs, as explained in the following sections.

4.3 Implementation

In the following section, we explain the implementation of FabricCRDT in detail. We introduce our approach to integrating CRDTs into the system. We also explain the mechanism for merging JSON objects using CRDT techniques. We implemented FabricCRDT based on Fabric v1.4.0.

4.3.1 Merging CRDT Transactions

The CRDT transactions in a block bypass the MVCC validation and are merged before getting committed. Algorithm 1 explains our approach for committing transactions in a block on FabricCRDT.

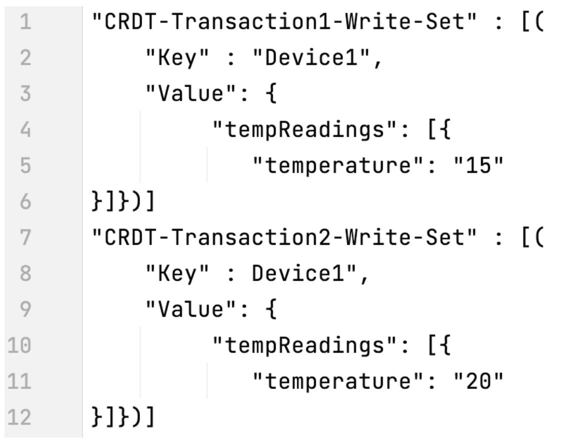
For resolving the CRDT transactions in a block, first, we iterate through all transactions in the block, and for each transaction, we iterate through the key-value pairs in the transaction's write-set (Lines 3 to 14 in Algorithm 1). If the key-value pair is not marked as a CRDT, we skip the key-value pair to be handled as a non-CRDT transaction. However, if the key-value pair is flagged as a CRDT, the algorithm first checks if a CRDT object with the same key already exists in a local set containing all CRDT objects (Line 7). If a

Algorithm 1: Merging CRDT transactions in a block on FabricCRDT.

```
1  ValidateMergeCRDTTransactions ($Block$)
       input   : $Block$, a block received from the orderer.
       output  : $MergedCRDTSBlock$, a block with merged CRDT transactions to be
                 committed to the ledger.
2      $CRDTs = newSet()$
3      foreach $TS_i$ in $Block.Transactions$ do
4          foreach $key_j, value_j$ in $TS_i.WriteSet$ do
5              if $value_j.IsCRDTObject()$ then
6                  $value_j.SkipMVCCValidation()$
7                  $CRDT = CRDTs.GetObjectIfExists(key_j)$
8                  if $CRDT == Null$ then
9                      $CRDT = InitEmptyCRDT(key_j, value_j)$
10                     $CRDTs.SetObject(CRDT)$
11                 $MergeCRDT(CRDT, value_j)$
12                 $CRDTs.SetObject(CRDT)$
13             else
14                 // Skip it and let it be handled as non-CRDT transactions.
15     $DoMVCCValidationOnNonCRDTTransactions(Block)$
16     foreach $TS_i$ in $Block.Transactions$ do
17         foreach $key_j, value_j$ in $TS_i.WriteSet$ do
18             if $value_j.IsCRDTObject()$ then
19                 $CRDT = CRDTs.GetObjectIfExists(key_j)$
20                 $DataTypeObject = CRDT.ConvertCRDTToDataType()$
21                 $value_j = DataTypeObject.ConvertToBinary()$
22                 $TS_i.UpdateWriteSet(key_j, value_j)$
23     return $Block$
```

CRDT object does not exist, the algorithm instantiates a new CRDT object with the key and adds it to the set (Lines 9 and 10). The type of CRDT object depends on the type of CRDT value in the key-value pair. For example, for a JSON CRDT, an empty JSON CRDT object is instantiated. Afterward, the peer converts the binary value of the key-value pair to the corresponding type and merges it with the CRDT object. Then, the set containing all CRDT objects is updated (Lines 10 to 12). We discuss the steps required for merging the individual CRDTs in the following subsection.

After the first iteration, the peer performs MVCC validation on non-CRDT transactions (Line 15). Afterward, the algorithm iterates through every transaction's write-set once more to check if a CRDT object exists for that key in the local CRDT set (Lines 16 to 22). If a CRDT object exists (Line 19), then the CRDT object is converted to the corresponding data type. For example, a JSON CRDT is converted to a JSON object (Line 20). The converted object represents the data type with all the CRDT-related metadata cleaned up and removed. Finally, the object is converted into a byte array that replaces the key-value pair's value in the transaction's write-set (Lines 21 and 22). The second iteration through every transaction's write-set is necessary because the peer is unaware of all key-value pairs of CRDT transactions in the block that need to be merged until the end of the first iteration. Once all CRDT transactions are merged, the peer finalizes and cleans up the metadata of CRDT objects and updates the write values of the corresponding transactions with the new converged value (Lines 20 to 22), which is then committed to the ledger by the peer.

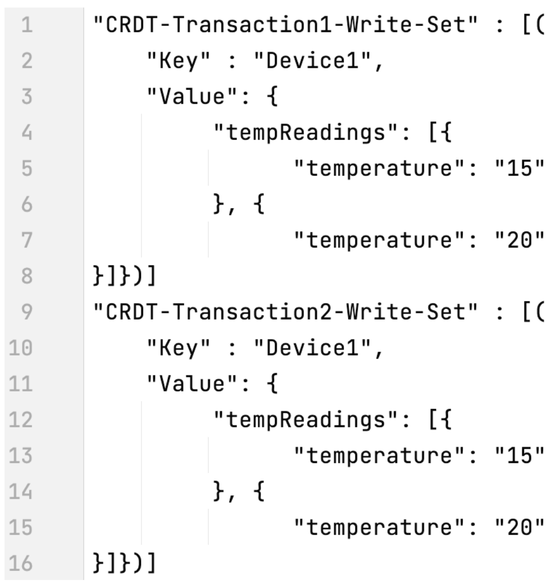


```
"CRDT-Transaction1-Write-Set" : [(
    "Key" : "Device1",
    "Value": {
         "tempReadings": [{
            "temperature": "15"
}]})]
"CRDT-Transaction2-Write-Set" : [(
    "Key" : Device1",
    "Value": {
         "tempReadings": [{
            "temperature": "20"
}]})]
```

Figure 4.3.1: JSON objects in CRDT transactions' write-set.

For example, consider two JSON objects in the write-sets of two different transactions with the same key, as depicted in Figure 4.3.1. Since the values have JSON types, Algorithm 1 creates one JSON CRDT with the identifier *Device1* and extends and merges the created JSON CRDT with both values.

The result of merging the two CRDT values is shown in Figure 4.3.2. The write-set of *Transaction 2* is identical to the write-set of *Transaction 1*, which is committed to the ledger.



```
"CRDT-Transaction1-Write-Set" : [(
    "Key" : "Device1",
    "Value": {
         "tempReadings": [{
               "temperature": "15"
         }, {
               "temperature": "20"
}]})]
"CRDT-Transaction2-Write-Set" : [(
    "Key" : "Device1",
    "Value": {
         "tempReadings": [{
               "temperature": "15"
         }, {
               "temperature": "20"
}]})]
```

Figure 4.3.2: Merged JSON objects in CRDT transactions' write-set.

4.3.2 Enabling JSON CRDT on FabricCRDT

Although the approach discussed in Algorithm 1 is independent of the CRDTs, plenty of CRDTs exist with different specifications and requirements for resolving conflicts. Hence, every CRDT requires specific implementation. In our prototype of FabricCRDT, we focused on implementing and integrating *JSON CRDT* [54], which provides a general-purpose data structure for complex use cases and enables us to fulfill the *Use Case Generality* design requirement.

We implemented JSON CRDT based on the theoretical work of Kleppmann et al. [54] and a Go language-based JSON CRDT implementation [170]. The authors of JSON CRDT introduce the formal semantics and an approach for implementing an API for interacting with a JSON CRDT. The approach provides an API for modifying JSON CRDT objects, such as inserting, assigning, deleting, and reading values from the object. The reading API does not cause any modification to the object. Modifying the object is performed using *CRDT operations*, which have globally unique identifiers.

Although necessary for ensuring the automatic conflict resolution among several processes, the modification API described by the authors is cumbersome for chaincode developers due to its complexity. In FabricCRDT, every peer observes the transactions in a block in the same order. We exploit this property to simplify the API. To use the JSON CRDTs in the chaincode, similar to chaincodes on Fabric, developers should create JSON objects. However, for submitting the key-value pairs to the ledger, the developer should use the CRDT-specific *putCRDT* method we implemented in the chaincode shim. This method only informs the peer that this value is a CRDT and does not perform direct modification on the CRDT. The operations required for merging the JSON CRDTs are performed on the peers without the directions of the chaincode developer.

Algorithm 2 describes our approach for merging JSON CRDTs. This algorithm is the *MergeCRDT* function in line 11 of Algorithm 1. Algorithm 2 is adapted based on an existing implementation of JSON CRDT [170]. It iterates through each key-value pair in the JSON object, where the value is either a string, a list, or a map (Line 2 in Algorithm 2). The items included in the list or map may include nested maps or lists. For each value in the JSON object, first, we create an empty elements list and an empty dependencies list (Lines 3 and 4). The elements list defines the path from the head of the JSON CRDT to the location where the modification of the JSON CRDT is applied. A modification to the JSON object is interaction, such as adding or deleting. The dependencies list contains the unique identifier of all operations which should be performed before the current operation is executed. We ensure that the operations identifiers are globally unique by using an instance of a *Lamport clock* [74] for each JSON CRDT instantiation. The Lamport clock is incremented by one with every newly applied operation to ensure the causal order of the operations is preserved.

If the value of a key in the JSON object is a string, the algorithm executes lines 6 to 9. First, it extends the elements with the current key and increments the Lamport clock by one. Then, the modification is applied to the JSON CRDT, based on the dependencies list, and the location in the JSON CRDT where the modification occurs (Line 8).

For applying the operation, first, we check if all dependencies in the operation's dependencies list are already applied. If some of the operations are missing, we queue the operation until all dependencies are applied. If there is no pending operation, we apply the operation by using the operation's elements to span from the head of the JSON CRDT. For every element in the elements, if the element already exists, we add the identifier of

Algorithm 2: Merging a JSON object with the JSON CRDT on FabricCRDT.

```
1  MergeCRDT ($JSONCRDT, JSONObj$)
       input  : $JSONCRDT$, an initialized JSON CRDT object.
       input  : $JSONObj$, a JSON object to be added to the JSON CRDT object.
2      foreach $key_i$, $value_i$ in $JSONObj$ do
3          $elements = newElementsList()$
4          $dependencies = newDependenciesList()$
5          if $value_i.IsString()$ then
6              $CreateElement(elements, key_i)$
7              $JSONCRDT.IncrementClock()$
8              $ApplyOperation(JSONCRDT, dependencies, elements, key_i, value_i)$
9              $dependencies.Add(JSONCRDT.OperationID())$
10         else if $value_i.IsList()$ then
11             foreach $listValue_j$ in $value_i.GetListItems()$ do
12                 $CreateElement(elements, key_i)$
13                 $RecursivelyAddListItemToJSONCRDT(JSONCRDT,$
                     $key_i, listValue_j, dependencies, elements)$
14                 $RemoveElement(elements, key_i)$
15         else if $value_i.IsMap()$ then
16             foreach $mapKey_j$, $mapValue_j$ in $value_i.GetMapItems()$ do
17                 $CreateElement(elements, key_i)$
18                 $RecursivelyAddMapItemToJSONCRDT(JSONCRDT,$
                     $mapKey_j, mapValue_j, dependencies, elements)$
19                 $RemoveElement(elements, key_i)$
```

the current operation to the element to track the current operation's dependencies. If the element is missing in the JSON CRDT, we add the element to the JSON CRDT and the operation's identifier to the element's dependencies. Once we reach the end of the path and the modification location is reached, we apply the modification to the JSON CRDT. For adding the element, we insert a key-value pair with the key as the operation identifier and the value as the string value from the JSON object.

When the value of the JSON object is a list, we iterate through the list's items (Lines 12 to 14). For every list item, first, we append the element with the current key in the JSON object, then call a recursive function that extends the JSON CRDT with the list item's content. We use a recursive function since the value of the list item could either be a string, a list, or a map, which may contain further nested list or map items. The recursive function extends the JSON CRDT with the string value described in the algorithm (Lines 6 to 9). If the value is a list or a map, it extends the JSON CRDT (Lines 11 to 14 or 16 to 19, respectively.) When the value of the JSON object is a map (Lines 16 to 19), we follow the same approach as the list type, but we extend the element with the key of the key-value pairs in the map instead of the key of the current JSON object.

To limit the complexity of our prototype, the JSON lists in our system only support string, map, and list. Therefore, when clients require to use other data types, such as numbers or Boolean, they should convert the desired data type to strings.

4.4 Potentials and Limitations of FabricCRDT over Fabric

Numerous CRDT use cases, such as data metering, global voting platforms, and shared document editing applications, benefit from the decentralized trust offered by FabricCRDT [67, 70, 171, 172, 173, 174]. Our CRDT-enabled permissioned blockchain facilitates the realization of these use cases.

FabricCRDT, as an extension to Fabric, supports all use cases that can be implemented on Fabric. However, based on FabricCRDT's design requirements, by taking advantage of CRDTs, we offer two additional properties that benefit the CRDT use cases. FabricCRDT ensures that (1) all submitted transactions that pass the endorsement policy validation are committed successfully (*No Transaction Failure* requirement) and (2) as an SEC system, no client updates are lost when concurrent updates on the shared key-value pairs are submitted (*No Update Loss* requirement).

One major use case that benefits from FabricCRDT are collaborative document editing platforms [171, 175]. These platforms provide an environment for clients to work on shared documents concurrently. Due to the inherent concurrent nature of these platforms, conflicts from updating the shared content may frequently occur. CRDTs are a practical technique for resolving these kinds of conflicts [171, 175]. Developers can create FabricCRDT-based document editing applications using CRDT features that our system offers, like JSON CRDTs. On FabricCRDT, documents are stored as JSON objects, and edit updates are committed as CRDT transactions. According to our design requirements, updates are merged without losing the client's data, and no CRDT transaction containing the updates fails, so clients do not need to redo and resubmit their edits. Clients also benefit from the trust and security that FabricCRDT offers.

Another prominent application of permissioned blockchains is supply-chain management applications for tracing and ensuring the quality of different products from food to pharma industries [176, 177]. During transportation and storage, sensitive goods like drugs, fresh fruits, and vegetables should be kept within specific conditions, such as temperature, humidity, and light. To ensure that these goods are treated in compliance with regulations and guidelines, sensors continuously monitor the goods and record the readings on the blockchain to keep them secured against manipulations. Although storing a stream of sensor readings from IoT devices can be implemented on Fabric, we argue that this use case is a better fit for FabricCRDT. Depending on the system's design, different readings from different IoT devices may collide, for example, when a temperature sensor and a humidity sensor concurrently submit records to update a shared list of the sensor readings of the same good. Using FabricCRDT, it is ensured that conflicts are merged automatically and that all sensor data end up in the world state (*No Update Loss* requirement). Due to the resource limitations of IoT devices (e.g., regarding energy), the extra effort required for resubmitting failed transactions may be prohibitive. FabricCRDT makes it possible for IoT devices to submit transactions *once* without needing to e concerned about transaction failures and data loss (*No Transaction Failure* requirement).

There exist limitations to FabricCRDT. Use cases that require transactional isolation of repeatable reads [178] or contain non-I-confluent invariant conditions may not be implemented on our system, as FabricCRDT commits transactions even if their read-set is outdated. This includes use cases for transferring assets. For example, financial applications like *SmallBank* [36] or *FabCoin* [13], developed for Fabric, are bad choices to be adapted as a CRDT-based blockchain application. These applications represent asset creation and transfers between owners. Interacting with the state of such applications using CRDT-transactions results in vulnerabilities, such as the double-spending attack [134], where an attacker submits several transactions to transfer a single asset to numerous owners. On Fabric, only one of the attacker's transactions is successfully committed. The MVCC validation fails the other transactions since the first successfully committed transaction outdates the read-set of other transactions. However, FabricCRDT skips the MVCC validation, merges the transactions' values, and successfully commits all of the attacker's transactions.

4.5 Evaluation

In the following section, we provide a comprehensive evaluation of FabricCRDT. We also conducted several experiments to compare the performance of FabricCRDT to Fabric.

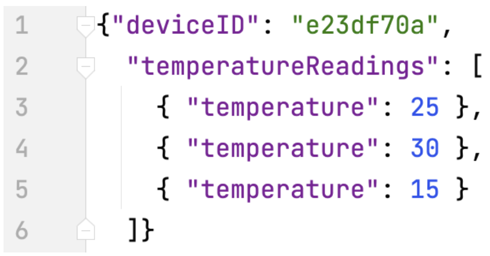
4.5.1 Experimental Applications

The blockchain research community needs a standard workload and benchmarking approach for evaluating different blockchain systems. Benchmarks such as TPC-C [179] and TPC-H [180] from the database community are not directly applicable to blockchains. They are created for database systems and are not directly compatible with FabricCRDT and Fabric. Adapting these workloads to the transactional structures of Fabric or other blockchain systems requires a steady community effort.

As explained in the previous section, the supply-chain use cases are among the regular applications of permissioned blockchains. To evaluate the performance of FabricCRDT and Fabric, we developed an IoT-based application to monitor the temperature of perishable goods in a supply-chain scenario, as follows:

IoT Application – We implemented a set of chaincodes for FabricCRDT and Fabric for storing temperature records. Each chaincode receives and stores a set of temperature readings of an IoT device. The device also sends its identifier. Upon execution, the chaincode first reads a key-value pair from the ledger, where the key is the device's identifier, and the value is a JSON object containing the previous temperature readings of the device. Then, the new temperature reading is inserted into the JSON object. Finally, the device's identifier and the modified JSON object are sent as a key-value pair to be committed to the ledger.

For example, Figure 4.5.1 shows the JSON object that a transaction submits to be committed with one property for the device's identifier and a list containing three temperature readings. For each experiment, the structure of the JSON object and the number of submitted JSON objects differ and are control variables, which we specify accordingly. However, the logic and behavior of the chaincode are the same for all experiments.



```
{"deviceID": "e23df70a",
 "temperatureReadings": [
   { "temperature": 25 },
   { "temperature": 30 },
   { "temperature": 15 }
 ]}
```

Figure 4.5.1: Sample JSON object submitted by a transaction of IoT application.

4.5.2 Workloads, Control Variables and Metrics

We created a custom workload based on the IoT application. While creating the workload, we focused on understanding the limitations and potentials of a CRDT-enabled Fabric. Since standard transactions in FabricCRDT and Fabric are processed based on identical workflows, both systems show similar performance for conflict-free workloads. For this reason, we evaluate the performance of FabricCRDT on workloads of conflicting read-write-transactions for every except one experiment. We perform one set of experiments with workloads consisting of conflicting and non-conflicting transactions in different ratios.

The experiments consist of several control variables. Some control variables are set constant for all experiments. However, others are configured differently for every experiment. We first explain the constant control variables. During the execution of the experiment, a fixed number of four clients submitted 10,000 transactions on FabricCRDT and Fabric. The load on the peers is distributed uniformly. The number of organizations, peers per organization, orderers, and channels is constant. All experiments of FabricCRDT and Fabric are conducted with a network of three organizations, two peers per organization, one orderer node, and one channel. We configured the number of organizations, peers, orderers, and channels constant since FabricCRDT uses the same components as Fabric, responsible for the communication between different components over the network. Hence, we focus on evaluating and comparing the internal mechanism of peers in FabricCRDT and Fabric. We also configured the maximum and preferred number of bytes for a block to 128 MB and the blocking timeout to 2 seconds.

The experiments also include several control variables configured differently for each experiment. The transaction arrival rate is a control variable, defined as *transactions per second (tps)* of the system as the total number of transactions submitted by all clients to the system. The other control variable is the block size regarding the maximum number of transactions in a block. We configured the number of read-write key-value pairs in the chaincode. The complexity of JSON objects written to the ledger concerning the number of keys and their nesting depth is also a control variable.

Each experiment is conducted on an initially empty ledger. Before executing the experiment, we populate the ledger with keys read during the experiment. The executed experiments do not include any Byzantine or non-Byzantine failures.

We collect several performance metrics during the experiment. We measure the *transaction throughput*, the *average transaction latency*, and the *number of successfully committed transactions*. The transaction throughput is the total number of successfully committed transactions divided by the total time required to commit these transactions. The transaction latency is the response time per transaction from sending the proposal until receiving the acknowledgment from the peer on whether the transaction is committed successfully. The number of successfully committed transactions specifies the total number of transactions in the workload successfully committed to the ledger by the peers.

4.5.3 Experimental Setup

We deployed FabricCRDT and Fabric networks on a *Kubernetes* v1.11 cluster consisting of three *controller* nodes, three *worker* nodes, one *DNS* and *load balancer* node, and one *Network File System* node. All nodes run on *Ubuntu* 16.04 virtual machines (VMs) with 16 vCPUs and 41 GB RAM. All VMs are initiated on the *OpenStack Mitaka's KVM* and interconnected by 10 GB Ethernet. We use *CouchDB* [42] as the world state database and *Apache Kafka/Zookeeper* [181, 182] for the ordering service.

We used *Hyperledger Caliper (Caliper)* v0.1.0 [183] for generating and submitting transactions and collecting the performance metrics. Caliper is a performance benchmarking framework developed by the Hyperledger Project to study the performance of a few blockchains, such as Fabric and Ethereum. We extended and adapted Caliper to be able to evaluate FabricCRDT.

4.5.4 Experimental Results for IoT Applications

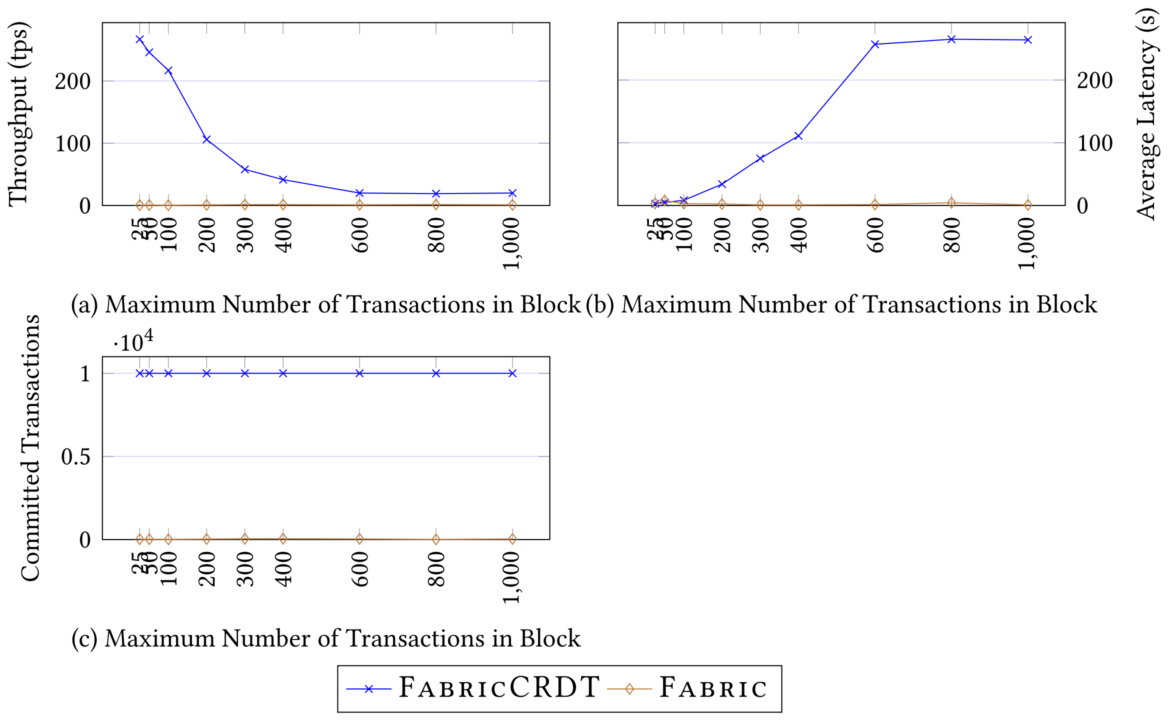
Table 4.5.1 displays the control variables and their default values for the IoT application on FabricCRDT and Fabric. For each experience, as explained in the following parts, we set a few control variables to specific configurations and the other control variables to the default value.

Table 4.5.1: Default values of control variables for the IoT experimental application.

Control Variable	**Default Configuration**
FabricCRDT's maximum block size	25 transactions
Fabric's maximum block size	400 transactions
Transaction arrival rate	300 tps
Number of read key-value pairs per transaction	1 pair
Number of write key-value pairs per transaction	1 pair
Number of keys in JSON object	2 keys
Depth of each key's value	1st key: 1 depth, 2nd key: 2 depth

Effect of Different Block Sizes – We examine the impact of the block size on the total number of successful transactions, the throughput of successful transactions, and the average latency of successful transactions. We gradually increased the maximum allowed number of transactions in a block from 25 to 1000 transactions. Each chaincode invocation reads one key-value pair from the ledger and writes one key-value pair back. The JSON object written to the ledger has two keys, containing a string constant and a list, as specified for the default values in Table 4.5.1.

In order to find the best configuration of FabricCRDT and Fabric under worst-case workloads, all transactions modify identical keys. Hence, the transactions depend on each other and are conflicting. FabricCRDT merges every key-value pair of every transaction in each block. Therefore, a higher number of transactions in a block potentially induces a higher overhead for the peer to merge a higher number of JSON CRDTs.



(a) Maximum Number of Transactions in Block (b) Maximum Number of Transactions in Block

(c) Maximum Number of Transactions in Block

Figure 4.5.2: Effect of block size on FabricCRDT and Fabric.

The results of the experiments are summarized in Figure 4.5.2. In Figure 4.5.2(a), we observe that FabricCRDT demonstrates a higher throughput for smaller block sizes. The main reason for the degradation of throughput in FabricCRDT with larger block sizes is the higher overhead required for merging a higher number of JSON CRDTs. For

FabricCRDT, the highest throughput overall was 267 transactions per second for a block size of 25 transactions.

Figure 4.5.2(b) shows that FabricCRDT demonstrates a higher transaction latency for larger block sizes because of the lower throughput, resulting in more time required to commit the transactions. In Figure 4.5.2(c), we observe that FabricCRDT successfully commits *all* submitted transactions. In Fabric, several conflicting transactions fail due to MVCC validation, while in FabricCRDT, all conflicts are automatically merged, and all transactions are successfully committed.

In the following experiments, we set the block size to 25 transactions per block for FabricCRDT and 400 transactions per block for Fabric. This way, we run both systems in their best configuration to get a fair comparison.

Effect of Different Number of Reads and Writes – To understand the effect of a higher number of key-value pairs in the transaction's read-write-set, we gradually increased the number of key-value pairs that were read from and written to the ledger in the chaincode. We configured 1, 3, or 5 key-value pairs for each experiment to be read and written, as shown in Table 4.5.2. During each experiment, we kept the read and wrote keys identical for all transactions. For example, in the experiment with five reads and five write keys, we read or write the same set of 5 distinct keys in every transaction.

Table 4.5.2: Effect of an increasing number of read and write key-value pairs.

Control Variable	**Executed Configuration**
Number of read key-value pairs per transaction	{1, 3, 5 } pair
Number of write key-value pairs per transaction	{1, 3, 5 } pair

Figure 4.5.3 summarizes the results of the experiments. As expected, we observe in Figure 4.5.3(a) that the throughput of FabricCRDT decreases as the read-write-set grows because of the increased overhead for merging a higher number of CRDTs. We observe that FabricCRDT is affected by both the number of reads and writes in the transactions. In comparison to FabricCRDT, Fabric shows a lower transaction throughput (Figure 4.5.3(a)) and a lower total number of successful transactions (Fig-

ure 4.5.3(c)). However, FabricCRDT has a higher commit latency in comparison to Fabric (Figure 4.5.3(b)).

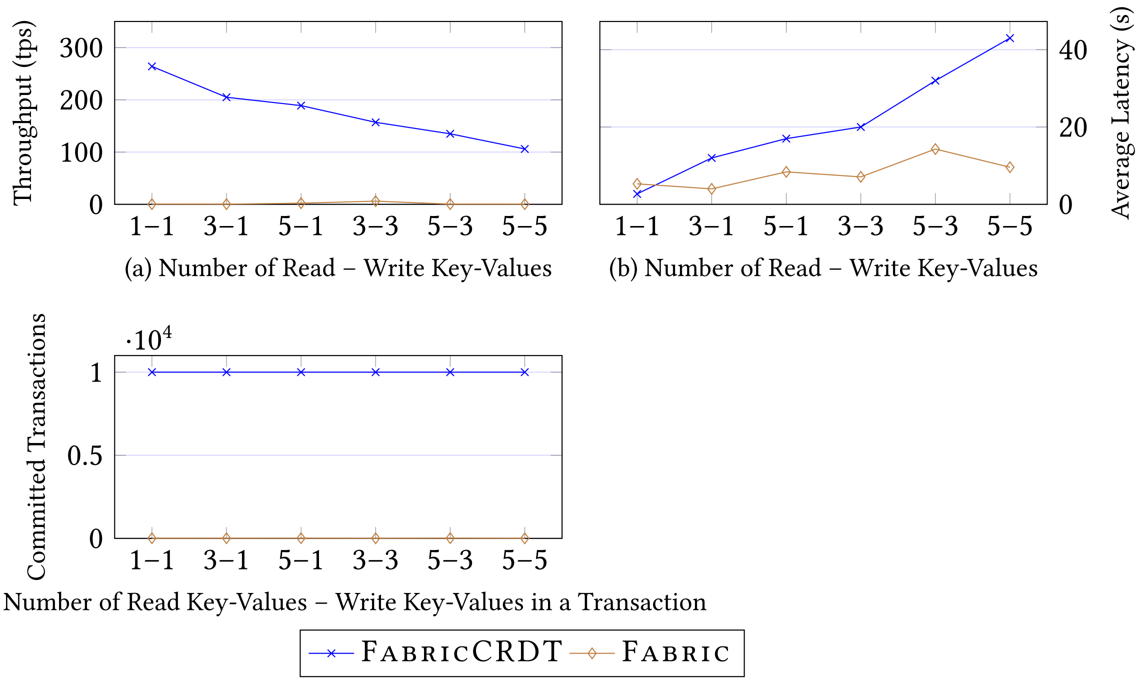
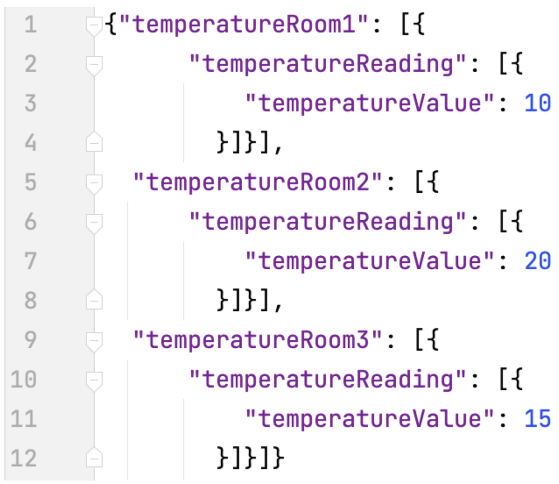


Figure 4.5.3: Effect of a different number of read and write key-value pairs.

Impact of Varying Complexity of JSON Objects – We evaluate the effect of varying complexity of JSON objects committed to the ledger. In particular, we study the changes to the throughput and latency of FabricCRDT, as merging more complex JSON objects induces more overhead. Table 4.5.3 shows the experiment's configuration. Each transaction reads one JSON object from the ledger with a specific number of keys and a specific nesting depth of the values. Then, the transaction modifies the JSON object and writes back the JSON object to the ledger.

For example, Figure 4.5.4 illustrates a JSON object with "3-3 complexity". In other words, the transaction submits a JSON object with three key-value pairs, where each value has a depth of three from the root of the JSON object.

Figure 4.5.5 summarizes the results of the experiments. We observe that the throughput decreases and the latency increases for FabricCRDT as the complexity of JSON objects



```
{"temperatureRoom1": [{
        "temperatureReading": [{
            "temperatureValue": 10
        }]}],
  "temperatureRoom2": [{
        "temperatureReading": [{
            "temperatureValue": 20
        }]}],
  "temperatureRoom3": [{
        "temperatureReading": [{
            "temperatureValue": 15
        }]}]}
```

Figure 4.5.4: A JSON object with "3-3" complexity.

Table 4.5.3: Impact of different complexity of JSON objects.

Control Variable	**Executed Configuration**
Number of keys in JSON object	{2, 3, 4, 5, 6 } keys
Depth of each key's value	{2, 3, 4, 5, 6 } depth

increases, as demonstrated in Figure 4.5.5(a) and Figure 4.5.5(b), respectively. The reason is the increased overhead for merging larger CRDT objects. Unlike FabricCRDT, Fabric does not interact with the content of the JSON objects. Therefore, the throughput and latency of Fabric are not correlated to the complexity of the JSON objects.

Impact of Increasing Transaction Arrival Rates – We evaluated the effect of increasing transaction arrival rates on FabricCRDT and Fabric. We gradually increased the transactions arrival rate from 100 to 500 tps.

As the results of the experiments in Figure 4.5.6(a) show, FabricCRDT's throughput increases until it reaches a saturation point at about 250 tps. Meanwhile, Figure 4.5.6(b) shows that the latency increases as the transaction arrival rate increases for FabricCRDT. The enormous increase in latency in FabricCRDT is attributed to the effects of queuing when the transaction arrival rate exceeds the throughput.

Impact of Different Percentage of Conflicting Transactions – In order to understand

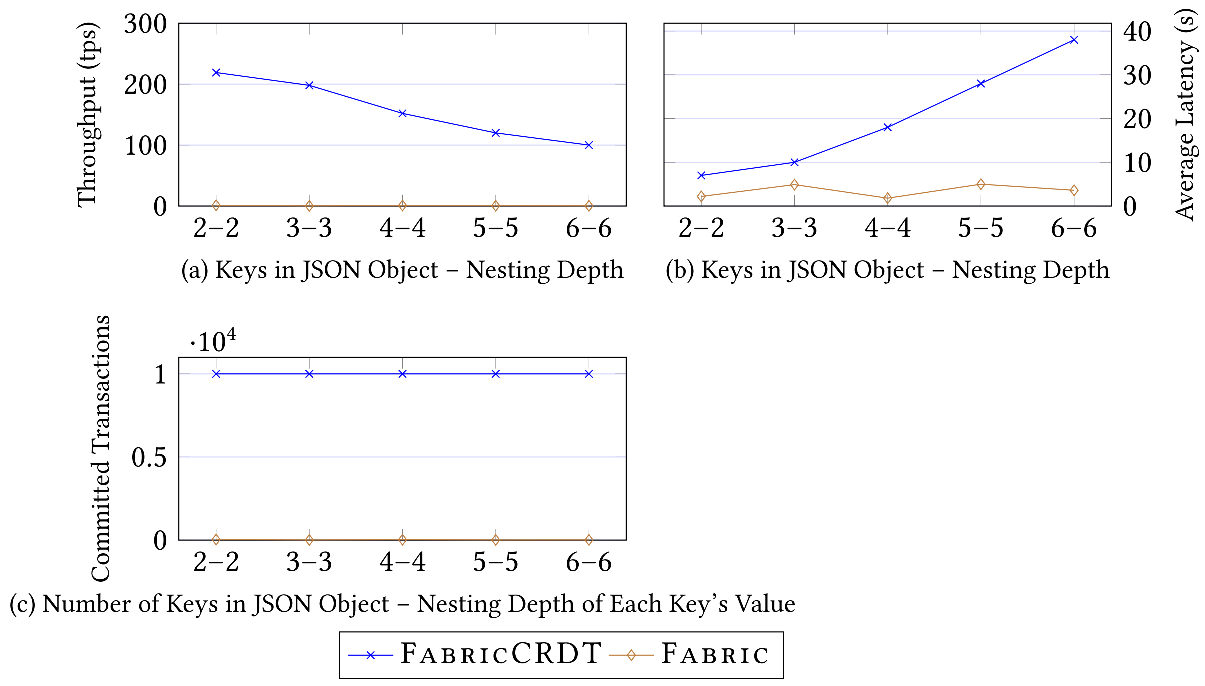


Figure 4.5.5: Effect of the increasing complexity of JSON objects.

the limitations and potentials of FabricCRDT, in the previous experiments, we used workloads where all transactions are conflicting. However, in the real-world deployment of FabricCRDT and Fabric, the conflict rate of transactions in the workload varies, and blocks contain conflicting and non-conflicting transactions. To study the effects of different percentages of conflicting transactions in the workload, we gradually increased the conflict rate in the transactions. For each experiment, a fixed percentage of transactions are conflicting, where the conflicting transactions are merged in FabricCRDT and rejected in Fabric.

Figure 4.5.7 summarizes the results of the experiment. We observe for workloads where a lower percentage of transactions are conflicting, that the throughput and latency of FabricCRDT are similar to Fabric (Figure 4.5.7(a) and Figure 4.5.7(b)). However, when the percentage of conflicting transactions increases, the number of failures also increases in Fabric (Figure 4.5.7(c)), while no failures occur in FabricCRDT.

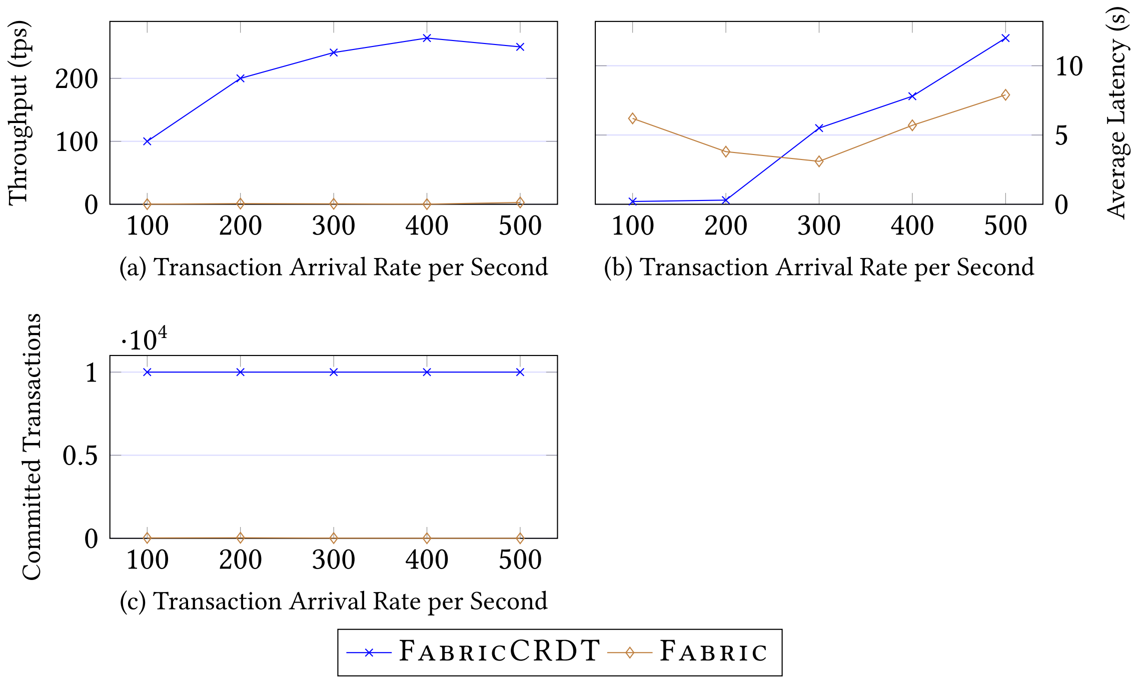


Figure 4.5.6: Impact of increasing transaction arrival rates for IoT application..

4.5.5 Discussion

FabricCRDT bypasses the MVCC validation and merges the conflicting transactions instead of rejecting them. Therefore, it successfully commits *all* transactions in all experiments. In stark contrast, Fabric only successfully commits very few transactions for a higher transactions conflict rate. The significantly improved successful commit rate of FabricCRDT contributes considerably to increasing Fabric's throughput and improving its scalability by eliminating transaction failures. Besides the improved throughput, eliminating the transaction failures and offering the *No Transaction Failure* and *No Update Loss* decreases the developer's burden for developing blockchain-based applications. Handling a large number of transaction failures in the application, as observed in Fabric, increases the complexity of developing applications.

In our experiments, we observed that FabricCRDT, in comparison to Fabric, suffers from a higher latency, directly resulting from the extra processing time required for merging many JSON CRDTs. As the JSON object increases in size, its metadata and complexity

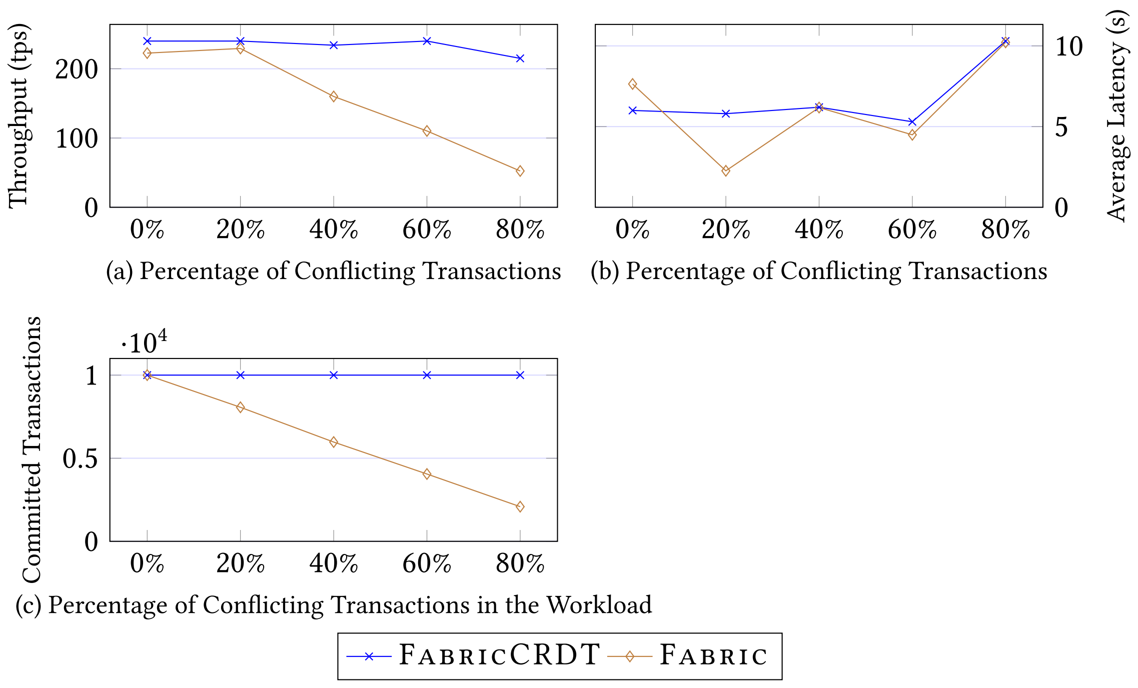


Figure 4.5.7: Effect of different percentages of conflicting transactions.

increase when more keys and values are added. However, in most of our evaluations, we investigated the worst-case scenarios where all transactions in a block are conflicting and must be merged. For scenarios where conflicting and non-conflicting transactions coexist, experiments show that Fabric and FabricCRDT have comparable latency and throughput.

4.6 Summary

This chapter introduced an approach to eliminating failures of the concurrent dependent transactions on Fabric. We presented FabricCRDT, an extension of Fabric, that successfully commits conflicting and concurrent transactions. FabricCRDT employs an approach for identifying conflicting transactions within one block and uses a CRDT-based technique to resolve the conflict transactions and merge their values without data loss.

We conducted extensive evaluations to study the performance of FabricCRDT and its improved throughput over Fabric. Our finding confirms the applicability of our approach for significantly improving throughput by avoiding transaction failures and decreasing the development effort required for creating real-world blockchain-based applications.

Despite the improved throughput, FabricCRDT can correctly execute I-confluent applications. This is because FabricCRDT refrains from MVCC-based validations used by Fabric, which preserves the I-confluent invariant conditions on Fabric. Another limitation of Fabric is its decreased performance as the size of JSON CRDT increases, demonstrating the requirement for pruning and cleaning metadata from objects.

5

OrderlessChain: A Permissioned Blockchain without Coordination

In this chapter, we present OrderlessChain, a coordination-free permissioned blockchain without total global order of transactions. As explained in the previous chapters, coordination to reach a consensus to offer Byzantine Fault Tolerance and to serialize transactions in a trustless and distributed system is a bottleneck for scalability. In general, decreasing coordination plays a vital role in improving the performance of any distributed systems [39]. A coordination-free blockchain could enable the concurrent execution of transactions, leading to higher throughput and lower latency. However, simply eliminating the coordination may jeopardize the correctness depending on the application. For example, a payment processing application may require rejecting transactions that result in the payee's account's balance turning negative. A coordination-free blockchain cannot preserve this requirement [39, 40].

In contrast, I-confluent invariant conditions can be preserved in a coordination-free distributed system. For example, transactions that only deposit funds to an account can execute 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. Hence the transactions are *commutative* and *convergent*. One technique capable of creating I-confluent commutative and convergent transactions is CRDTs [12].

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 [39]. However, preserving the safety and liveness of applications in a Byzantine environment depends on paying a high coordination cost in coordination-based protocols [8, 25, 26, 36, 49, 184]. By providing a Byzantine coordination-free environment where I-confluent applications continue to be safe and live, we benefit from improved performance while ensuring trust in a trustless environment. Therefore, OrderlessChain uses the properties of permissioned blockchains and CRDTs to offer a coordination-free innovative two-phase *execute-commit* protocol for creating safe and live applications in a Byzantine environment.

This chapter is based on papers on OrderlessChain, either published or under submission [60, 61, 185].

The remainder of the chapter is organized as follows. We explain OrderlessChain's system and failure models in Section 5.1. In Section 5.2, we introduce our novel coordination-free two-phase protocol for the trusted execution of transactions in a trustless environment. Then, we present our approach to model OrderlessChain's use cases with CRDT in Section 5.3. In Section 5.4, we describe implementing the modeled use cases. We also demonstrate our approach for preserving I-confluent application-level correctness requirements in Section 5.5. In Section 5.6, we study the effect of Byzantine actors and provide proof for OrderlessChain's Byzantine Fault Tolerance. Finally, in Section 5.7, we evaluate the performance of OrderlessChain and compare it to the coordination-based protocols of Fabric and FabricCRDT.

5.1 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 [186]. Additionally, studies have shown that coordination in highly concurrent use cases is a bottleneck [8, 36]. For example, in the case of FABRIC, Chacko et al. demonstrated that up to 90 percent of transactions for a digital voting application fail due to FABRIC's coordination-based protocol.

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 $ST_{App} = \cup_{i=1}^{n} ST_{O_i}$ 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 – The developer imposes an application's correctness 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 5.1.1. $\mathbf{ST_{App}}$ ***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 application 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 an affiliated 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 out 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 5.1.2. ***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 5.1.3. ***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 considered 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, to consider the transactions as committed, at least two organizations must commit a valid transaction.

Failure Model – We consider the organizations and clients to be potentially Byzantine. Byzantine organizations or clients can fail arbitrarily. We consider an organization to be non-faulty if and only if the organization processes every transaction according to 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 5.1.4. ***Safety.*** *Only valid transactions are successfully committed.*

Definition 5.1.5. ***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 not valid.
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 tempering 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. 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 $q \geq f + 1$. Additionally, the application is live if $n - q \geq f$. We provide proof of the safety and liveness of OrderlessChain in Section 5.6. 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 [187], if in every partition there exists 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.

5.2 Architecture and Protocol

In the following section, we first explain the architecture of OrderlessChain's network and internal components of organizations. Afterward, we describe our two-phase *execute-commit* protocol for executing and committing transactions.

5.2.1 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; (2) a database. The hash-chain log contains all transactions that the organization has received since the beginning of time in a hash-chain data structure. By sequentially executing every transaction in the hash-chain log, we reach the application state ST_{O_i}. For a more efficient approach, an organization applies each transaction to its database when the transaction is 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)* [188]. 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. OrderlessChain supports executing smart contracts with a Turing-complete logic written in the *Go language* [64]. Each smart contract can contain any number of functions. Each function encapsulates a task that the application performs. To execute a smart contract, clients submit a request to an organization that executes the smart contract with the provided inputs and returns a result.

5.2.2 Protocol and Transaction Lifecycle

OrderlessChain follows a two-phase *execute-commit* protocol and transaction lifecycle. 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 5.2.1 demonstrates the complete transaction lifecycle for an application with endorsement policy $EP : \{q\ of\ n\}$.

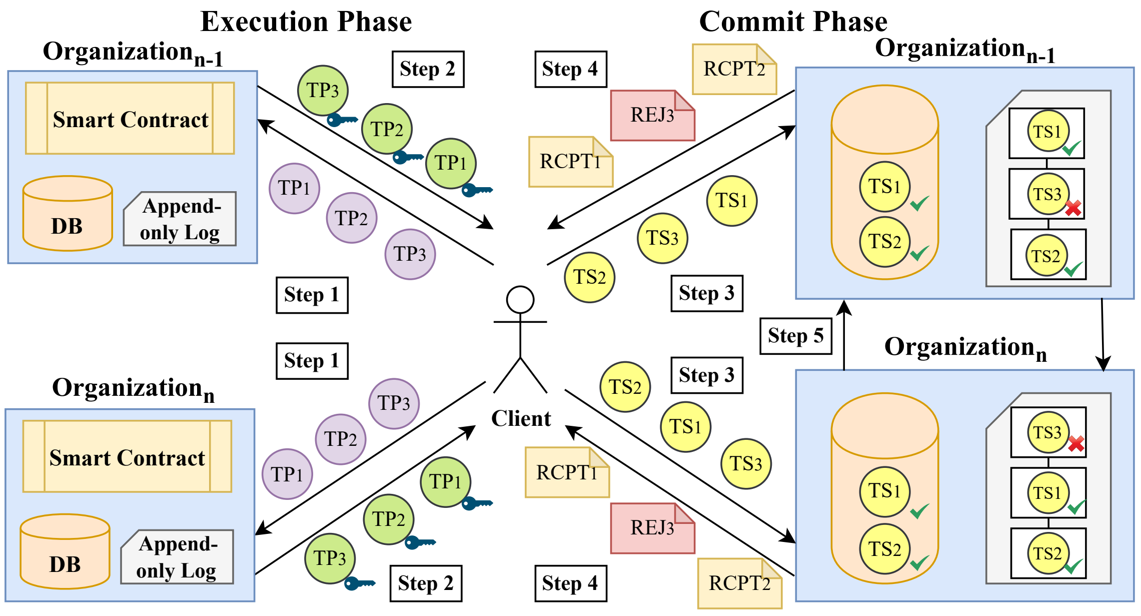


Figure 5.2.1: Transaction lifecycle on OrderlessChain.

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 (Step 1 in Figure 5.2.1).

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 5.2.1). 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.

Phase 2 / Commit Phase – The client waits until it receives the minimum number of endorsements required by the endorsement policy. 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 the operations 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 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 endorsement policy (Step 3). These organizations could be different from those who initially endorsed the proposal. If an organization has not previously committed 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 endorsement policy. The organization hashes the transaction's write-set to verify the validity of endorsements and the client's signature. It uses the public keys of endorsing organizations and clients to verify their signatures. This verification shows that the endorsing organizations created identical write-sets, and the client did not tamper with the write-set. If the transaction passes the signature validation, it is marked as valid. Otherwise, the transaction 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. 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 $RCPT_i = HashSign(Block_h, Valid)$, which is 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 $REJ_i = HashSign(Block_h, Invalid)$ 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 waits until it receives the minimum number of receipts required by the endorsement policy. 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 (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 the client. 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.

5.3 Realizing Decentralized Use Cases on OrderlessChain

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.

5.3.1 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

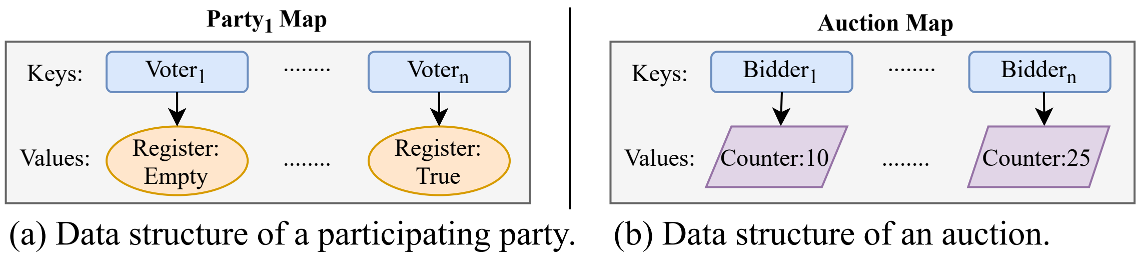


Figure 5.3.1: Application modeling for the voting and auction applications.

discuss modeling two use cases:

Voting Application – One possible solution for modeling our running voting example in a smart contract is shown in Figure 5.3.1(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 and popular use cases of blockchains [189]. 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*.

One possible design is as follows, as shown in Figure 5.3.1(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.

5.3.2 CRDT Abstractions

As explained, CRDTs provide a solution for creating commutative convergent update operations, and we use CRDTs in smart contracts. The proposed OrderlessChain protocol is independent of CRDTs used in smart contracts. CRDTs are also replaceable with alternative techniques that provide commutable operations to develop new types of applications, such as *Operational Transformation* [190, 191]. However, there exists a plethora of CRDTs for various data types. To enable the execution of these CRDTs, their specifications need to be supported by the smart contract execution environment. In the current implementation, OrderlessChain supports the specifications of grow-only counters (G-Counter) [12], CRDT Maps [54], and multi-value registers (MV-Register) [54]. 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, according to the available literature, must be added to OrderlessChain [12, 68, 171].

Table 5.3.1: Modification and read APIs of the CRDTs.

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

The three CRDTs represent the following data structures:

- **G-Counter:** It is a monotonically increasing numeric variable.
- **CRDT Map:** This CRDT is built upon a map data structure. A map is an unordered data structure containing key-value pairs. The key is a unique identifier; the value can be any object.
- **MV-Register:** This is a shared variable capable of containing multiple values at a time.

Every CRDT provides modification and read APIs as shown in Table 5.3.1. 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 modification API calls. The value must be `null` for deleting a value with modification APIs of CRDT Map and MV-Register. 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 5.3.1(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.

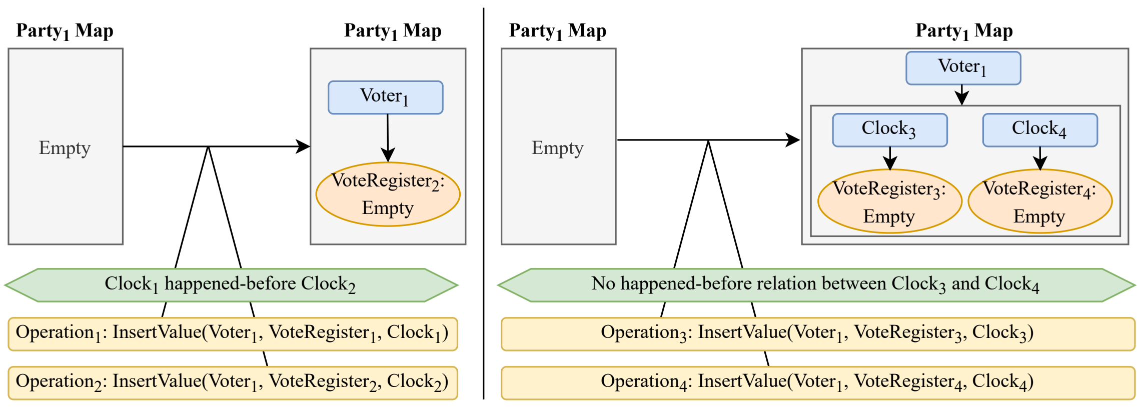
Auction application – In the modeled auction application shown in Figure 5.3.1(b), we use a map for modeling the auction and increase-only counters for bids. Hence, we use a CRDT Map to model the auction's map and G-Counters to model the bids of each bidder.

To evaluate an operation's effects, the operation needs to be applied to the CRDT, which may cause conflicts. The CRDTs must provide a built-in mechanism for resolving conflicts of modification operations. We identify the conflicting operations of the three CRDTs and offer a conflict resolution accordingly:

- **G-Counter:** Every operation increase the counter's value. Hence, the modification operations are inherently commutative and cause no conflict.

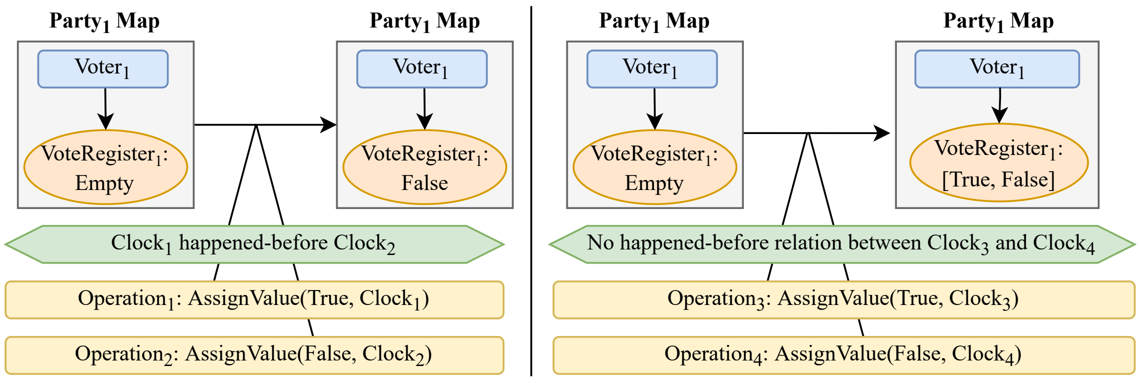
- **CRDT Map:** The modification operations that modify different keys in the map are commutative and non-conflicting and can be applied concurrently. However, the operations that modify identical keys are conflicting. The conflict is resolved based on the happened-before relations among operations. 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 5.3.2.

- **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. However, if the happened-before relation cannot be inferred from the clocks, the register stores all values, as shown in Figure 5.3.3, and the client may specify which value to use as the final value of the register.



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 5.3.2: 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 5.3.3: Applying MV-Register modification operations.

5.4 Implementation

We implemented a prototype of OrderlessChain with the *Go language* [64] and *Protocol Buffer-based gRPC* [192]. We decided on these technologies due to high-performant built-in support for concurrent programming in Go. Due to the efficient serialization and deserialization of gRPC, we used this framework. We open-sourced the code and the smart contracts discussed in this paper [57].

5.4.1 Developing CRDT-enabled 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* [74], which is 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, Figure 5.4.1 shows the function in the smart contract for creating the operations for voting for a party. The function 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 submitting a vote.

```go
func (vc *VotingContract) Vote(SCL contractinterface.SCL, args []string)
                                        (*protos.ProposalResponse, error) {
    voterId := args[0]
    voterClock := args[1]
    votedPartyId := args[2]
    electionId := args[3]
    operationId := SCL.MakeOperationId(voterId, voterClock)
    operationsList := &protos.CRDTOperationsList{
        CRDTObjectId: votedPartyId,
        Operations:   []*protos.CRDTOperation{},
    }
    operationsList.Operations = append(operationsList.Operations,
        &protos.CRDTOperation{
            OperationId:   operationId,
            ValueType:     protos.CRDTOperation_MV_REGISTER,
            Value:         "true",
            Clock:         voterClock,
            OperationPath: []string{voterId},
    })
    SCL.PutCRDTOperations(votedPartyId, operationsList)
    for _, partyId := range vc.Elections[electionId].AllParties {
        if partyId != votedPartyId {
            operationsList = &protos.CRDTOperationsList{
                CRDTObjectId: partyId,
                Operations:   []*protos.CRDTOperation{},
            }
            operationsList.Operations = append(operationsList.Operations,
                &protos.CRDTOperation{
                    OperationId:   operationId,
                    ValueType:     protos.CRDTOperation_MV_REGISTER,
                    Value:         "false",
                    Clock:         voterClock,
                    OperationPath: []string{voterId},
            })
            SCL.PutCRDTOperations(partyId, operationsList)
        }
    }
    return SCL.Success(), nil
}
```

Figure 5.4.1: The smart contract of the voting application to cast a vote.

5.4.2 Applying CRDT 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 [193, 194]. Hence, we implemented an optimization to address this issue. As Section 5.2 explains, the ledger contains a database beside the hash-chain log. The database is updated with every valid transaction. It consists of a conventional key-value database, namely *LevelDB [65]*, 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 3. 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 [195].

Algorithm 3 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 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 3 is $O(n)$, where n is the number of operations being applied. Hence, the complexity of applying of operation is $O(1)$.

In Section 5.6, we prove the SEC property. However, first, we demonstrate that the

Algorithm 3: Applying CRDT operations to the object on OrderlessChain.

```
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         $CRDTObj.Apply(Location, Op_i.Val, Op_i.ValType, Op_i.Clock)$
```

application state ST_{O_i} is independent of the order of transactions. We formulate the following lemma:

Lemma 5.4.1. *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 the built-in conflict resolution mechanism, applying the operations in the write-set of operations by using Algorithm 3 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. □

5.5 Preserving Invariant Conditions

As explained, submitting a set of transactions $\{TS_1, ..., TS_m\}$ in a coordination-free manner preserves the invariants $\{I_1, ..., I_s\}$ if the set of transactions is I-confluent concerning the invariants. Organizations can commit a set of I-confluent transactions 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. As developers define the logic for creating operations in a smart contract, they 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. We consider an election with two participating parties to determine that the invariant can be preserved by creating I-confluent operations. As explained in Section 5.4, 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*.

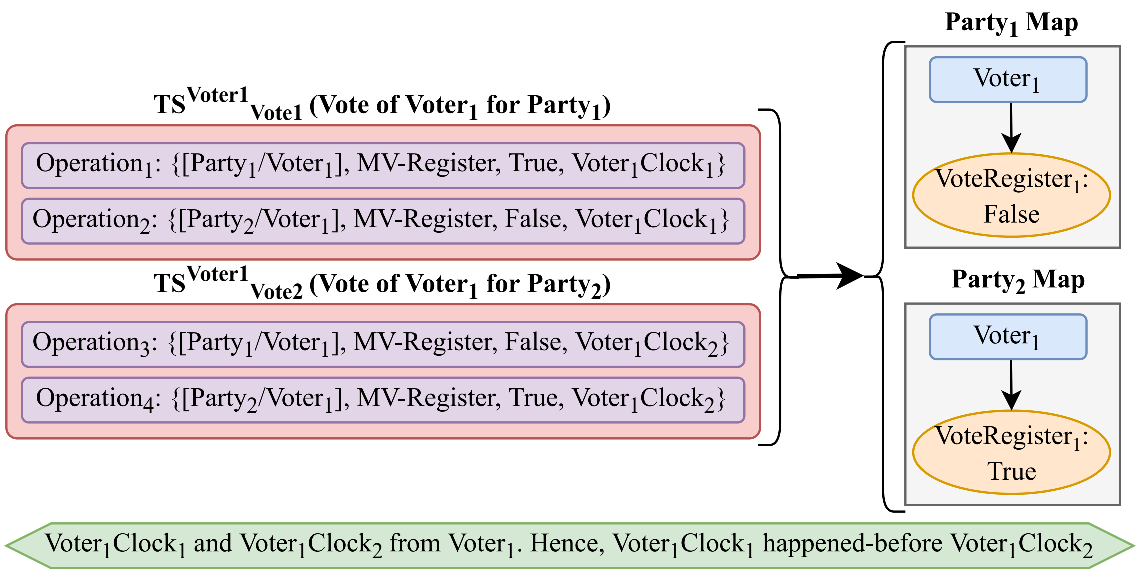

$TS^{Voter1}{}_{Vote1}$ (Vote of $Voter_1$ for $Party_1$)
$Operation_1$: {[$Party_1$/$Voter_1$], MV-Register, True, $Voter_1Clock_1$}
$Operation_2$: {[$Party_2$/$Voter_1$], MV-Register, False, $Voter_1Clock_1$}
$TS^{Voter1}{}_{Vote2}$ (Vote of $Voter_1$ for $Party_2$)
$Operation_3$: {[$Party_1$/$Voter_1$], MV-Register, False, $Voter_1Clock_2$}
$Operation_4$: {[$Party_2$/$Voter_1$], MV-Register, True, $Voter_1Clock_2$}
$Party_1$ Map
$Voter_1$
$VoteRegister_1$: False
$Party_2$ Map
$Voter_1$
$VoteRegister_1$: True
$Voter_1Clock_1$ and $Voter_1Clock_2$ from $Voter_1$. Hence, $Voter_1Clock_1$ happened-before $Voter_1Clock_2$

Figure 5.5.1: Preserving the invariant condition of the voting application.

There is no coordination among organizations, so 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 $Voter_1$, as shown in Figure 5.5.1. Each transaction contains two operations. $Voter_1$ 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 $Voter_1$. 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.

5.6 Byzantine Fault Tolerance

We assume that organizations or clients can be potentially Byzantine. We discuss four potential attacks by Byzantine clients and the approach of the system to prevent these attacks. Hence, Byzantine clients cannot jeopardize the system:

1. A Byzantine client might send proposals to the organizations without submitting the transaction to be committed. This behavior does not leave any lasting side effects on the system. However, it can be used for DDoS attacks on the system and decrease performance. However, note that only authenticated clients can communicate with the organizations. Therefore, if the authenticated Byzantine clients try to overload the system, they can be detected, and their permissions can be revoked.
2. A Byzantine client may send the transactions to some organizations during the commit phase and avoid sending them to other organizations. As the organizations gossip the transactions to other organizations after committing the transaction, all organizations eventually receive the transactions.
3. Clients may send the wrong logical clocks to the organizations with the proposals. If clients send different clocks to different organizations with the same proposal, then the operations in the endorsements do not match. Hence, the client cannot create a valid transaction and cannot successfully commit the transaction.
4. Suppose the client does not increment the clock with every proposal. In that case, no happened-before relation between clocks can be inferred, and the CRDT's conflict resolution mechanism manages the operations accordingly, as explained in Section 5.3, without causing data corruption.

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

Theorem 5.6.1. *Let the endorsement policy for an application be $EP : \{q \text{ of } n\}$ 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.

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. Therefore, if the messages are suspected to be lost, they can be resent by clients.

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. As 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 contact, 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 5.6.2. *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 the background chapter, an SEC system must satisfy two requirements of *eventual delivery of transactions* and *strong convergence of nodes*. In Theorem 5.6.1, we demonstrated that every valid transaction is committed for a safe and live application. Additionally, non-faulty organizations gossip about 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 5.4.1, 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. As the *eventual delivery of transactions* requirement for the safe and live application is satisfied, if 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 5.4.1, 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. □

5.7 Evaluation

We first evaluate OrderlessChain. Then, we compare it to Fabric and Fabric-CRDT. We decided to compare our system to these systems, as Fabric is the most prominent coordination-based permissioned blockchain capable of executing Turing-complete applications similar to OrderlessChain. FabricCRDT , to the best of our knowledge, is the only permissioned blockchain capable of running CRDT-enabled applications.

We compare OrderlessChain to a prototype of Fabric and FabricCRDT, which we implemented based on the Go language, gRPC, and LevelDB. We do so because the original Fabric and FabricCRDT offer many security and network-related features that we do not provide in OrderlessChain. As these features impose performance penalties, we replaced the original implementations for a fair comparison since we intended to compare our coordination-free protocol to their coordination-based protocols and not our source code to their codes. The extensive evaluation of Fabric performed by Chacko et al. [8] demonstrates these performance issues and confirms the fairness of our approach for using our prototypes of Fabric and FabricCRDT. 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.

5.7.1 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 Fabric and FabricCRDT. Every application consists of one smart contract, and in total, we developed seven smart contracts. 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 $Modify(ClientId_i, Clock_i, ObjCount, OpsPerObjCount, CRDTType)$ and $Read(ObjCount)$. The $Modify$ function receives the client identification and clock, the number of CRDT objects to be modified, the number of operations per each CRDT object modification, and the CRDT type. CRDT type is either a G-Counter, CRDT Map, or MV-Register. The write-set of the transaction includes $ObjCount \times OpsPerObjCount$ operations. The $Read$ function reads a specific number of CRDT objects as specified by $ObjCount$.

Voting Application – We developed applications based on the voting example for OrderlessChain, Fabric, and FabricCRDT. The application's smart contract for OrderlessChain has two functions: $Vote(Voter_i, Clock_i, Party_j, Election_l)$ and $ReadVoteCount(Party_j, Election_l)$. 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 5.4. $ReadVoteCount$ retrieves the current number of votes of $Party_j$. The smart contracts for Fabric and FabricCRDT also include $Vote$ and $ReadVoteCount$ functions, which are implemented based on the best practices for developing smart contracts on these systems [8, 13].

Auction Application – The auction applications are implemented for OrderlessChain, Fabric, and FabricCRDT. The application's smart contract of OrderlessChain has two functions: $Bid(Bidder_i, Clock_i, BidIncrease_i, Auction_j)$ and $GetHighestBid(Auction_j)$. 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 for Fabric and FabricCRDT also include a Bid and a $GetHighestBid$ function.

5.7.2 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 as *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 of organizations, 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 $EP : \{q\ of\ n\}$, the clients send the proposals and transactions to exactly q organizations. Organizations can contain several peers on Fabric and FabricCRDT. In our experiments, each organization of these systems consists of one peer. Additionally, the blocks created by Fabric's and FabricCRDT's ordering service have a size of 50 transactions. Based on our investigation and other studies [8], this block size yields good performance. Each experiment is executed for 180 seconds.

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 [8]. 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.

5.7.3 Experimental Setup

Each organization of OrderlessChain, Fabric, and FabricCRDT runs on an individual KVM-based Ubuntu 20.04 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 delay, 4 ms jitter, and 100 Mbits 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. The ordering service of Fabric and FabricCRDT runs on a separate VM.

We also developed a distributed benchmarking tool written in the Go language that orchestrates a distributed deployment of clients, generates and submits transactions, and collects performance metrics.

5.7.4 Experimental Results for Synthetic Applications

Table 5.7.1 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 for each experiment, and the other control variables are set to the default value. The results of the experiments are summarized from Figure 5.7.1 to Figure 5.7.11. In Figure 5.7.1, Figure 5.7.2, Figure 5.7.4, Figure 5.7.5, Figure 5.7.6, Figure 5.7.7, Figure 5.7.8, Figure 5.7.9, the left figure demonstrates the average, 1st, and 99th percentiles transaction latencies for executed configurations. The right figure shows the throughput of the transaction.

Impact of Increasing Transaction Arrival Rates – As shown in Figure 5.7.1, we observe that the throughput increases with an increasing transaction arrival rate. Although the latency increases as the load on the system increase, we observe that the throughput increases ten folds while the latency decreases two folds. This result confirms the scalability of OrderlessChain for an increasing load on the system.

Table 5.7.1: Control variables of the evaluated synthetic application.

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

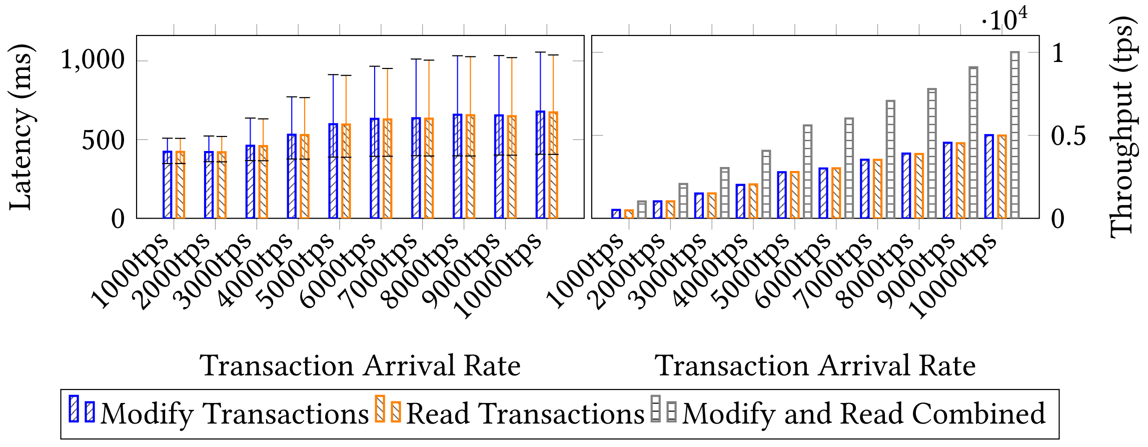


Figure 5.7.1: Impact of increasing transaction arrival rates.

Effect of Increasing Number of Organizations – We studied the effect of increasing the number of organizations on throughput and latency, as shown in Figure 5.7.2. For each experiment, we set the endorsement policy to $EP : \{4\ of\ NumerOfOrgs\}$. We observe that the system scales for an increasing number of organizations without affecting the throughput and latency. This result shows that OrderlessChain scales horizontally, and by increasing the number of organizations in the system, it can process a higher number of transactions per second. These findings are further confirmed by the experiments we executed to compare the average latency to throughput for an increasing number of organizations and arrival rates, as shown in Figure 5.7.3.

Effect of Increasing Number of Operations per Object – The result for an increasing number of operations per object demonstrates that the throughput and latency are

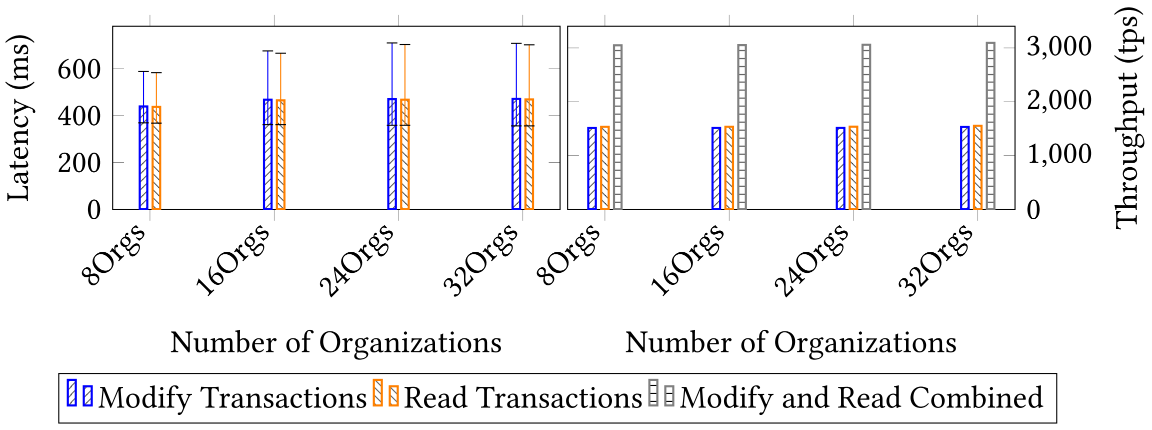


Figure 5.7.2: Effect of an increasing number of organizations.

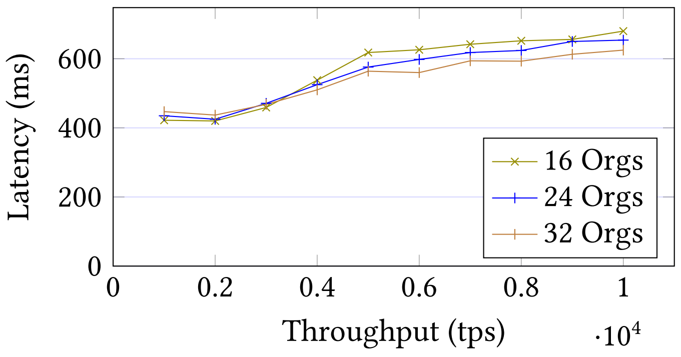


Figure 5.7.3: Latency to throughput for an increasing number of organizations.

unaffected by the number of operations, as shown in Figure 5.7.4. The reason is that applying every operation requires $O(1)$. Since applying different objects' operations is performed in parallel, the number of operations per object is not a bottleneck.

Impact of Increasing Number of Objects – In contrast to the increasing number of operations per object, we observed that the latency increases for a higher number of objects in the transaction, as shown in Figure 5.7.5. This increased latency can be explained due to our approach to lock the objects in the cache to avoid concurrent reads and writes while applying the modify-transactions. Therefore, transactions with more objects must wait for the lock on the objects to be released before their operations are applied, causing the latency to increase.

Effect of Different Endorsement Policies – With an increasing number of organi-

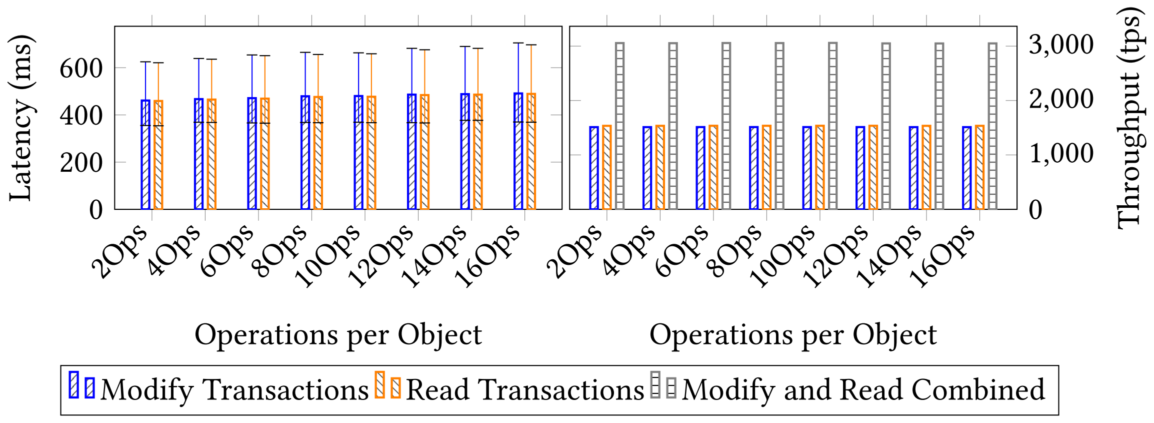


Figure 5.7.4: Effect of an increasing number of operations per object.

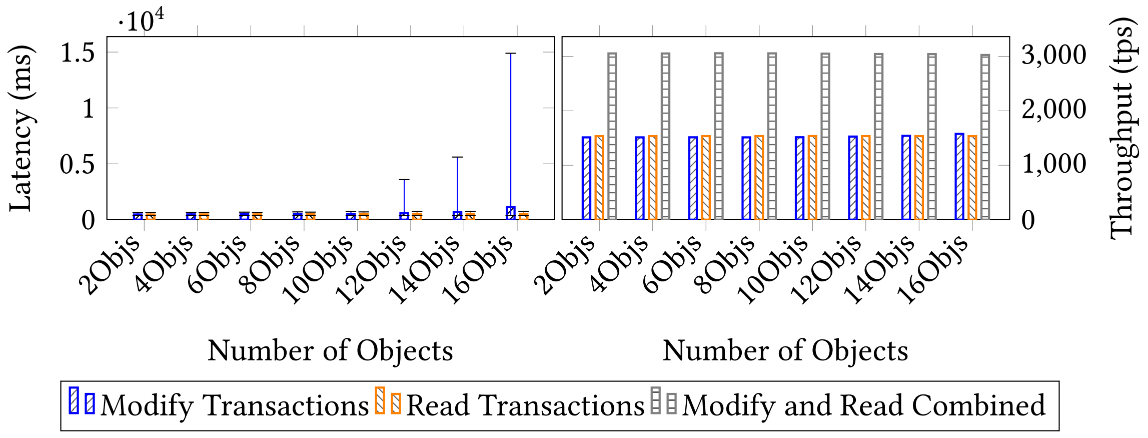


Figure 5.7.5: Impact of an increasing number of objects.

zations required by the endorsement policy, we observe that the latency increases as the load on the organization increase, as shown in Figure 5.7.6. The increased load on the organizations causes the transactions to be queued to be processed and committed, subsequently increasing the latency.

Effect of Different CRDT Types – We observed that latency and throughput are independent of CRDT types, demonstrated in Figure 5.7.7. The reason is that applying each CRDT operation requires $O(1)$, and independent of the CRDT, one operation is included in every modify-transactions.

Impact of Different Percentage of Read- and Modify-Transactions – We gradually decreased the modify-transactions in the workload from 90 percent to 10 percent. As

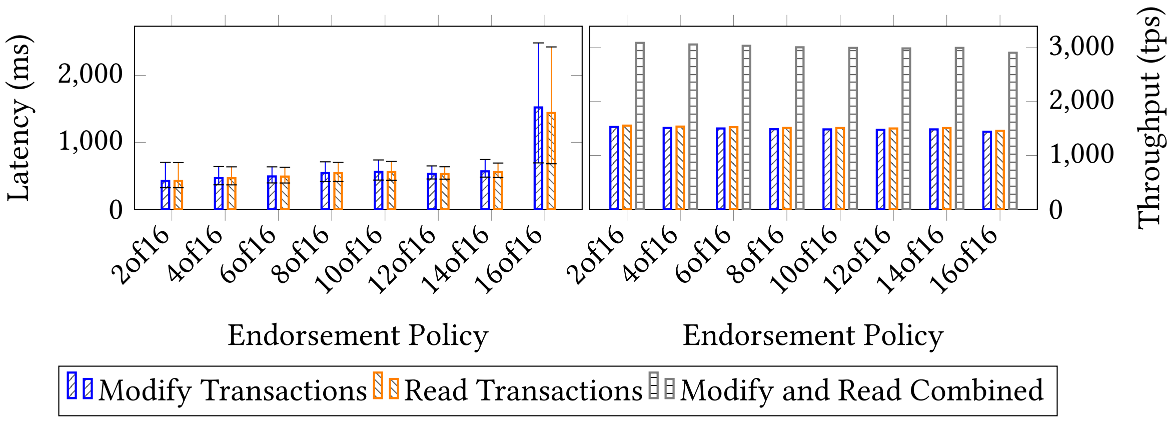


Figure 5.7.6: Effect of different endorsement policies.

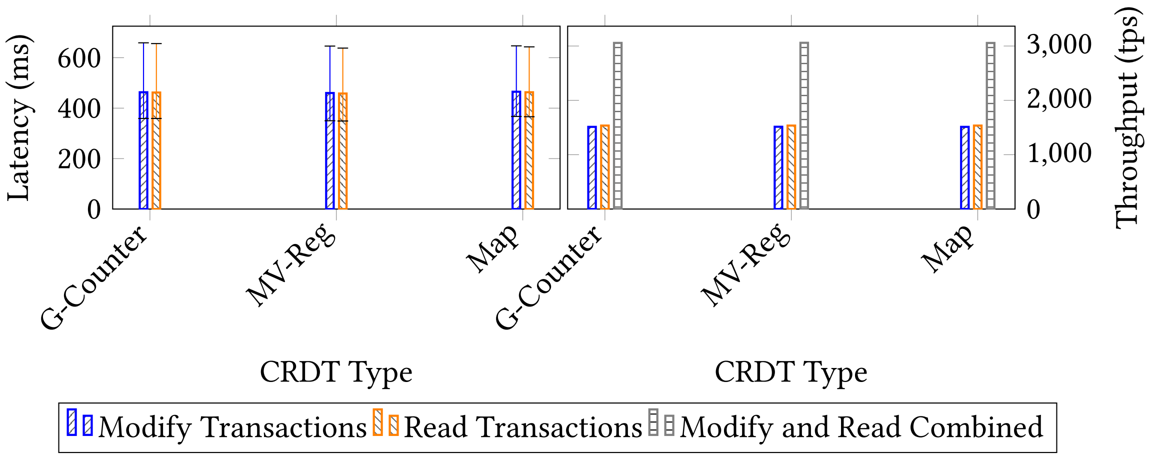


Figure 5.7.7: Effect of different CRDT types.

shown in Figure 5.7.8, we observed that the latency and throughput were unaffected by varying the percentage of read- and modify-transactions.

Effect of Increasing Gossip Ratio – We also studied the effect of increasing the Gossip Ratio by increasing the number of organizations. As shown in Figure 5.7.9, we do not observe a significant change in latency and throughput for an increasing gossip ratio either. The reason is that the bandwidth is not a bottleneck, and most of the gossiped transactions have been sent by the clients and are previously committed by the organizations. Hence, the gossiped transactions must not be committed.

Impact of Byzantine Organizations – We studied the effects of organizations' Byzan-

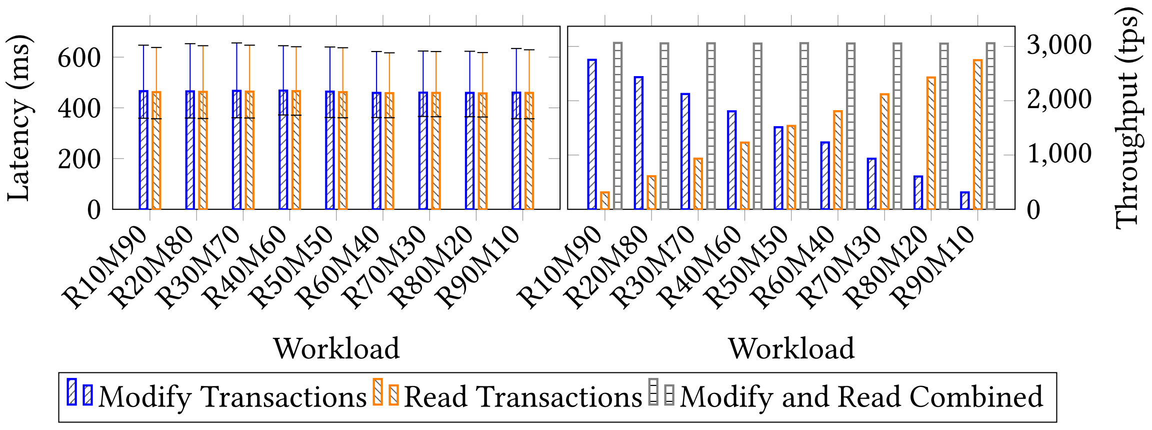


Figure 5.7.8: Impact of different percentage of read- and modify-transactions.

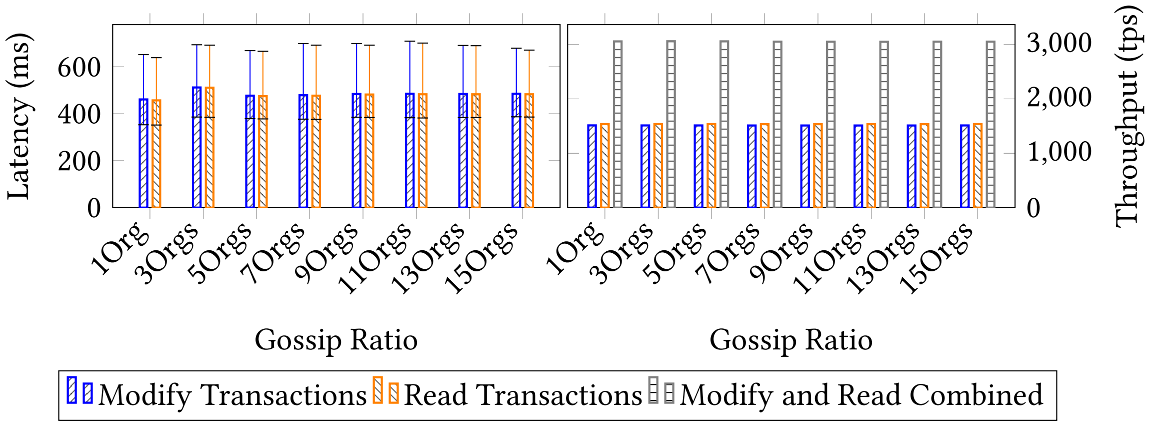


Figure 5.7.9: Effect of increasing gossip ratio.

tine failures. As shown in Figure 5.7.10, three randomly selected organizations behaved arbitrarily for a specific period during the experiment. The Byzantine organizations either randomly avoid responding to clients' proposals or transactions 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. We observed that the throughput decreases with every Byzantine failure. However, the latency is not affected. The reason for the decreasing throughput is that clients cannot collect an adequate number of endorsements due to the Byzantine organization not responding and the signature validation failure caused by the wrongly endorsed proposals.

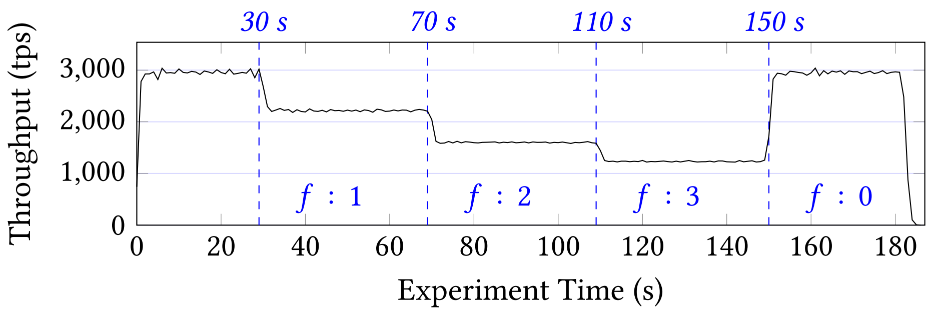


Figure 5.7.10: Throughput across experiments with Byzantine organizations.

Since clients can observe organizations that wrongly endorse or do not respond while other organizations respond with lower latency, they can avoid Byzantine organizations. To demonstrate this, we ran experiments where the clients avoided the Byzantine organization and changed to another randomly selected organization. As shown in Figure 5.7.11, the throughput returns to its pre-failure value immediately after clients avoid the Byzantine organizations, as shown by the solid green lines.

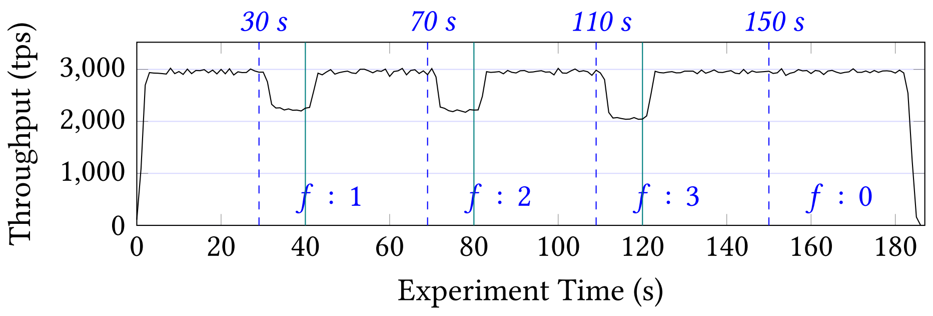


Figure 5.7.11: Experiments with clients avoiding Byzantine organizations.

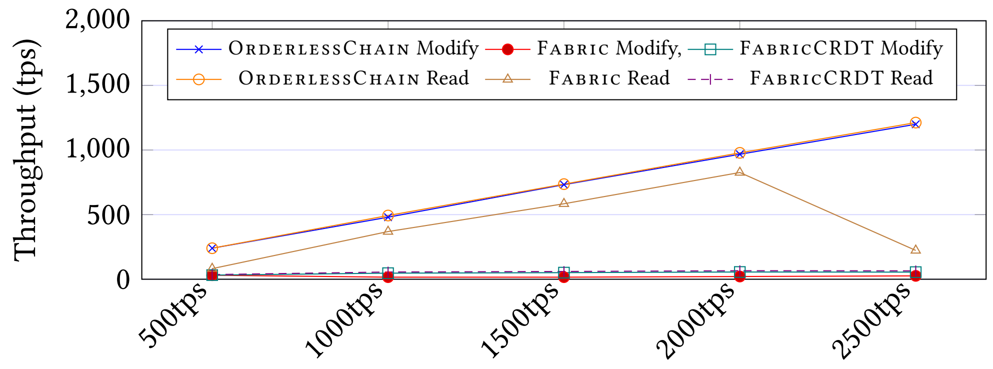
5.7.5 Experimental Results for the Voting and Auction Applications

We executed the voting and auction applications to compare our coordination-free approach of OrderlessChain to the coordination-based protocols of Fabric and Fabric. These experiments are conducted with eight organizations, the $EP : \{4\ of\ 8\}$ endorsement policy, and the uniform workload of 50 percent read-transactions and 50 percent write-transactions. Furthermore, no Byzantine organizations exist, and the gossip ratio for OrderlessChain is set to one.

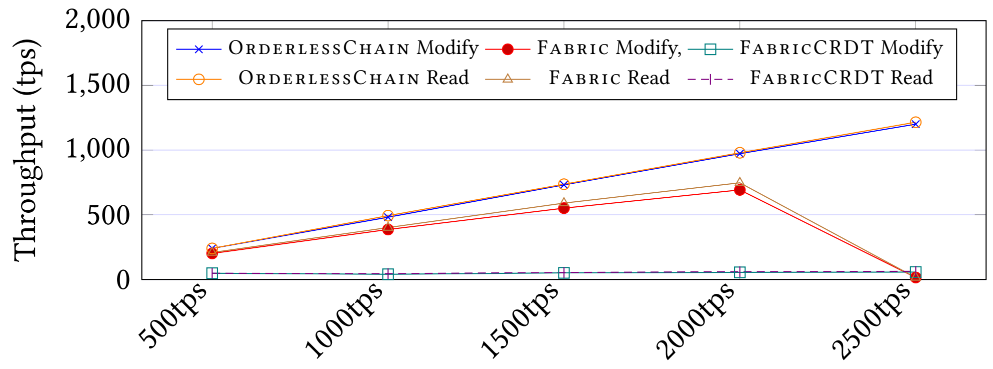
We increased the transaction arrival rate from 500 tps to 2500 tps for the voting and auction applications on the three systems. For FabricCRDT, we observed that latency significantly increases for a higher transaction arrival rate due to its state-based CRDT implementation, so we limited the transaction latency for FabricCRDT to 240 seconds, after which they are timed out and not considered for throughput and latency determination.

The observed throughput of the voting and auction applications are shown in Figure 5.7.12. The average, 1st and 99th percentiles transaction are demonstrated in Figure 5.7.13. As shown in Figure 5.7.12, we observe that OrderlessChain demonstrates a higher throughput of modify- and read-transactions for both applications. Although Fabric's read-transactions reach 2000 tsp, the throughput significantly drops at 2500 tsp arrival rates for both applications since the coordination-based ordering service becomes a bottleneck. Furthermore, we observe that many modify-transactions of the voting application fail due to the MVCC validation explaining its low throughput. This finding confirms the finding of Chacko et al. regarding the low throughput of highly concurrent applications on Fabric [8]. Although we used caching for the CRDT approach in FabricCRDT, its approach still is a bottleneck, and we observe low throughput and high latency for read- and modify-transactions of voting and auction applications.

As shown in Figure 5.7.13, OrderlessChain's latency remains constant under increasing arrival rates. Fabric's latency significantly increases for higher arrival rates for both applications. As mentioned, the reason is that Fabric's central ordering service is a bottleneck. The increased latency causes more transactions to fail due to MVCC validation, which explains the significant throughput decrease for Fabric. FabricCRDT demonstrates irregular latency patterns as timed-out transactions are not considered.



(a) Transaction Arrival Rate for the Voting Application (tps)



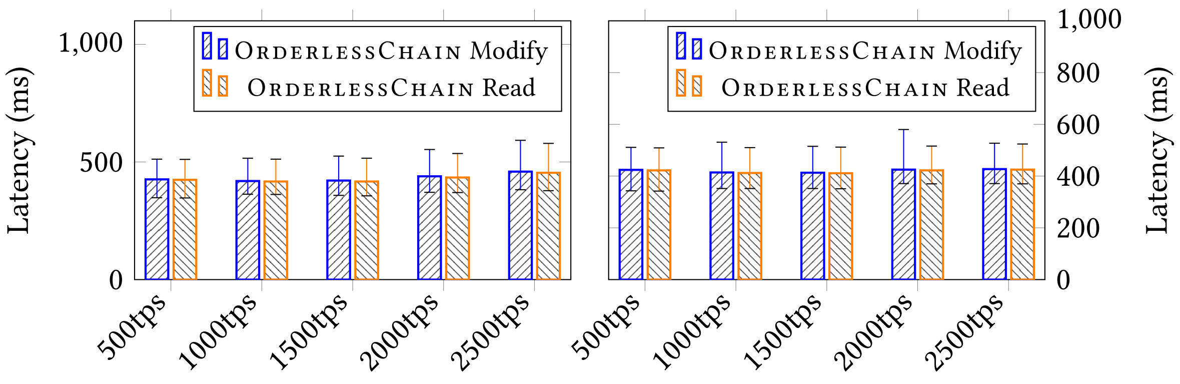
(b) Transaction Arrival Rate for the Auction Application (tps)

Figure 5.7.12: Throughput of voting and auction applications.

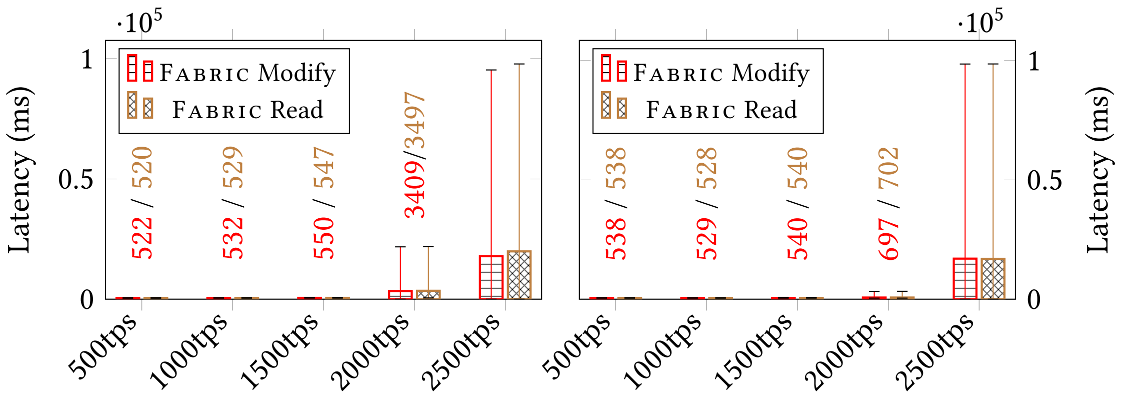
5.7.6 Discussion

We initiated this work to study to which extent we can reduce the coordination in a trustless environment while preserving the application's invariant conditions and offering Byzantine Fault Tolerance. We introduced a coordination-free protocol for Byzantine Fault Tolerance and proved its feasibility. However, as demonstrated, preserving invariant conditions without coordination depends on the application running on the permissioned blockchains. Suppose the invariant conditions can be modeled as I-confluent invariants. In that case, coordination is unnecessary, and our approach can significantly improve throughput and latency compared to coordination-based approaches.

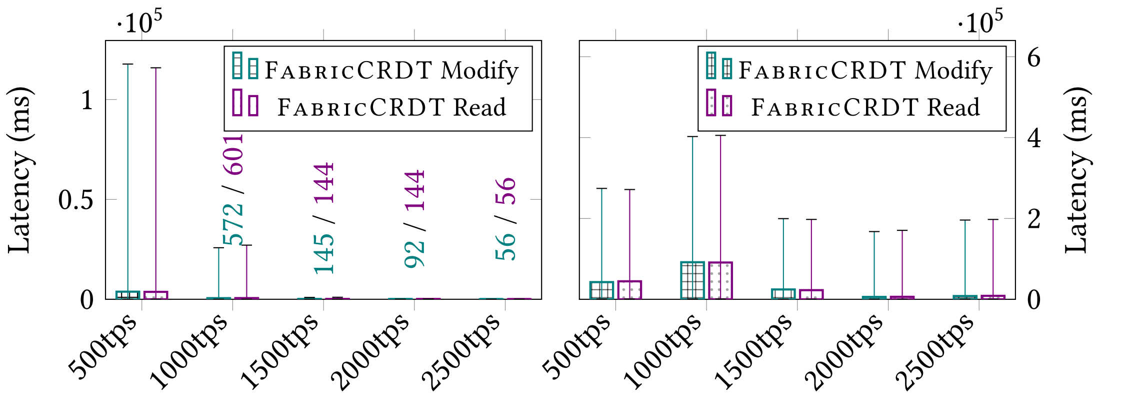
However, coordination to order the transactions is required for applications with non-I-confluent invariants. For example, suppose we require an invariant condition for



(a) Arrival Rate for Vote App (tps) (b) Arrival Rate for Auction App (tps)



(c) Arrival Rate for Vote App (tps) (d) Arrival Rate for Auction App (tps)



(e) Arrival Rate for Vote App (tps) (f) Arrival Rate for Auction App (tps)

Figure 5.7.13: Latency of voting and auction applications.

specifying a deadline for the end of an election or an auction, and the transactions that arrive after this deadline must be rejected. This is a non-I-confluent invariant and requires coordination to be preserved. One approach for enabling OrderlessChain to preserve such invariants is extending our system with coordination-based protocols such as the protocol used by Fabric and enabling this protocol when required. For example, given that the end of an election or an auction specifies only a short period of time of the whole time these events are running, which can be up to a few hours or days, the coordination-based protocol can be enabled when we are close to the end of an election or auction. Otherwise, we use our coordination-free approach during the long period when the non-I-confluent invariant is not relevant and benefit from the increased throughput and latency.

There exist several I-confluent CRDT-based use cases [70, 169, 172, 173, 174, 194, 196, 197, 198, 199, 200], from key-value stores to multi-user collaborative environments, which can be implemented on OrderlessChain, benefiting from the trust and scalability our system offers. Furthermore, there exist CRDT, and I-confluent automation tools such as *Katara* [201] and *Lucy* [126], which decrease the development effort for modeling CRDT applications and identifying the I-confluent invariants. Katara offers a solution for automatically synthesizing CRDTs from sequential non-CRDT implementations. In other words, Katara provides a solution for studying the logic of applications and proposing CRDTs for converting the application into a CRDT-enabled application. Lucy also provides an environment for determining whether invariant conditions are I-confluent. By taking advantage of such tools, developers can more easily develop applications that can be developed on OrderlessChain and eliminate the unnecessary coordination that limits the scalability.

We implemented additional use cases on OrderlessChain 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 and a private and distributed federated learning system by extending OrderlessChain with customized CRDTs, which we discuss in the next chapter. The development of these applications on OrderlessChain was relatively straightforward.

5.8 Summary

In this work, we presented OrderlessChain, a coordination-free permissioned blockchain without total global order of transactions. OrderlessChain uses the permissioned property of permissioned blockchains to offer a BFT coordination-free protocol. Furthermore, OrderlessChain takes advantage of CRDTs to enable the development of safe and live I-confluent applications in a Byzantine environment.

We extensively evaluated OrderlessChain to demonstrate our system's scalability and Byzantine Fault Tolerance. We also executed several experiments, comparing the performance of OrderlessChain's coordination-free protocol to the coordination-based protocols of Fabric and FabricCRDT. We demonstrated a significant improvement in the scalability of our coordination-free design over coordination-based alternatives for I-confluent applications.

6

Extended Applications of OrderlessChain

The following chapter introduces two systems as two new use cases of OrderlessChain to demonstrate the applicability of our system to other domains. In Section 6.1, we introduce OrderlessFL, a permissioned blockchain-based *federated learning (FL)* system, which uses flCRDT, a novel CRDT for asynchronous and concurrent aggregation of FL model updates. The other system proposed in Section 6.2 is OrderlessFile, a permissioned blockchain-based file storage system. OrderlessFile uses FileCRDT, a new CRDT for sharding data and concurrently replicating and storing files.

The content of this chapter is based on the papers published on OrderlessFL , and OrderlessFile [62, 63].

6.1 OrderlessFL: A Blockchain-based Federated Learning System

Organizations across various industries and sectors, from health care to IoT infrastructures, produce a large amount of data that can be used for improving *Machine Learning*

(ML) models and potentially create more generalized and accurate ML models [150, 151]. However, the raw data can often not be shared among organizations due to privacy issues and various regulations limiting data access. To make use of the data locked within data silos, FL has gained popularity as a solution to privacy-preserving ML [137, 150, 151, 202].

FL enables the users to collectively collaborate to train global ML models without transferring and sharing their private data with other participants. In traditional ML approaches, a central entity often retrieves and stores large amounts of data from various organizations and trains the model on the central entity side.

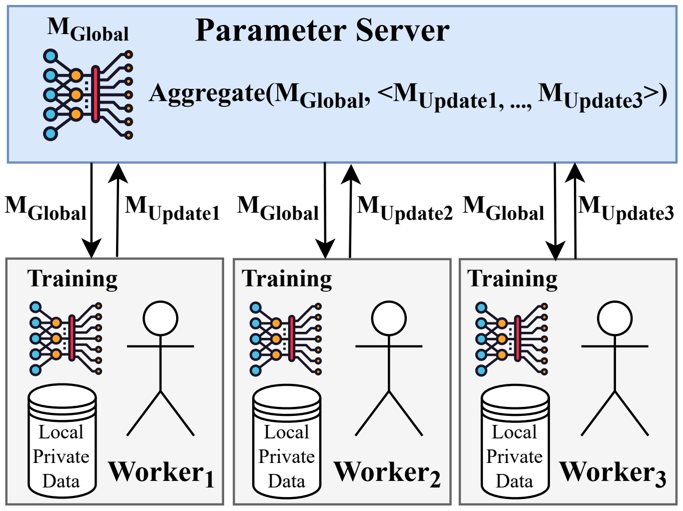


Figure 6.1.1: A conventional synchronous federated learning system.

A conventional synchronous FL system, as shown in Figure 6.1.1, consists of several workers and a central *Parameter Server (PS)* [141]. PS manages a global ML model, receives model updates from workers, and aggregates the model updates with the global model using an FL aggregation protocol such as *FedAvg* [56, 203]. The PS selects the workers participating in the training round based on some eligibility prerequisites. The workers fetch the global model from the PS and train it locally based on their local data, using a training algorithm such as the *Stochastic Gradient Descent* [204, 205]. Once the training is over, the model updates are sent to PS to be aggregated and integrated with the global model.

FL offers a more private distributed ML approach so that no data is shared with other participants. For enhancing privacy, the conventional approach is often integrated with other privacy-preserving solutions, such as *Differential Privacy* and *Homomorphic Encryption* [108, 152, 206, 207]. However, the central and potentially Byzantine PS can jeopardize the system in several ways [145, 150, 151]. For example, the model updates can be intentionally tampered with or not be aggregated with the global model. In order to offer a BFT solution where the Byzantine workers and PS cannot jeopardize the system, several blockchain-based solutions have been proposed. However, most of the existing blockchain-based FL solutions use PoW-based solutions, which suffer from the PoW's scalability limitations [150, 151]. Also, the non-PoW-based blockchain solutions rely on coordination-based solutions with limited scalability due to the coordination to reach a consensus. Furthermore, due to the blockchain's storage limitation, they depend on third-party off-chain storage solutions such as *InterPlanetary File System (IPFS)* [153]. However, integrating third-party solutions into any system may introduce new security and privacy threats.

Besides the potential risks of Byzantine PS, the performance of synchronous FL is affected by slow workers or *stragglers* [202]. During the execution of an asynchronous FL protocol, the PS must wait for the end of the training on the stragglers before aggregating the model updates. As the number of workers increases, the negative effect of stragglers on the system's performance increases and causes a scalability bottleneck. In contrast, asynchronous FL protocols exist, where faster workers' model updates are more frequently integrated with the global model. Although this approach mitigates the effect of stragglers of synchronous FL protocols, it introduces the *gradient staleness* problem, where the workers train on an outdated global model and prevents the convergence of the global model [154, 155].

Training a distributed ML model shares similar problems with the concurrent reads and writes in distributed computing. One technique for addressing the problems is CRDTs. By offering a CRDT-enabled FL system, where the ML models are created using CRDTs, we enable the concurrent and asynchronous aggregation of FL models. Furthermore, we can use the logical clock-based timestamped CRDT operations to measure the staleness of model updates to limit and mitigate the gradient staleness problem.

In order to address these issues, we adapted and extended OrderlessChain to OrderlessFL and introduced flCRDT, a novel CRDT enabling the concurrent and asynchronous aggregation of models in FL. FLCRDT uses the properties of CRDTs to mitigate the gradient staleness problem. OrderlessFL is a permissioned blockchain-based FL system using FLCRDT and offers a safe protocol for storing and aggregating models where Byzantine actors cannot tamper with the models. Furthermore, to avoid the potential risks of third-party solutions, OrderlessFL offers a complete solution for storing and aggregating models on-chain.

6.1.1 Architecture and FL Protocol

OrderlessFL is an SEC asynchronous permissioned blockchain-based FL system. Since OrderlessFL is an extension of OrderlessChain, the system model of OrderlessChain discussed in the previous chapter also applies to OrderlessFL.

An OrderlessFL network consists of a set of organizations $\{O_1, ..., O_n\}$ and a set of workers $\{W_1, ..., W_r\}$. Organizations store and distribute global ML models and receive and aggregate updates, and are equivalent to PS in an FL protocol. The workers receive a global model from an organization and train the model using its local data. The system uses the following *training-aggregation* two-phase asynchronous FL protocol, shown in Figure 6.1.2:

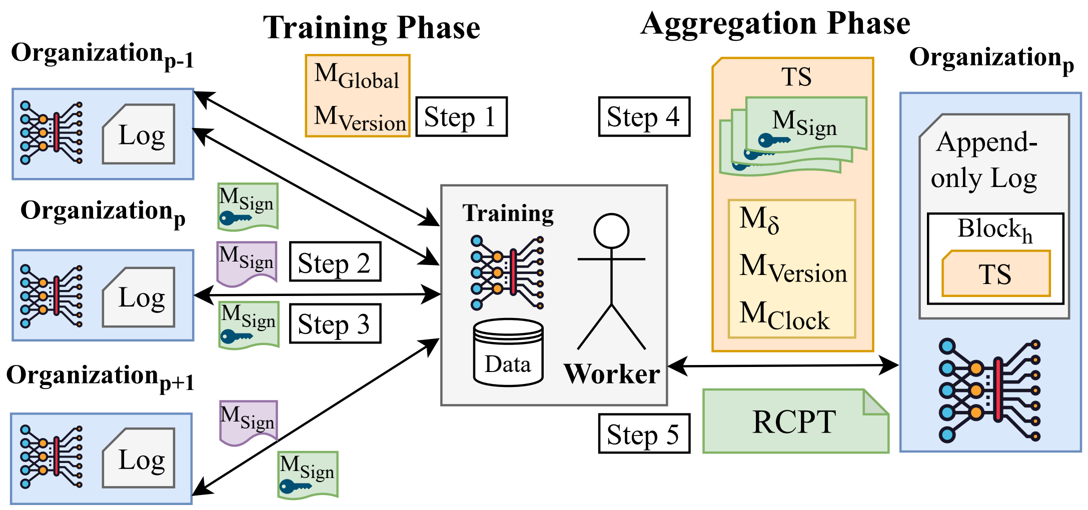


Figure 6.1.2: Workflow for training a model on OrderlessFL.

Phase 1 / Training Phase – The worker first contacts any organization to receive $< M_{Global}, M_{Version} >$, the weights of the global model and the model's current version (Step 1 in Figure 6.1.2). The model architecture and learning algorithm are global system configuration settings. The worker initializes a model with M_{Global} and trains the model using its local data. Afterward, $M_{\delta} = M_{Local} - M_{Global}$, the weight difference between the locally trained and global models, is calculated. The worker also keeps track of a logical clock M_{Clock}, incrementing it with every update. Then, the worker creates a model signature $M_{Sign} = Hash(< M_{\delta}, M_{Version}, M_{Clock} >)$ and sends it to the organizations (Step 2). The organization signs M_{Sign} with its private key based on public key cryptography and sends the signed response, also known as an endorsement, to the worker (Step 3).

Phase 2 / Aggregation Phase – The worker waits to receive the endorsements from organizations and verifies the signatures' validity. If every endorsement is valid, a transaction TS that contains the endorsements and $< M_{\delta}, M_{Version}, M_{Clock} >$ is created and sent to the organizations (Step 4). The organization appends TS to its append-only hash-chain log by creating a block $Block_h =< Hash(TS), Hash(Block_{h-1}) >$, which contains TS and the hash of the previous block, and appends the block to the log. The organization verifies whether every endorsement in TS is valid. This ensures that every organization has received identical data from the worker and that the client did not tamper with the endorsements. Hence Byzantine malicious behavior is prevented. For valid transactions, a hashed and signed receipt $RCPT = HashSign(Block_h, Valid)$ is sent to the worker (Step 5). A signed rejection is sent otherwise. As the receipt contains the block's hash value which is also dependent on the previous blocks, any Byzantine modification of the organization invalidates the receipts of TS and other transactions. Finally, the organization aggregates M_{δ} of the valid transaction into the global model, explained below.

We open-sourced the code of the system on GitHub [58].

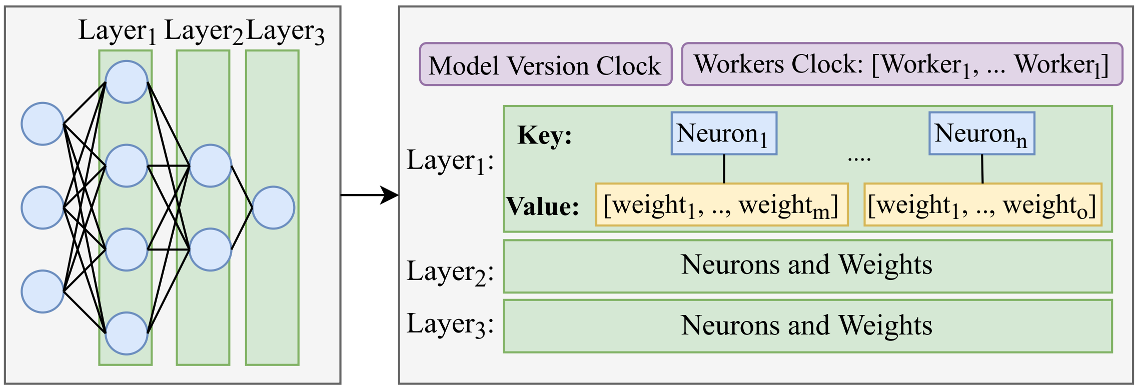


Figure 6.1.3: Modeling a Deep Neural Network model with FLCRDT.

6.1.2 FLCRDT: A Federated Learning CRDT

We introduce FLCRDT, a nested CRDT for modeling a wide range of ML models. In this work, we only discuss modeling a *Deep Neural Network (DNN)* [208, 209], as shown in Figure 6.1.3. However, FLCRDT can be extended to other ML models including but not limited to *Regression Models* and *Decision Trees* [210].

A DNN model consists of several layers, with every layer consisting of several neurons with their weights [208, 209]. For accommodating a multi-layer data structure, FLCRDT is a nested data structure. The layers and neurons are modeled as nested data structures in an instance of FLCRDT. The root of FLCRDT is a map consisting of key-value pairs, where the key is a unique identifier of the layer, and the value consists of nested maps containing other layers or the neurons' weights. FLCRDT also contains two logical clocks: a model clock and a workers' vector clock. The model clock stores the model version and increments it with every aggregated update. The workers' vector clock stores the observed clocks of workers.

Algorithm 4 displays our approach used by organizations to aggregate M_δ with the global model. The organization proceeds to aggregate M_δ if it has observed all previous updates sent by the worker (Line 2). Otherwise, the update is queued (Line 10). Before updating the model, the model version and the worker's clock are incremented (Lines 3 and 4). For mitigating the gradient staleness, a *staleness penalty* is calculated based on the current model version of M_{Global} and the global model used by the worker (Line 5).

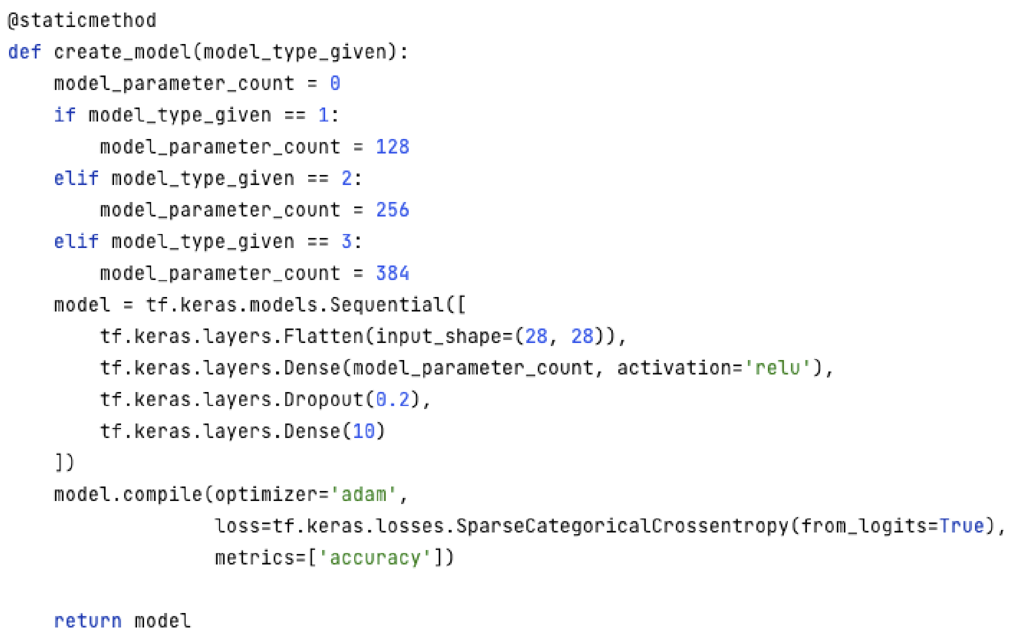
Algorithm 4: Aggregating the FL model update with the global model on OrderlessFL.

```
1  AggregateToGlobalModel ($M_{Global}, M_\delta, M_{Version}, M_{Clock}, WorkerID$)
       input  : $M_{Global}$, the global model at the organization.
       input  : $M_\delta$, a model update from worker.
       input  : $M_{Version}$, the global model's version number used by the worker.
       input  : $M_{Clock}$, the worker's clock.
       input  : $WorkerID$, the identifier of the the worker.
2      if $M_{Clock} - M_{Global}.WorkerClocks[WorkerID] == 1$ then
3          $M_{Global}.Version += 1$
4          $M_{Global}.WorkerClocks[WorkerID] += 1$
5          $StalenessPenalty = (M_{Global}.Version - M_{Version})^{-1}$
6          $UpdateRate = StalenessPenalty * M_{Global}.WorkerClocks.Length^{-1}$
7          foreach $layer$ in $MW_\delta.Layers$ do
8              $M_{Global}.Layers[layer] += M_\delta.Layers[layer] * UpdateRate$
9      else
10         $M_{Global}.EnqueueUpdate(< M_\delta, M_{Version}, M_{Clock}, WorkerID >)$
11     $M_{Global}.ProcessQueuedUpdates()$
```

As the difference between the versions of the two global models increases, the effect of the staleness penalty is more significant. Hence, the model update's staleness implies the model update's significance in being integrated with the global model.

Furthermore, we calculate an *update rate* based on the staleness penalty and the number of workers in the workers' vector clock (Line 6). We do so to calculate the average weights of the model updates concerning the workers in the system. Then, we iterate over the layers and aggregate each layer of the model update with the global model using the update rate (Lines 7 and 8). Finally, the queued updates are processed following the same procedure explained in Algorithm 4 (Line 11).

In order to determine whether OrderlessFL is I-confluence, we specify its invariant condition: *only the client's latest model update is applied.* flCRDT tracks the model version number and the worker's observed vector clock, and the model updates also contain the version number of the global model used for training. Since Algorithm 4 ensures that the worker's model update is only aggregated when the previous updates of the worker are aggregated, OrderlessFL can preserve the invariant condition. Hence, it is I-conflict concerning the recognized invariant.



```
@staticmethod
def create_model(model_type_given):
    model_parameter_count = 0
    if model_type_given == 1:
        model_parameter_count = 128
    elif model_type_given == 2:
        model_parameter_count = 256
    elif model_type_given == 3:
        model_parameter_count = 384
    model = tf.keras.models.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(model_parameter_count, activation='relu'),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10)
    ])
    model.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])

    return model
```

Figure 6.1.4: Structure of the evaluated DNN models on OrderlessFL.

6.1.3 Evaluation

For evaluating the performance of OrderlessFL, we deployed eight organizations. We evaluated the protocol for aggregating model updates and accuracy for training DNN models.

We trained the models using *TensorFlow 2/Keras* [211]. We trained three convolutional DNN models with 102, 203, and 305 thousand trainable parameters. Each model contains four sequential layers as shown in Figure 6.1.4, which is adapted from a straightforward exiting model [212]: one *Flatten Layer*, one *Dense Layer* with *ReLU Activation Function*, one *Dropout Layer*, and finally, one *Dense Layer* for classification. We used the *Adam optimizer* and *crossentropy loss function*. The models are trained on the *MNIST dataset* [213].

To evaluate the performance of our approach for aggregating model updates, we gradually increased the transactions arrival rate from 50 tps to 400 tps. Each transaction submits a model update. Each experiment is executed at least three times for 60 seconds, and the experiments' results are averaged. The average latency to throughput is shown in Figure 6.1.5. As we aimed to evaluate the scalability of our approach concerning model aggregation, the latency does not include the time required to retrieve the model from an organization and train the model (Step 1 in Figure 6.1.2 is excluded). The latency only reflects the time from the beginning of step 2 till the end of step 5 of the protocol.

We observe that the latency gradually increases for the higher arrival rates. We also observe the increased latency for larger models. The reason is that deserializing model updates and aggregating model updates with the global model are CPU-bounded tasks, which causes CPU saturation for the higher workload on the organizations causing the transactions to be queued before being processed.

We also measured the accuracy of the aggregated global model with 102 thousand trainable parameters for predicting the correct labels based on the test subset of the MNIST dataset and observed up to 85% accuracy.

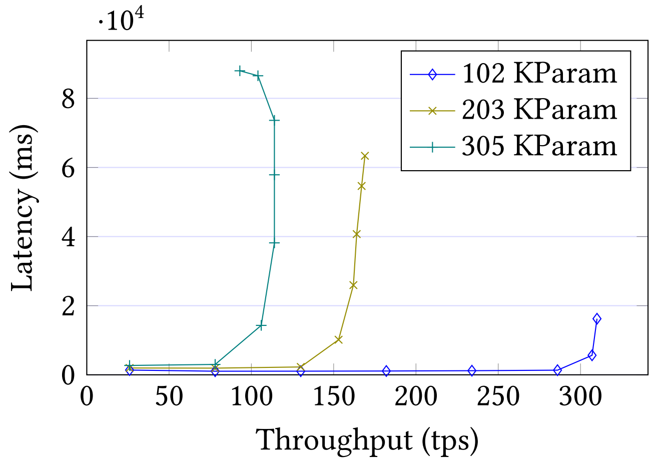


Figure 6.1.5: Latency to throughput for training three DNN models.

6.2 ORDERLESSFILE: A Blockchain for File Storage

Cloud storage services are an integral part of cloud ecosystems [161, 162]. Although cloud storage offers affordable and available *Storage-as-a-Service*, most cloud providers lack transparency on the privacy and security of stored files [161]. Hence, clients must trust the service-level agreements and be confident that providers do not tamper with the stored data and store them safely. Several blockchain-based distributed and decentralized file storage systems have been proposed to address these issues [160, 161, 162]. Although blockchain-based file storage systems are more secure and private than conventional cloud providers, several existing systems are based on PoW-based solutions such as Bitcoin and Ethereum [163, 164] and suffer from the common scalability issues of PoW-based protocols. Furthermore, due to the storage limitations on blockchains, existing solutions use various off-chain systems such as IPFS, which might present other security and privacy issues.

To address the scalability issues and limitations of stored file size, we extended OrderlessChain with FileCRDT, introducing OrderlessFile. OrderlessFile is a private and distributed permissioned blockchain-based file storage. It offers a scalable and safe protocol for replicating and storing encrypted files on-chain where Byzantine participants can not tamper with data and violate its integrity. FileCRDT enables the clients to split files into shards that are stored and replicated on OrderlessFile. Like conventional file storage, OrderlessFile can store various types of data, from DNS records to extensive IoT and machine learning datasets.

6.2.1 FileCRDT: A CRDT for File Storage

We introduce FileCRDT for sharding and storing files. The structure of FileCRDT is shown in Figure 6.2.1. Each file stored on OrderlessFile has a unique identifier and is versioned based on a logical clock tracked by the client. The logical clock indicates the file's version number, and the client increments it with every new version of the file sent to be uploaded.

FileCRDT is a nested data structure in the format of key-value pairs. Depending on the file size and OrderlessFile's configuration settings, the file is split into shards, where each shard is stored as a key-value pair in an instance of FileCRDT. The key is the shard's identifier which is unique per file. The value is another key-value data structure, where the key is the file's version, and the value is a multi-value register [54] containing the shard data. The purpose of keeping track of the versions is to resolve the conflicting updates sent by the client and prevent data corruption as the shards are gradually transmitted to be stored.

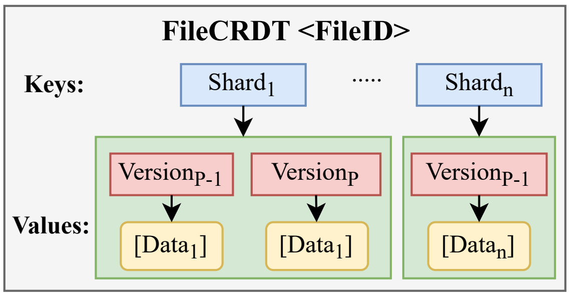


Figure 6.2.1: Structure of FileCRDT for sharding and storing a file.

6.2.2 Architecture and Protocol

The system model of OrderlessFile is similar to the system model of OrderlessChain since OrderlessFile is an extension of OrderlessChain. OrderlessFile's network consists of several organizations and clients. The organizations are responsible for receiving and storing client files in their ledger. The clients can communicate with every non-failed organization and upload and download files to the system.

For uploading files, clients follow a two-phase *endorsement-storage* protocol, as shown in Figure 6.2.2:

Phase 1 / Endorsement Phase – Depending on the file size and the maximum allowed shard size, the client first splits the file into shards. The allowed shard size is a global system configuration setting, which is set by the consortium of the organizations. For every shard, a shard signature $SS_m = Hash(< Encrypt(Shard_m), Version_p >)$ is created based on the shard's encrypted data and the file's current version. SS_m is sent to the organizations (Step 1 in Figure 6.2.2). The client sends SS_m to all organizations or a subset of them, depending on the replication factor intended for storing the file. The organization signs SS_m with its private key based on public key cryptography and sends

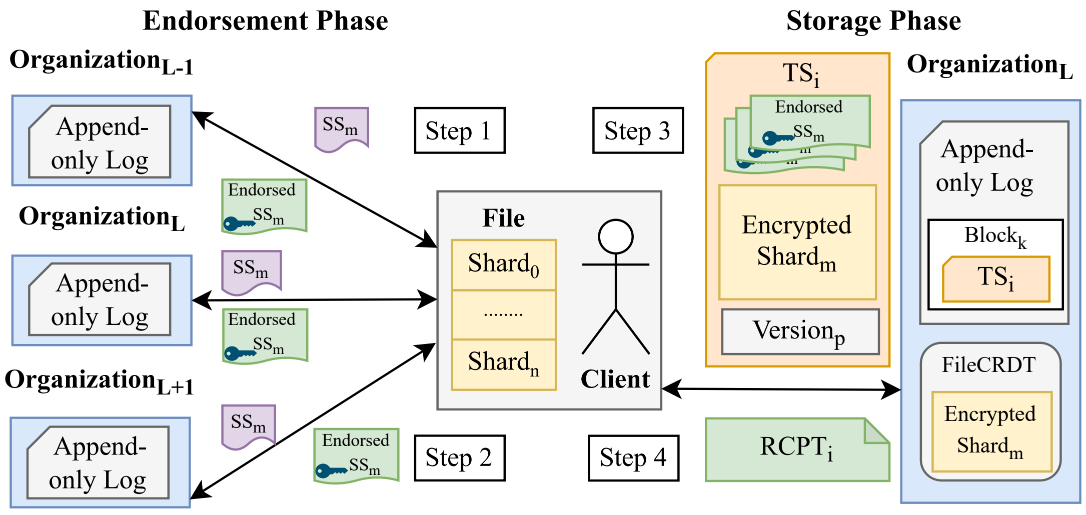


Figure 6.2.2: Workflow for uploading files on OrderlessFile.

the signed response, also called an endorsement, to the client (Step 2).

Phase 2 / Storage Phase – The client waits to receive the endorsements from organizations and verifies their validity by using the organization's public key to prevent Byzantine behavior of the organizations. If every endorsement is valid, the client creates a transaction TS_i that contains the endorsements, the encrypted shard data, the shard's identifier, and the file's version. The client sends the transaction to the same organizations (Step 3). The organization appends the transaction to an append-only hash-chain log. The organization first creates a block $Block_k =< Hash(TS_i), Hash(Block_{k-1}) >$, which contains the hash value of TS_i and the hash of the previous block and appends the block to the log.

Afterward, the organization validates every endorsement in TS_i to ensure that the organizations endorsed the identical SS_m. If the transaction is valid, a receipt $RCPT_i = HashSign(Block_k, Valid)$ is created and sent to the client (Step 4). The organization sends a signed rejection containing the block's hash for an invalid transaction. Since the created block contains the hash of the transaction and the previous block, the Byzantine organization cannot tamper with the content of the transaction without invalidating the receipt of TS_i and the previous transactions. If TS_i is valid, the organization updates the instance of FileCRDT that contains the file.

The shard in FileCRDT is modified based on the shard's identifier and the file's version. If a conflicting shard with an identical version exists, the shard data is added to the multi-value register, and the client decides which data to use. Finally, the organization iterates through the shards to verify whether every shard has a $Version_p$ key. If $Version_p$ key exits in every shard in FileCRDT instance, the key-value pairs with the key $Version_{p-1}$ are removed to free up disk space.

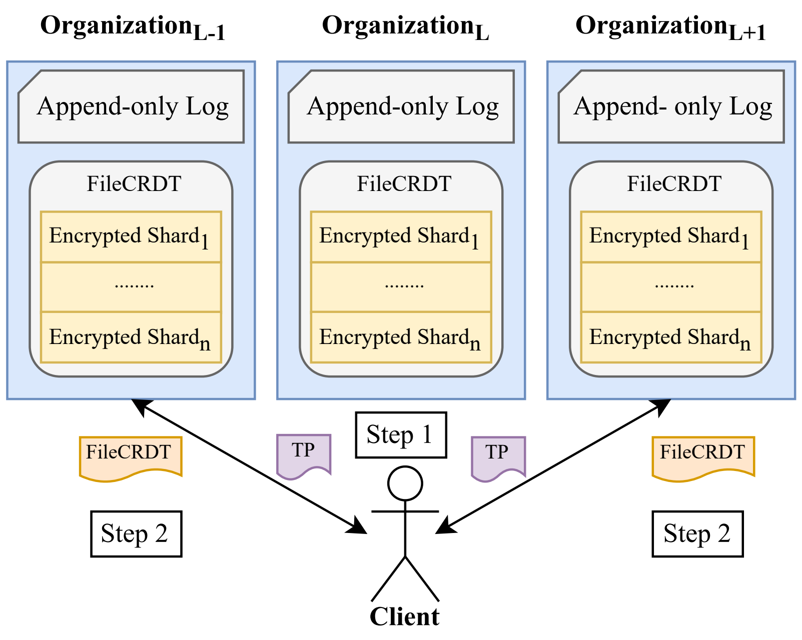


Figure 6.2.3: Workflow for downloading files on OrderlessFile.

For downloading a file, the client follows the procedure shown in Figure 6.2.3. The client sends a transaction proposal of TP containing the file's identifier to any organizations in the network (Step 1 in Figure 6.2.3). If the organization stores a replica of the file in an instance of FileCRDT, it serializes FileCRDT into binary and sends the data to the client (Step 2).

Finally, the client deserializes FileCRDT instance, decrypts the shards, and verifies whether the client's signature on the encrypted shard is valid. The client may send several requests to every organization that stores a replica to compare encrypted shard data from one organization to another if the client suspects any Byzantine behavior.

For OrderlessFile, we consider one invariant condition: *The latest version of the file is stored.* FileCRDT tracks the version of the file in the shard's key-value pairs, and OrderlessFile only removes the older version of the shard's data once the shards of the newer version are entirely uploaded. Therefore, OrderlessFile ensures that only the latest version of the files is stored, and the data of the latest version overwrite the previous versions. It preserves the system's invariant conditions. Hence, OrderlessFile is I-confluence concerning the invariant.

We open-sourced the code on GitHub [59].

6.2.3 Evaluation

For evaluating OrderlessFile, we deployed a network of 16 organizations with 1000 clients downloading and uploading files concurrently. The workload consists of half download and half upload transactions uniformly distributed during the experiments. Each experiment is executed at least three times for 60 seconds, and the experiments' results are averaged. Each client owns one file consisting of ten shards with a replication factor of two. We gradually increased the maximum shard size from 25KB to 100KB.

Figure 6.2.4 show the average latency to the throughput of upload and download transactions. For uploading transactions, latency remains constant for smaller shard sizes. However, the latency increases for larger shard sizes as they result in more significant transactions that require more time to be transmitted. Furthermore, larger shard sizes have a more significant overheard on the organization for deserializing the transactions and writing the data on the disk, which increases the average latency for transactions with larger shard sizes. The latency for download requests increases as throughput and the shard size increase since larger files require more time to be transmitted.

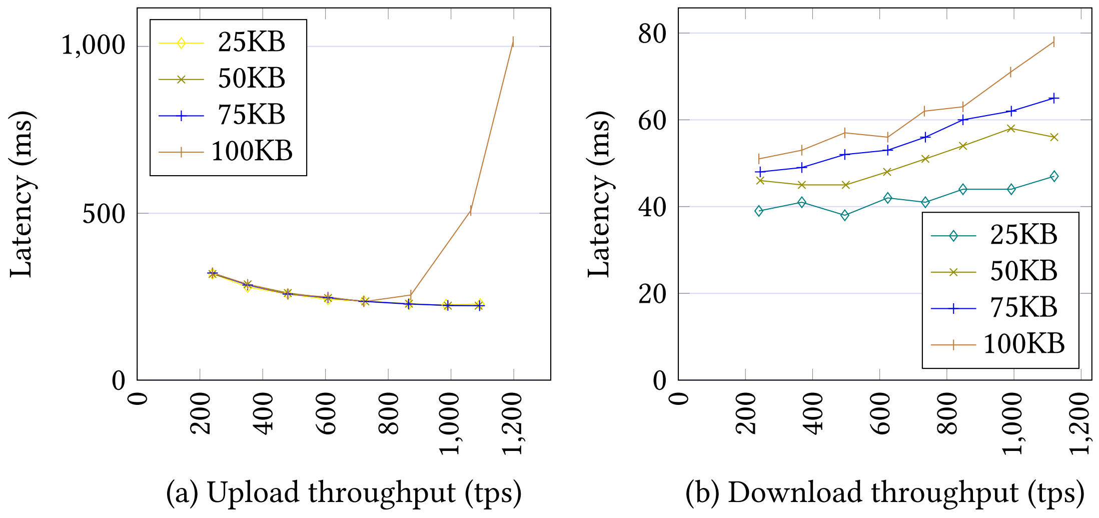


Figure 6.2.4: Latency to the throughput of download and upload transactions.

6.3 Summary

In this chapter, we introduced two use cases of OrderlessChain to demonstrate the applicability of our system in other domains. First, we introduced OrderlessFL, an extension of OrderlessChain for executing FL protocols. OrderlessFL uses flCRDT, a novel CRDT for concurrent and asynchronous aggregation of FL models where the gradient staleness problem is mitigated.

The other use case is extending OrderlessChain to the cloud provider's domains. We introduced OrderlessFile, a safe and secure blockchain-based file storage as an alternative to the non-transparent storage services provided by the cloud providers. OrderlessFile uses FileCRDT, a customized CRDT for sharding and secretly storing and replicating data on OrderlessFile.

Our evaluation of both systems demonstrates the applicability of OrderlessChain to other domains. Also, the development effort required for extending OrderlessChain to these systems was straightforward, indicating the practicality of our system.

7 Conclusions

In this dissertation, we provided two CRDT-based approaches for reducing coordination in permissioned blockchains and improving their scalability. First, we offered a CRDT-based solution that uses the CRDTs' automatic conflict resolution mechanism to eliminate the failures of concurrent transactions in Fabric. Second, we introduced OrderlessChain, a scalable permissioned blockchain for the safe and live execution of I-confluent applications in a Byzantine coordination-free environment. We also demonstrated the applicability of OrderlessChain to several domains, including FL and distributed file storage systems. In the following chapter, we summarize the concluding remarks of this work and offer an outlook on future work.

7.1 Summary

FabricCRDT offers a novel CRDT-based approach to address the failure of concurrent transactions in Fabric. The high latency induced by Fabric's three-phase protocol and its MVCC validation mechanism causes the failure of a high percentage of dependent concurrent transactions. Our proposed solution detects conflicting transactions within one block automatically and applies CRDT-based techniques to resolve the conflicts and merge the value of the transactions without resulting in data loss, corruption, or inconsistency.

FabricCRDT's proposed solution is CRDT-agnostic, and the protocol is independent of the CRDT used in chaincodes. However, to enable developers to model and develop applications based on various CRDTs, the exact specification of each CRDT must be supported by FabricCRDT's chaincode execution environment. In order to enable the development of a wide range of applications, we enabled the support for executing JSON CRDTs. Furthermore, to facilitate the adoption of permissioned blockchains and decrease the learning curve for developing decentralized applications, we maintain FabricCRDT backward compatible with existing applications developed for Fabric. In other words, based on application design requirements, the developers of FabricCRDT have the option between Fabric's three-phase protocol with MVCC validation or creating CRDT-enabled applications where the conflicts of concurrent transactions are resolved internally without developers' intervention.

We evaluated FabricCRDT and compared its performance to Fabric by developing IoT-based supply chain applications for monitoring the temperature of perishable goods. The result of our experiments demonstrates the improved throughput of successful transactions on FabricCRDT over Fabric. Our approach successfully commits every valid transaction, while a high percentage of the transaction fails on Fabric due to MVCC validation-related failures. We reduced the complexity of developing decentralized applications by eliminating the failure of conflicting transactions. Since we ensure the successful commit of every correctly endorsed transaction, the developer is not required to consider concurrency-related failures and handle corresponding failure scenarios.

Despite the significantly improved throughput, FabricCRDT's use cases are limited to applications with exclusively I-confluent invariant conditions. The reason is that FabricCRDT forgoes Fabric's optimistic MVCC validation for improving the scalability and throughput, which is used for preserving non-I-confluent invariant conditions. Furthermore, as observed in the evaluation, as the size of JSON CRDT objects increases, the latency of transactions increases. The increased metadata causes this added latency in the JSON CRDT object, which decreases the efficiency of the proposed algorithms for applying the modifications to JSON CRDT objects. FabricCRDT must periodically prune the metadata to improve its memory utilization and efficiency.

Even though by enabling CRDT functionalities on FabricCRDT, we eliminate the coordination required for the correct execution of I-confluent applications, we used Fabric's coordination-based protocol in FabricCRDT for backward compatibility purposes. In order to realize the potentials of CRDTs in coordination-free permissioned blockchains, we proposed OrderlessChain. OrderlessChain uses a novel BFT coordination-free protocol for the safe and live execution of applications without paying the high coordination cost. OrderlessChain's organizations do not require coordination to reach a consensus to prevent Byzantine failures or serialize transactions into a total global order.

The proposed coordination-free protocol takes advantage of the permissioned property of blockchains to prevent Byzantine failures by enforcing endorsement policies. This approach enables OrderlessChain's honest and non-faulty organizations to detect Byzantine failures and prevent malicious behaviors without coordinating with other organizations.

We introduced several applications across various fields using the operation-based CRDT approach of OrderlessChain, including voting and auction applications, a distributed file storage system, and an FL system. We modeled these applications and systems using several CRDTs, including CRDT Maps, G-Counters, MV-Registers, and two novel CRDTs namely flCRDT and FileCRDT. By modeling the applications based on the supported CRDTs, OrderlessChain creates commutative and convergent transactions that can be processed in any order without requiring the organizations to coordinate for transaction serialization. Thus, it preserves the applications' I-confluent invariant conditions without coordination.

The extensive evaluation of OrderlessChain using the synthetic application demonstrated its scalability, high transaction throughput, and low latency. To display the potential of our coordination-free approach over the coordination-based protocols of Fabric and FabricCRDT, we conducted several experiments using the voting and auction applications. The results of their experiments demonstrated the significantly higher throughput and lower latency of OrderlessChain compared to Fabric and FabricCRDT.

Furthermore, we introduced OrderlessFL and OrderlessFile, which provide safe and private environments for realizing FL solutions and distributed file storage systems, respectively. The evaluation of these systems and the relatively effortless adaptions and extension of OrderlessChain with the novel CRDTs flCRDT and FileCRDT demonstrates the applicability of OrderlessChain to other domains and industries.

OrderlessChain's use cases are limited to applications that are I-confluent. Nevertheless, as discussed in previous chapters, many applications with I-confluent invariant conditions exist that are compatible with OrderlessChain. They benefit from the scalability and decentralized trust of OrderlessChain.

In summary, our evaluations and the realized decentralized applications confirm the applicability and the significant potential of the BFT coordination-free protocol of OrderlessChain as a scalable alternative over coordination-based permissioned blockchains for I-confluent applications.

7.2 Future Work

Although the proposed solutions of FabricCRDT and OrderlessChain are CRDT-agnostic, the specifications of various CRDTs must be implemented in the smart contract execution environment. We explained the integration of various CRDTs in both systems, and enabling the support for these CRDTs did not require a significant effort. However, future research should provide a generalized solution for realizing CRDT-adoptable applications. For example, the approaches used in Katara [201] for automatically creating CRDT-enabled applications from sequential non-CRDT implementations may also be implemented on FabricCRDT and OrderlessChain. Furthermore, despite the improved performance of OrderlessChain's operation-based CRDT modification approach over FabricCRDT's approach, future studies should address the problems regarding the high memory utilization and low efficiency of CRDTs, as the size of the CRDT objects increases.

Preserving the non-I-confluent invariant conditions of applications in coordination-free Byzantine environments remains a challenge and an open problem. As this dissertation demonstrates, reducing and eliminating coordination is critical for improving the scalability and throughput of permissioned blockchains. Therefore, future research should propose solutions for the correct execution of CRDT-based applications with non-I-confluent invariant conditions in BFT coordination-free environments. One potential approach for addressing this problem is taking advantage of *Reversible CRDTs* [214]. Through using the reversibility property of such CRDTs, coordination-free protocols on eventually consistent permissioned blockchains may be feasible, where the malicious modifications of CRDTs or the modifications that violate non-I-confluent invariant conditions are reversed. Hence, permissioned blockchains based on these protocols offer a scalable, safe, and live BFT system for the correct execution of non-I-confluent applications.

List of Acronyms and Abbreviations

BFT Byzantine Fault Tolerance

CRDT Conflict-free Replicated Data Type

DAG Directed Acyclic Graph

DDoS Distributed Denial-of-Service

DNN Deep Neural Network

EOV Execute-Order-Validate

EP Endorsement Policy

FL Federated Learning

G-Counter Grow-Only Counter

I-confluence Invariant Confluence

IoT Internet of Things

IPFS InterPlanetary File System

JSON JavaScript Object Notation

ML Machine Learning

MVCC Multiversion Concurrency Control

MV-Register Multi-Value Register

PKI Public Key Infrastructure

PoW Proof-of-Work

PS Parameter Server

SCL Smart Contract Library

SEC Strong Eventual Consistency

TPS Transactions per Second

VM Virtual Machine

VSCC Validation System Chaincode

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] W. Chen, Z. Xu, S. Shi, Y. Zhao, and J. Zhao. “A Survey of Blockchain Applications in Different Domains.” In: *Proceedings of the 2018 International Conference on Blockchain Technology and Application*. ICBTA 2018. Xi’an, China: ACM, 2018, pp. 17–21. ISBN: 9781450366465. DOI: 10.1145/3301403.3301407.

[2] L. Carter and J. Ubacht. “Blockchain Applications in Government.” In: *Proceedings of the 19th Annual International Conference on Digital Government Research: Governance in the Data Age*. dg.o ’18. Delft, The Netherlands: ACM, 2018. ISBN: 9781450365260. DOI: 10.1145/3209281.3209329.

[3] T. Alladi, V. Chamola, R. M. Parizi, and K.-K. R. Choo. “Blockchain Applications for Industry 4.0 and Industrial IoT: A Review.” In: *IEEE Access* 7 (2019), pp. 176935–176951. DOI: 10.1109/ACCESS.2019.2956748.

[4] K. Wüst and A. Gervais. “Do you Need a Blockchain?” In: *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. 2018, pp. 45–54. DOI: 10.1109/CVCBT.2018.00011.

[5] C. Cachin and M. Vukolic. “Blockchain Consensus Protocols in the Wild.” In: *CoRR* abs/1707.01873 (2017). arXiv: 1707.01873.

[6] C. Berger and H. P. Reiser. “Scaling Byzantine Consensus: A Broad Analysis.” In: *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. SERIAL’18. Rennes, France: ACM, 2018, pp. 13–18. ISBN: 9781450361101. DOI: 10.1145/3284764.3284767.

[7] D. Bradbury. “The Problem with Bitcoin.” In: *Computer Fraud and Security* 2013.11 (2013), pp. 5–8. ISSN: 1361-3723. DOI: 10.1016/S1361-3723(13)70101-5.

[8] J. A. Chacko, R. Mayer, and H.-A. Jacobsen. “Why Do My Blockchain Transactions Fail? A Study of Hyperledger Fabric.” In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD ’21. Virtual Event, China: ACM, 2021, pp. 221–234. ISBN: 9781450383431. DOI: 10.1145/3448016.3452823.

[9] S. Tikhomirov. “Ethereum: State of Knowledge and Research Perspectives.” In: *Foundations and Practice of Security*. Cham: Springer International Publishing, 2018, pp. 206–221. ISBN: 978-3-319-75650-9.

[10] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang. "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends." In: *2017 IEEE International Congress on Big Data (BigData Congress)*. 2017, pp. 557–564. DOI: 10.1109/BigDataCongress.2017.85.

[11] J. Eberhardt and S. Tai. "On or Off the Blockchain? Insights on Off-Chaining Computation and Data." In: *European Conference on Service-Oriented and Cloud Computing*. Springer International Publishing, 2017, pp. 3–15. ISBN: 978-3-319-67262-5.

[12] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-Free Replicated Data Types." In: *Stabilization, Safety, and Security of Distributed Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400. ISBN: 978-3-642-24550-3.

[13] E. Androulaki, A. Barger, V. Bortnikov, et al. "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains." In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: ACM, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190538.

[14] S. Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System." In: *Decentralized Business Review* (2008).

[15] A. Hafid, A. S. Hafid, and M. Samih. "Scaling Blockchains: A Comprehensive Survey." In: *IEEE Access* 8 (2020), pp. 125244–125262. DOI: 10.1109/ACCESS.2020.3007251.

[16] N. Chaudhry and M. M. Yousaf. "Consensus Algorithms in Blockchain: Comparative Analysis, Challenges and Opportunities." In: *2018 12th International Conference on Open Source Systems and Technologies (ICOSST)*. 2018, pp. 54–63. DOI: 10.1109/ICOSST.2018.8632190.

[17] D. Mingxiao, M. Xiaofeng, Z. Zhe, W. Xiangwei, and C. Qijun. "A Review on Consensus Algorithm of Blockchain." In: *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2017, pp. 2567–2572. DOI: 10.1109/SMC.2017.8123011.

[18] M. Barborak, A. Dahbura, and M. Malek. "The Consensus Problem in Fault-Tolerant Computing." In: *ACM Comput. Surv.* 25.2 (June 1993), pp. 171–220. ISSN: 0360-0300. DOI: 10.1145/152610.152612.

[19] M. Treaster. "A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems." In: *CoRR* abs/cs/0501002 (2005). arXiv: cs/0501002.

[20] D. Dujak and D. Sajter. "Blockchain Applications in Supply Chain." In: *SMART Supply Network*. Cham: Springer International Publishing, 2019, pp. 21–46. ISBN: 978-3-319-91668-2. DOI: 10.1007/978-3-319-91668-2_2.

[21] D. Tse, B. Zhang, Y. Yang, C. Cheng, and H. Mu. "Blockchain Application in Food Supply Information Security." In: *2017 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*. 2017, pp. 1357–1361. DOI: 10.1109/IEEM.2017.8290114.

[22] P. Zhang, D. C. Schmidt, J. White, and G. Lenz. "Chapter One - Blockchain Technology Use Cases in Healthcare." In: *Blockchain Technology: Platforms, Tools and Use Cases*. Ed. by P. Raj and G. C. Deka. Vol. 111. Advances in Computers. Elsevier, 2018, pp. 1–41. DOI: 10.1016/bs.adcom.2018.03.006.

[23] C. Dannen. *Introducing Ethereum and Solidity.* Springer, 2017. DOI: 10.1007/978-1-4842-2535-6.

[24] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. "Zcash Protocol Specification." In: *Tech. rep. 2016–1.10. Zerocoin Electric Coin Company, Tech. Rep.* (2016).

[25] L. S. Sankar, M. Sindhu, and M. Sethumadhavan. "Survey of Consensus Protocols on Blockchain Applications." In: *2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS).* IEEE, 2017, pp. 1–5. DOI: 10.1109/ICACCS.2017.8014672.

[26] S. Kim, Y. Kwon, and S. Cho. "A Survey of Scalability Solutions on Blockchain." In: *2018 International Conference on Information and Communication Technology Convergence (ICTC).* 2018, pp. 1204–1207. DOI: 10.1109/ICTC.2018.8539529.

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

[28] Q. Zhou, H. Huang, Z. Zheng, and J. Bian. "Solutions to Scalability of Blockchain: A Survey." In: *IEEE Access* 8 (2020), pp. 16440–16455. DOI: 10.1109/ACCESS.2020.2967218.

[29] L. Badea and M. C. Mungiu-Pupazan. "The Economic and Environmental Impact of Bitcoin." In: *IEEE Access* 9 (2021), pp. 48091–48104. DOI: 10.1109/ACCESS.2021.3068636.

[30] A. De Vries. "Bitcoin's Growing Energy Problem." In: *Joule* 2.5 (2018), pp. 801–805.

[31] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse. "Bitcoin-NG: A Scalable Blockchain Protocol." In: *13th USENIX symposium on networked systems design and implementation (NSDI 16).* 2016, pp. 45–59.

[32] C. T. Nguyen, D. T. Hoang, D. N. Nguyen, et al. "Proof-of-Stake Consensus Mechanisms for Future Blockchain Networks: Fundamentals, Applications and Opportunities." In: *IEEE Access* 7 (2019), pp. 85727–85745. DOI: 10.1109/ACCESS.2019.2925010.

[33] D. Ongaro and J. Ousterhout. "In Search of an Understandable Consensus Algorithm." In: *2014 USENIX Annual Technical Conference (USENIX ATC 14).* Philadelphia, PA: USENIX Association, 2014, pp. 305–319. ISBN: 978-1-931971-10-2.

[34] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery." In: *ACM Trans. Comput. Syst.* 20.4 (2002), pp. 398–461. ISSN: 0734-2071. DOI: 10.1145/571637.571640.

[35] A. Bessani, J. Sousa, and E. E. Alchieri. "State Machine Replication for the Masses with BFT-SMART." In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* 2014, pp. 355–362. DOI: 10.1109/DSN.2014.43.

[36] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich. "Blurring the Lines between Blockchains and Database Systems: The Case of Hyperledger Fabric." In: *Proceedings of the 2019 International Conference on Management of Data.* SIGMOD '19. Amsterdam, Netherlands: ACM, 2019, pp. 105–122. ISBN: 9781450356435. DOI: 10.1145/3299869.3319883.

[37] Ethereum Wiki Project. *On Sharding Blockchains.* Accessed: 2020-05-25. URL: https://github.com/ethereum/wiki/wiki/Sharding-FAQ.

[38] K. Zhang and H.-A. Jacobsen. "Towards Dependable, Scalable, and Pervasive Distributed Ledgers with Blockchains." In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 2018, pp. 1337–1346. DOI: 10.1109/ICDCS.2018.00134.

[39] P. Bailis, A. Fekete, M. J. Franklin, et al. "Coordination Avoidance in Database Systems." In: vol. 8. 3. VLDB Endowment, 2014, pp. 185–196. DOI: 10.14778/2735508.2735509.

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

[41] A. Sharma, F. M. Schuhknecht, and J. Dittrich. "Accelerating Analytical Processing in MVCC Using Fine-Granular High-Frequency Virtual Snapshotting." In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018, pp. 245–258. ISBN: 9781450347037. DOI: 10.1145/3183713.3196904.

[42] Apache Software Foundation. *CouchDB*. URL: https://couchdb.apache.org/.

[43] SAP. *HANA*. URL: https://sap.com/products/hana.html.

[44] P. Ruan, D. Loghin, Q.-T. Ta, et al. "A Transactional Perspective on Execute-Order-Validate Blockchains." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: ACM, 2020, pp. 543–557. ISBN: 9781450367356. DOI: 10.1145/3318464.3389693.

[45] S. Goel, A. Singh, R. Garg, M. Verma, and P. Jayachandran. "Resource Fairness and Prioritization of Transactions in Permissioned Blockchain Systems (Industry Track)." In: *Proceedings of the 19th International Middleware Conference Industry*. Middleware '18. Rennes, France: ACM, 2018, pp. 46–53. ISBN: 9781450360166. DOI: 10.1145/3284028.3284035.

[46] B. Pi, Y. Pan, E. Zhou, et al. "xFabLedger: Extensible Ledger Storage for Hyperledger Fabric." In: *2021 IEEE 11th International Conference on Electronics Information and Emergency Communication (ICEIEC)2021 IEEE 11th International Conference on Electronics Information and Emergency Communication (ICEIEC)*. 2021, pp. 5–11. DOI: 10.1109/ICEIEC51955.2021.9463838.

[47] P. Thakkar and S. Natarajan. "Scaling Blockchains Using Pipelined Execution and Sparse Peers." In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '21. Seattle, WA, USA: ACM, 2021, pp. 489–502. ISBN: 9781450386388. DOI: 10.1145/3472883.3486975.

[48] Z. István, A. Sorniotti, and M. Vukolić. "StreamChain: Do Blockchains Need Blocks?" In: *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. SERIAL'18. Rennes, France: ACM, 2018, pp. 1–6. ISBN: 9781450361101. DOI: 10.1145/3284764.3284765.

[49] C. Gorenflo, S. Lee, L. Golab, and S. Keshav. "FastFabric: Scaling Hyperledger Fabric to 20000 Transactions per Second." In: *International Journal of Network Management* 30.5 (2020). DOI: 10.1002/nem.2099.

[50] W. Zhao, M. Babi, W. Yang, et al. "Byzantine Fault Tolerance for Collaborative Editing with Commutative Operations." In: *2016 IEEE International Conference on Electro Information Technology (EIT)*. 2016, pp. 0246–0251. DOI: 10.1109/EIT.2016.7535248.

[51] A. Shoker, H. Yactine, and C. Baquero. "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*. PaPoC '17. Belgrade, Serbia: ACM, 2017. ISBN: 9781450349338. DOI: 10.1145/3064889.3064895.

[52] M. Capretto, M. Ceresa, A. F. Anta, A. Russo, and C. Sánchez. "Setchain: Improving Blockchain Scalability with Byzantine Distributed Sets and Barriers." In: *2022 IEEE International Conference on Blockchain (Blockchain)*. 2022, pp. 87–96. DOI: 10.1109/Blockchain55522.2022.00022.

[53] J.-P. Martin and L. Alvisi. "Fast Byzantine Consensus." In: *IEEE Transactions on Dependable and Secure Computing* 3.3 (2006), pp. 202–215. DOI: 10.1109/TDSC.2006.35.

[54] M. Kleppmann and A. R. Beresford. "A Conflict-Free Replicated JSON Datatype." In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (2017), pp. 2733–2746. DOI: 10.1109/TPDS.2017.2697382.

[55] P. Nasirifard, R. Mayer, and H.-A. Jacobsen. "FabricCRDT: A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains." In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. Davis, CA, USA: ACM, 2019, pp. 110–122. ISBN: 9781450370097. DOI: 10.1145/3361525.3361540.

[56] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas. "Communication-Efficient Learning of Deep Networks from Decentralized Data." In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Vol. 54. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1273–1282.

[57] P. Nasirifard. *OrderlessChain Source Code*. URL: https://github.com/orderlesschain/orderlesschain.

[58] P. Nasirifard. *OrderlessFL Source Code*. URL: https://github.com/orderlesschain/orderlessfl.

[59] P. Nasirifard. *OrderlessFile Source Code*. URL: https://github.com/orderlesschain/orderlessfile.

[60] P. Nasirifard, R. Mayer, and H.-A. Jacobsen. "OrderlessChain: A CRDT-based Coordination-free Blockchain Without Global Order of Transactions." In: *Proceedings of the 24th International Middleware Conference*. Middleware '23. Bologna, Italy: ACM, 2023.

[61] P. Nasirifard, R. Mayer, and H.-A. Jacobsen. "OrderlessChain: A CRDT-Enabled Blockchain without Total Global Order of Transactions: Poster Abstract." In: *Proceedings of the 23rd International Middleware Conference: Demos and Posters*. Middleware '22. Quebec, Quebec City, Canada: ACM, 2022, pp. 5–6. ISBN: 9781450399319. DOI: 10.1145/3565386.3565486.

[62] P. Nasirifard, R. Mayer, and H.-A. Jacobsen. "OrderlessFile: A CRDT-Enabled Permissioned Blockchain for File Storage: Poster Abstract." In: *Proceedings of the 23rd International Middleware Conference: Demos and Posters*. Middleware '22. Quebec, Quebec City, Canada: ACM, 2022, pp. 15–16. ISBN: 9781450399319. DOI: 10.1145/3565386.3565491.

[63] P. Nasirifard, R. Mayer, and H.-A. Jacobsen. "OrderlessFL: A CRDT-Enabled Permissioned Blockchain for Federated Learning: Poster Abstract." In: *Proceedings of the 23rd International Middleware Conference: Demos and Posters.* Middleware '22. Quebec, Quebec City, Canada: ACM, 2022, pp. 7–8. ISBN: 9781450399319. DOI: 10.1145/3565386.3565487.

[64] *Golang, Go Programming Language.* URL: https://golang.org/.

[65] *LevelDB.* URL: https://github.com/google/leveldb.

[66] L. Lamport. "Paxos Made Simple." In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), pp. 51–58.

[67] N. Preguiça, C. Baquero, and M. Shapiro. "Conflict-Free Replicated Data Types (CRDTs)." In: *Encyclopedia of Big Data Technologies.* Springer International Publishing, 2018, pp. 1–10. ISBN: 978-3-319-63962-8. DOI: 10.1007/978-3-319-63962-8_185-1.

[68] N. M. Preguiça. "Conflict-free Replicated Data Types: An Overview." In: *CoRR* abs/1806.10254 (2018). arXiv: 1806.10254.

[69] H. Liang and X. Feng. "Abstraction for Conflict-Free Replicated Data Types." In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* PLDI 2021. Virtual, Canada: ACM, 2021, pp. 636–650. ISBN: 9781450383912. DOI: 10.1145/3453483.3454067.

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

[71] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. "A Commutative Replicated Data Type for Cooperative Editing." In: *2009 29th IEEE International Conference on Distributed Computing Systems.* 2009, pp. 395–403. DOI: 10.1109/ICDCS.2009.20.

[72] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. "Riak DT Map: A Composable, Convergent Replicated Dictionary." In: *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency.* PaPEC '14. Amsterdam, The Netherlands: ACM, 2014. ISBN: 9781450327169. DOI: 10.1145/2596631.2596633.

[73] P. S. Almeida, A. Shoker, and C. Baquero. "Delta State Replicated Data Types." In: *CoRR* abs/1603.01529 (2016). arXiv: 1603.01529.

[74] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." In: *Commun. ACM* 21.7 (1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563.

[75] X. Défago, A. Schiper, and P. Urbán. "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey." In: *ACM Comput. Surv.* 36.4 (2004), pp. 372–421. ISSN: 0360-0300. DOI: 10.1145/1041680.1041682.

[76] C. E. Bezerra, F. Pedone, and R. Van Renesse. "Scalable State-Machine Replication." In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* 2014, pp. 331–342. DOI: 10.1109/DSN.2014.41.

[77] A. Bessani, J. Sousa, and E. E. Alchieri. “State Machine Replication for the Masses with BFT-SMART.” In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2014, pp. 355–362. DOI: 10.1109/DSN.2014.43.

[78] J. Polge, J. Robert, and Y. Le Traon. “Permissioned Blockchain Frameworks in the Industry: A Comparison.” In: *ICT Express* 7.2 (2021), pp. 229–233. ISSN: 2405-9595. DOI: 10.1016/j.icte.2020.09.002.

[79] G. Greenspan. *MultiChain Private Blockchain – White Paper*. 2015. URL: https://multichain.com/download/MultiChain-White-Paper.pdf.

[80] J. P. Morgan Chase. *A Permissioned Implementation of Ethereum*. 2018. URL: https://github.com/ConsenSys/quorum.

[81] H. Moniz. “The Istanbul BFT Consensus Algorithm.” In: *CoRR* abs/2002.03613 (2020). arXiv: 2002.03613.

[82] A. Barger, Y. Manevich, H. Meir, and Y. Tock. “A Byzantine Fault-Tolerant Consensus Library for Hyperledger Fabric.” In: *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2021, pp. 1–9. DOI: 10.1109/ICBC51069.2021.9461099.

[83] J. Sousa, A. Bessani, and M. Vukolic. “A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform.” In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018, pp. 51–58. DOI: 10.1109/DSN.2018.00018.

[84] A. Baliga, I. Subhod, P. Kamat, and S. Chatterjee. “Performance Evaluation of the Quorum Blockchain Platform.” In: *CoRR* abs/1809.03421 (2018). arXiv: 1809.03421.

[85] A. Baliga, N. Solanki, S. Verekar, et al. “Performance Characterization of Hyperledger Fabric.” In: *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. 2018, pp. 65–74. DOI: 10.1109/CVCBT.2018.00013.

[86] P. Thakkar, S. Nathan, and B. Viswanathan. “Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform.” In: (2018), pp. 264–276. DOI: 10.1109/MASCOTS.2018.00034.

[87] X. Xu, X. Wang, Z. Li, et al. “Mitigating Conflicting Transactions in Hyperledger Fabric-Permissioned Blockchain for Delay-Sensitive IoT Applications.” In: *IEEE Internet of Things Journal* 8.13 (2021), pp. 10596–10607. DOI: 10.1109/JIOT.2021.3050244.

[88] C. Gorenflo, L. Golab, and S. Keshav. “XOX Fabric: A Hybrid Approach to Transaction Execution.” In: *CoRR* abs/1906.11229 (2019). arXiv: 1906.11229.

[89] B. Ding, L. Kot, and J. Gehrke. “Improving Optimistic Concurrency Control through Transaction Batching and Operation Reordering.” In: *Proc. VLDB Endow.* 12.2 (2018), pp. 169–182. ISSN: 2150-8097. DOI: 10.14778/3282495.3282502.

[90] Q. Sun and Y. Yuan. “GBCL: Reduce Concurrency Conflicts in Hyperledger Fabric.” In: *2022 IEEE 13th International Conference on Software Engineering and Service Science (ICSESS)*. 2022, pp. 15–19. DOI: 10.1109/ICSESS54813.2022.9930267.

[91] *Roshi: A Large-scale CRDT Set Implementation for Timestamped Events.* URL: https://github.com/soundcloud/roshi.

[92] B. Nédelec, P. Molli, A. Mostefaoui, and E. Desmontils. "LSEQ: An Adaptive Structure for Sequences in Distributed Collaborative Editing." In: *Proceedings of the 2013 ACM Symposium on Document Engineering*. DocEng '13. Florence, Italy: ACM, 2013, pp. 37–46. ISBN: 9781450317894. DOI: 10.1145/2494266.2494278.

[93] G. Younes, A. Shoker, P. S. Almeida, and C. Baquero. "Integration Challenges of Pure Operation-Based CRDTs in Redis." In: *First Workshop on Programming Models and Languages for Distributed Computing*. PMLDC '16. Rome, Italy: ACM, 2016. ISBN: 9781450347754. DOI: 10.1145/2957319.2957375.

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

[95] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma. "Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types." In: *Engineering the Web in the Big Data Era*. Springer International Publishing, 2015, pp. 675–678. ISBN: 978-3-319-19890-3.

[96] Concordant. *The Concordant Vision.* 2020. URL: https://concordant.io/uploads/visionpaper-concordant_2020.pdf.

[97] P. Lopes, J. Sousa, V. Balegas, et al. "Antidote SQL: Relaxed When Possible, Strict When Necessary." In: *CoRR* abs/1902.03576 (2019). arXiv: 1902.03576.

[98] M. Zawirski, C. Baquero, A. Bieniusa, N. Preguiça, and M. Shapiro. "Eventually Consistent Register Revisited." In: *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data.* PaPoC '16. London, United Kingdom: ACM, 2016. ISBN: 9781450342964. DOI: 10.1145/2911151.2911157.

[99] Netflix. *Dynomite.* URL: https://github.com/Netflix/dynomite.

[100] B. Fitzpatrick. "Distributed Caching with Memcached." In: *Linux journal* 2004.124 (2004), p. 5.

[101] *SoundCloud.* URL: https://soundcloud.com/.

[102] Apple. *Notes.* URL: https://icloud.com/notes.

[103] TomTom. *GPS Navigation with TomTom.* 2016. URL: https://speakerdeck.com/ajantis/practical-data-synchronization-with-crdts-strangeloop-2016.

[104] M. Barbosa, B. Ferreira, J. Marques, B. Portela, and N. Preguiça. "Secure Conflict-Free Replicated Data Types." In: *International Conference on Distributed Computing and Networking 2021.* ICDCN '21. Nara, Japan: ACM, 2021, pp. 6–15. ISBN: 9781450389334. DOI: 10.1145/3427796.3427831.

[105] M. Kleppmann. "Making CRDTs Byzantine Fault Tolerant." In: *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data.* PaPoC '22. Rennes, France: ACM, 2022, pp. 8–15. ISBN: 9781450392563. DOI: 10.1145/3517209.3524042.

[106] F. Jacob, C. Beer, N. Henze, and H. Hartenstein. "Analysis of the Matrix Event Graph Replicated Data Type." In: *IEEE Access* 9 (2021), pp. 28317–28333. DOI: 10.1109/ACCESS.2021.3058576.

[107] A. Auvolat and F. Taïani. "Merkle Search Trees: Efficient State-Based CRDTs in Open Networks." In: *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. 2019, pp. 221–22109. DOI: 10.1109/SRDS47363.2019.00032.

[108] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. "A Survey on Homomorphic Encryption Schemes: Theory and Implementation." In: *ACM Comput. Surv.* 51.4 (2018). ISSN: 0360-0300. DOI: 10.1145/3214303.

[109] V. Cholvi, A. F. Anta, C. Georgiou, et al. "Byzantine-tolerant Distributed Grow-only Sets: Specification and Applications." In: *CoRR* abs/2103.08936 (2021). arXiv: 2103.08936.

[110] M. Sauwens, K. Jannes, B. Lagaisse, and W. Joosen. "SCEW: Programmable BFT-Consensus with Smart Contracts for Client-Centric P2P Web Applications." In: *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '21. Online, United Kingdom: ACM, 2021. ISBN: 9781450383387. DOI: 10.1145/3447865.3457965.

[111] K. Karlsson, W. Jiang, S. Wicker, et al. "Vegvisir: A Partition-Tolerant Blockchain for the Internet-of-Things." In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 2018, pp. 1150–1158. DOI: 10.1109/ICDCS.2018.00114.

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

[113] H. Y. Wu, L. J. Li, H.-Y. Paik, and S. S. Kanhere. "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 (TrustCom)*. 2021, pp. 976–987. DOI: 10.1109/TrustCom53373.2021.00136.

[114] *Enhanced Concurrency Control*. Accessed: 2019-05-06. 2019. URL: https://jira.hyperledger.org/browse/FAB-10711.

[115] *Twitter*. URL: https://twitter.com/.

[116] F. Pedone and A. Schiper. "Generic Broadcast." In: *Distributed Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 94–106. ISBN: 978-3-540-48169-0.

[117] L. Lamport. "Generalized Consensus and Paxos." In: (2005).

[118] L. Lamport. "Lower Bounds for Asynchronous Consensus." In: *Distributed Computing* 19.2 (2006), pp. 104–125.

[119] C. Li, D. Porto, A. Clement, et al. "Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary." In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 265–278. ISBN: 9781931971966.

[120] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "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.* SOSP '11. Cascais, Portugal: ACM, 2011, pp. 401–416. ISBN: 9781450309776. DOI: 10.1145/2043556.2043593.

[121] F. Cristian, H. Aghili, R. Strong, and D. Dolev. "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement." In: *Information and Computation* 118.1 (1995), pp. 158–179.

[122] P. E. O'Neil. "The Escrow Transactional Method." In: *ACM Trans. Database Syst.* (1986), pp. 405–430. DOI: 10.1145/7239.7265.

[123] V. Balegas, D. Serra, S. Duarte, et al. "Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants." In: *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS).* 2015, pp. 31–36. DOI: 10.1109/SRDS.2015.32.

[124] V. Balegas, S. Duarte, C. Ferreira, et al. "Putting Consistency Back into Eventual Consistency." In: *Proceedings of the Tenth European Conference on Computer Systems.* EuroSys '15. Bordeaux, France: ACM, 2015. ISBN: 9781450332385. DOI: 10.1145/2741948.2741972.

[125] J. Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. "Warranties for Faster Strong Consistency." In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14).* Seattle, WA: USENIX Association, 2014, pp. 503–517. ISBN: 978-1-931971-09-6.

[126] M. Whittaker and J. M. Hellerstein. "Checking Invariant Confluence, In Whole or In Parts." In: *SIGMOD Rec.* 49.1 (2020), pp. 7–14. ISSN: 0163-5808. DOI: 10.1145/3422648.3422651.

[127] R. Friedman and K. Birman. *Trading Consistency for Availability in Distributed Systems.* Tech. rep. 1996.

[128] J. M. Hellerstein. "The Declarative Imperative: Experiences and Conjectures in Distributed Logic." In: *SIGMOD Rec.* 39.1 (2010), pp. 5–19. ISSN: 0163-5808. DOI: 10.1145/1860702.1860704.

[129] T. J. Ameloot, F. Neven, and J. Van Den Bussche. "Relational Transducers for Declarative Networking." In: *J. ACM* 60.2 (2013). ISSN: 0004-5411. DOI: 10.1145/2450142.2450151.

[130] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. "Logic and Lattices for Distributed Programming." In: *Proceedings of the Third ACM Symposium on Cloud Computing.* SoCC '12. San Jose, California: ACM, 2012. ISBN: 9781450317610. DOI: 10.1145/2391229.2391230.

[131] M. Pires, S. Ravi, and R. Rodrigues. "Generalized Paxos Made Byzantine (and Less Complex)." In: *Stabilization, Safety, and Security of Distributed Systems.* Cham: Springer International Publishing, 2017, pp. 203–218. ISBN: 978-3-319-69084-1.

[132] P. Raykov, N. Schiper, and F. Pedone. "Byzantine Fault-Tolerance with Commutative Commands." In: *Principles of Distributed Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 329–342. ISBN: 978-3-642-25873-2.

[133] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovič, and D.-A. Seredinschi. “The Consensus Number of a Cryptocurrency.” In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. PODC ’19. Toronto ON, Canada: ACM, 2019, pp. 307–316. ISBN: 9781450362177. DOI: 10.1145/3293611.3331589.

[134] G. O. Karame, E. Androulaki, and S. Capkun. “Two Bitcoins at the Price of One? Double-Spending Attacks on Fast Payments in Bitcoin.” In: (2012).

[135] J.-M. Chang and N. F. Maxemchuk. “Reliable Broadcast Protocols.” In: *ACM Trans. Comput. Syst.* 2.3 (1984), pp. 251–273. ISSN: 0734-2071. DOI: 10.1145/989.357400.

[136] D. Collins, R. Guerraoui, J. Komatovic, et al. “Online Payments by Merely Broadcasting Messages.” In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2020, pp. 26–38. DOI: 10.1109/DSN48063.2020.00023.

[137] M. Aledhari, R. Razzak, R. M. Parizi, and F. Saeed. “Federated Learning: A Survey on Enabling Technologies, Protocols, and Applications.” In: *IEEE Access* 8 (2020), pp. 140699–140725. DOI: 10.1109/ACCESS.2020.3013541.

[138] J. Verbraeken, M. Wolting, J. Katzy, et al. “A Survey on Distributed Machine Learning.” In: *ACM Comput. Surv.* 53.2 (Mar. 2020). ISSN: 0360-0300. DOI: 10.1145/3377454.

[139] X. Yin, Y. Zhu, and J. Hu. “A Comprehensive Survey of Privacy-Preserving Federated Learning: A Taxonomy, Review, and Future Directions.” In: *ACM Comput. Surv.* 54.6 (2021). ISSN: 0360-0300. DOI: 10.1145/3460427.

[140] B. Liu, M. Ding, S. Shaham, et al. “When Machine Learning Meets Privacy: A Survey and Outlook.” In: *ACM Comput. Surv.* 54.2 (Mar. 2021). ISSN: 0360-0300. DOI: 10.1145/3436755.

[141] K. Bonawitz, H. Eichner, W. Grieskamp, et al. “Towards Federated Learning at Scale: System Design.” In: 1 (2019), pp. 374–388.

[142] V. Mothukuri, R. M. Parizi, S. Pouriyeh, et al. “A Survey on Security and Privacy of Federated Learning.” In: *Future Generation Computer Systems* 115 (2021), pp. 619–640. ISSN: 0167-739X. DOI: 10.1016/j.future.2020.10.007.

[143] L. Lyu, H. Yu, and Q. Yang. “Threats to Federated Learning: A Survey.” In: *CoRR* abs/2003.02133 (2020). arXiv: 2003.02133.

[144] V. Mugunthan, R. Rahman, and L. Kagal. “BlockFLow: An Accountable and Privacy-Preserving Solution for Federated Learning.” In: *CoRR* abs/2007.03856 (2020). arXiv: 2007.03856.

[145] P. Ramanan and K. Nakayama. “BAFFLE : Blockchain Based Aggregator Free Federated Learning.” In: *2020 IEEE International Conference on Blockchain (Blockchain)*. 2020, pp. 72–81. DOI: 10.1109/Blockchain50366.2020.00017.

[146] Y. Zhao, J. Zhao, L. Jiang, et al. “Privacy-Preserving Blockchain-Based Federated Learning for IoT Devices.” In: *IEEE Internet of Things Journal* 8.3 (2021), pp. 1817–1829. DOI: 10.1109/JIOT.2020.3017377.

[147] X. Wu, Z. Wang, J. Zhao, Y. Zhang, and Y. Wu. "FedBC: Blockchain-based Decentralized Federated Learning." In: *2020 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*. 2020, pp. 217–221. DOI: 10.1109/ICAICA50127.2020.9182705.

[148] Y. Hu, W. Xia, J. Xiao, and C. Wu. "GFL: A Decentralized Federated Learning Framework Based On Blockchain." In: *CoRR* abs/2010.10996 (2020). arXiv: 2010.10996.

[149] H. B. Desai, M. S. Ozdayi, and M. Kantarcioglu. "BlockFLA: Accountable Federated Learning via Hybrid Blockchain Architecture." In: *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*. CODASPY '21. Virtual Event, USA: ACM, 2021, pp. 101–112. ISBN: 9781450381437. DOI: 10.1145/3422337.3447837.

[150] Z. Wang and Q. Hu. "Blockchain-based Federated Learning: A Comprehensive Survey." In: *CoRR* abs/2110.02182 (2021). arXiv: 2110.02182.

[151] D. C. Nguyen, M. Ding, Q.-V. Pham, et al. "Federated Learning Meets Blockchain in Edge Computing: Opportunities and Challenges." In: *IEEE Internet of Things Journal* 8.16 (2021), pp. 12806–12825. DOI: 10.1109/JIOT.2021.3072611.

[152] C. Dwork. "Differential Privacy: A Survey of Results." In: *Theory and Applications of Models of Computation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–19. ISBN: 978-3-540-79228-4.

[153] J. Benet. *IPFS - Content Addressed, Versioned, P2P File System*. 2014.

[154] W. Dai, Y. Zhou, N. Dong, H. Zhang, and E. P. Xing. "Toward Understanding the Impact of Staleness in Distributed Machine Learning." In: *CoRR* abs/1810.03264 (2018). arXiv: 1810.03264.

[155] W. Zhang, S. Gupta, X. Lian, and J. Liu. "Staleness-Aware Async-SGD for Distributed Deep Learning." In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. IJCAI'16. New York, New York, USA: AAAI Press, 2016, pp. 2350–2356.

[156] Q. Ho, J. Cipar, H. Cui, et al. "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server." In: *Advances in Neural Information Processing Systems*. Vol. 26. Curran Associates, Inc., 2013.

[157] J. Jiang, B. Cui, C. Zhang, and L. Yu. "Heterogeneity-Aware Distributed Parameter Servers." In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, pp. 463–478. DOI: 10.1145/3035918.3035933.

[158] M. Li, D. G. Andersen, J. W. Park, et al. "Scaling Distributed Machine Learning with the Parameter Server." In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 583–598. ISBN: 9781931971164.

[159] C. Xie, S. Koyejo, and I. Gupta. "Asynchronous Federated Optimization." In: *CoRR* abs/1903.03934 (2019). arXiv: 1903.03934.

[160] P. Sharma, R. Jindal, and M. D. Borah. "Blockchain Technology for Cloud Storage: A Systematic Literature Review." In: *ACM Comput. Surv.* 53.4 (Aug. 2020). ISSN: 0360-0300. DOI: 10.1145/3403954.

[161] N. Zahed Benisi, M. Aminian, and B. Javadi. “Blockchain-based Decentralized Storage Networks: A Survey.” In: *Journal of Network and Computer Applications* 162 (2020), p. 102656. ISSN: 1084-8045. DOI: `10.1016/j.jnca.2020.102656`.

[162] N. Deepa, Q.-V. Pham, D. C. Nguyen, et al. “A Survey on Blockchain for Big Data: Approaches, Opportunities, and Future Directions.” In: *Future Generation Computer Systems* 131 (2022), pp. 209–226. ISSN: 0167-739X. DOI: `10.1016/j.future.2022.01.017`.

[163] Storj Labs. *Storj: A Decentralized Cloud Storage Network Framework.*

[164] D. Vorick and L. Champine. *Sia: Simple Decentralized Storage.*

[165] Protocol Labs. *Filecoin: A Decentralized Storage Network.*

[166] S. Ruj, M. S. Rahman, A. Basu, and S. Kiyomoto. “BlockStore: A Secure Decentralized Storage Framework on Blockchain.” In: *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA).* 2018, pp. 1096–1103. DOI: `10.1109/AINA.2018.00157`.

[167] B. Guidi, A. Michienzi, and L. Ricci. “Evaluating the Decentralisation of Filecoin.” In: DICG ’22. Quebec, Quebec City, Canada: ACM, 2022, pp. 13–18. ISBN: 9781450399289. DOI: `10.1145/3565383.3566108`.

[168] R. Vaillant, D. Vasilas, M. Shapiro, and T. L. Nguyen. “CRDTs for Truly Concurrent File Systems.” In: *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems.* HotStorage ’21. Virtual, USA: ACM, 2021, pp. 35–41. ISBN: 9781450385503. DOI: `10.1145/3465332.3470872`.

[169] V. Tao, M. Shapiro, and V. Rancurel. “Merging Semantics for Conflict Updates in Geo-Distributed File Systems.” In: *Proceedings of the 8th ACM International Systems and Storage Conference.* SYSTOR ’15. Haifa, Israel: ACM, 2015. ISBN: 9781450336079. DOI: `10.1145/2757667.2757683`.

[170] G. Pestana. *Conflict-Free Replicated JSON Implementation in Go.* Accessed: 2019-05-05. 2018. URL: `https://github.com/gpestana/rdoc`.

[171] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso. “Evaluating CRDTs for Real-Time Document Editing.” In: *Proceedings of the 11th ACM Symposium on Document Engineering.* DocEng ’11. Mountain View, California, USA: ACM, 2011, pp. 103–112. ISBN: 9781450308632. DOI: `10.1145/2034691.2034717`.

[172] P. van Hardenberg and M. Kleppmann. “PushPin: Towards Production-Quality Peer-to-Peer Collaboration.” In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data.* PaPoC ’20. Heraklion, Greece: ACM, 2020. ISBN: 9781450375245. DOI: `10.1145/3380787.3393683`.

[173] B. Chandramouli, G. Prasaad, D. Kossmann, et al. “FASTER: A Concurrent Key-Value Store with In-Place Updates.” In: *Proceedings of the 2018 International Conference on Management of Data.* SIGMOD ’18. Houston, TX, USA: ACM, 2018, pp. 275–290. ISBN: 9781450347037. DOI: `10.1145/3183713.3196898`.

[174] M. Zawirski, N. Preguiça, S. Duarte, et al. "Write Fast, Read in the Past: Causal Consistency for Client-Side Applications." In: *Proceedings of the 16th Annual Middleware Conference*. Middleware '15. Vancouver, BC, Canada: ACM, 2015, pp. 75–87. ISBN: 9781450336185. DOI: 10.1145/2814576.2814733.

[175] A.-N. Mehdi, P. Urso, V. Balegas, and N. Perguiça. "Merging OT and CRDT Algorithms." In: *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*. PaPEC '14. Amsterdam, The Netherlands: ACM, 2014. DOI: 10.1145/2596631.2596636.

[176] T. Bocek, B. B. Rodrigues, T. Strasser, and B. Stiller. "Blockchains Everywhere - a Use-Case of Blockchains in the Pharma Supply-Chain." In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 2017, pp. 772–777. DOI: 10.23919/INM.2017.7987376.

[177] F. Tian. "A Supply Chain Traceability System for Food Safety Based on HACCP, Blockchain and Internet of Things." In: *2017 International Conference on Service Systems and Service Management*. 2017, pp. 1–6. DOI: 10.1109/ICSSSM.2017.7996119.

[178] P. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 2009.

[179] *TPC-C*. URL: http://tpc.org/tpcc/.

[180] *TPC-H*. URL: http://tpc.org/tpch/.

[181] *Apache Kafka*. URL: https://kafka.apache.org/.

[182] *Apache ZooKeeper*. URL: https://zookeeper.apache.org/.

[183] *Hyperledger Caliper*. URL: https://hyperledger.github.io/caliper/.

[184] H. Chai and W. Zhao. "Byzantine Fault Tolerance for Services with Commutative Operations." In: *2014 IEEE International Conference on Services Computing*. 2014, pp. 219–226. DOI: 10.1109/SCC.2014.37.

[185] P. Nasirifard, R. Mayer, and H.-A. Jacobsen. *OrderlessChain: Do Permissioned Blockchains Need Total Global Order of Transactions?* 2022. DOI: 10.48550/ARXIV.2210.01477.

[186] J. Huang, D. He, M. S. Obaidat, et al. "The Application of the Blockchain Technology in Voting Systems: A Review." In: *ACM Comput. Surv.* 54.3 (2021). ISSN: 0360-0300. DOI: 10.1145/3439725.

[187] S. Gilbert and N. Lynch. "Perspectives on the CAP Theorem." In: *Computer* 45.2 (2012), pp. 30–36. DOI: 10.1109/MC.2011.389.

[188] U. Maurer. "Modelling a Public-key Infrastructure." In: *Computer Security — ESORICS 96*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 325–350. ISBN: 978-3-540-70675-5.

[189] H. S. Galal and A. M. Youssef. "Verifiable Sealed-Bid Auction on the Ethereum Blockchain." In: *Financial Cryptography and Data Security: FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers*. Nieuwpoort, Curaçao: Springer Berlin Heidelberg, 2018, pp. 265–278. ISBN: 978-3-662-58819-2. DOI: 10.1007/978-3-662-58820-8_18.

[190] D. Sun, S. Xia, C. Sun, and D. Chen. “Operational Transformation for Collaborative Word Processing.” In: *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*. CSCW ’04. Chicago, Illinois, USA: ACM, 2004, pp. 437–446. ISBN: 1581138105. DOI: 10.1145/1031607.1031681.

[191] C. Sun and C. Ellis. “Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements.” In: *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*. CSCW ’98. Seattle, Washington, USA: ACM, 1998, pp. 59–68. ISBN: 1581130090. DOI: 10.1145/289444.289469.

[192] *gRPC, a High Performance, Open-Source Universal RPC Framework*. URL: https://grpc.io/.

[193] J. Bauwens and E. Gonzalez Boix. “Memory Efficient CRDTs in Dynamic Environments.” In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2019. Athens, Greece: ACM, 2019, pp. 48–57. ISBN: 9781450369879. DOI: 10.1145/3358504.3361231.

[194] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan. “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*. Onward! 2019. Athens, Greece: ACM, 2019, pp. 154–178. ISBN: 9781450369954. DOI: 10.1145/3359591.3359737.

[195] Oracle. *Read-Your-Writes Consistency*. URL: https://bit.ly/3dIAXOp.

[196] S. J. Castiñeira and A. Bieniusa. “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*. PaPoC ’15. Bordeaux, France: ACM, 2015. ISBN: 9781450335379. DOI: 10.1145/2745947.2745952.

[197] D. Mealha, N. Preguiça, M. C. Gomes, and J. Leitão. “Data Replication on the Cloud/Edge.” In: *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC ’19. Dresden, Germany: ACM, 2019. ISBN: 9781450362764. DOI: 10.1145/3301419.3323973.

[198] T. Jungnickel and L. Oldenburg. “Pluto: The CRDT-Driven IMAP Server.” In: *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC ’17. Belgrade, Serbia: ACM, 2017. ISBN: 9781450349338. DOI: 10.1145/3064889.3064891.

[199] A. van der Linde, P. Fouto, J. Leitão, et al. “Legion: Enriching Internet Services with Peer-to-Peer Interactions.” In: *Proceedings of the 26th International Conference on World Wide Web*. WWW ’17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 283–292. ISBN: 9781450349130. DOI: 10.1145/3038912.3052673.

[200] M. Najafzadeh, M. Shapiro, and P. Eugster. “Co-Design and Verification of an Available File System.” In: *Verification, Model Checking, and Abstract Interpretation*. Springer International Publishing, 2018, pp. 358–381. ISBN: 978-3-319-73721-8.

[201] S. Laddad, C. Power, M. Milano, A. Cheung, and J. M. Hellerstein. "Katara: Synthesizing CRDTs with Verified Lifting." In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (2022). DOI: 10.1145/3563336.

[202] Q. Li, Z. Wen, Z. Wu, et al. "A Survey on Federated Learning Systems: Vision, Hype and Reality for Data Privacy and Protection." In: *IEEE Transactions on Knowledge and Data Engineering* (2021), pp. 1–1. DOI: 10.1109/TKDE.2021.3124599.

[203] Y. Zhou, Q. Ye, and J. Lv. "Communication-Efficient Federated Learning With Compensated Overlap-FedAvg." In: *IEEE Transactions on Parallel and Distributed Systems* 33.1 (2022), pp. 192–205. DOI: 10.1109/TPDS.2021.3090331.

[204] L. Bottou. "Large-Scale Machine Learning with Stochastic Gradient Descent." In: *Proceedings of COMPSTAT'2010.* Heidelberg: Physica-Verlag HD, 2010, pp. 177–186. ISBN: 978-3-7908-2604-3.

[205] M. Assran, N. Loizou, N. Ballas, and M. G. Rabbat. "Stochastic Gradient Push for Distributed Deep Learning." In: vol. abs/1811.10792. 2018. arXiv: 1811.10792.

[206] M. Abadi, A. Chu, I. Goodfellow, et al. "Deep Learning with Differential Privacy." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* CCS '16. Vienna, Austria: ACM, 2016, pp. 308–318. ISBN: 9781450341394. DOI: 10.1145/2976749.2978318.

[207] M. Naehrig, K. Lauter, and V. Vaikuntanathan. "Can Homomorphic Encryption Be Practical?" In: *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop.* CCSW '11. Chicago, Illinois, USA: ACM, 2011, pp. 113–124. ISBN: 9781450310048. DOI: 10.1145/2046660.2046682.

[208] W. Samek, A. Binder, G. Montavon, S. Lapuschkin, and K.-R. Müller. "Evaluating the Visualization of What a Deep Neural Network Has Learned." In: *IEEE Transactions on Neural Networks and Learning Systems* 28.11 (2017), pp. 2660–2673. DOI: 10.1109/TNNLS.2016.2599820.

[209] A. Canziani, A. Paszke, and E. Culurciello. "An Analysis of Deep Neural Network Models for Practical Applications." In: *CoRR* abs/1605.07678 (2016). arXiv: 1605.07678.

[210] A. J. Myles, R. N. Feudale, Y. Liu, N. A. Woody, and S. D. Brown. "An Introduction to Decision Tree Modeling." In: *Journal of Chemometrics: A Journal of the Chemometrics Society* 18.6 (2004), pp. 275–285.

[211] *TensorFlow.* URL: https://tensorflow.org/.

[212] *Keras Sequential DNN Model.* Accessed: 2022-09-15. URL: https://tensorflow.org/tutorials/quickstart/beginner.

[213] *MNIST Dataset.* URL: https://yann.lecun.com/exdb/mnist/.

[214] Y. Mao, Z. Liu, and H.-A. Jacobsen. "Reversible Conflict-Free Replicated Data Types." In: *Proceedings of the 23rd Conference on 23rd ACM/IFIP International Middleware Conference.* Middleware '22. Quebec, QC, Canada: ACM, 2022, pp. 295–307. ISBN: 9781450393409. DOI: 10.1145/3528535.3565252.