University of California
Santa Barbara

Large-Scale Data Management Using Permissioned Blockchains

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Mohammad Javad Amiri

Committee in charge:

Professor Divyakant Agrawal, Co-Chair
Professor Amr El Abbadi, Co-Chair
Professor Tevfik Bultan

June 2020

The Dissertation of Mohammad Javad Amiri is approved.

Professor Tevfik Bultan

Professor Divyakant Agrawal, Committee Co-Chair

Professor Amr El Abbadi, Committee Co-Chair

June 2020

Large-Scale Data Management Using

Permissioned Blockchains

by

Mohammad Javad Amiri

To my family.

Acknowledgements

I would like to express my sincere gratitude to my advisors, Divy Agrawal and Amr El Abbadi. I could not have imagined having better advisors and mentors for my PhD study. I started working with them in my fourth year of PhD, almost two and a half years ago, and without their guidance, patience, continuous support, enthusiasm, encouragement, and useful critiques I clearly was not able to finish my PhD. Both Divy and Amr have the attitude and the substance of a genius: they convincingly guided and encouraged me to be professional and do the right thing even when the road got tough. Their guidance significantly helped me shape my PhD research directions. It was really a great pleasure working with them and from where I am to where I will be, I owe my thanks to them.

I am deeply indebted to my initial advisor, Professor Jianwen Su. I had spent three fruitful years in his research group, truly enjoyed taking his graduate classes, and learned a lot from him. I am really thankful for his valuable and constructive suggestions and for allowing me to grow as a researcher.

I would also like to acknowledge Professor Tevfik Bultan for serving on my dissertation committee. His valuable feedback throughout my graduate studies taught me maturity and inspired my work.

I have been fortunate to work with many great researchers during my PhD. Collaborating with Joris Duguépéroux and Professor Tristan Allard was a great pleasure. At Distributed Systems Lab also, I enjoyed the discussions, brainstorming, and work with Vaibhav Arora, Victor Zakhary, Sujaya Maiyya, Abtin Bateni, Matthew Ho, and Ishtiyaque Ahmad.

I would like to thank my friends in Santa Barbara who made the PhD time a pleasurable experience. I would like to mention Navid, Mitra, Soheil, Neda, Saeed, Razieh, Kasra, Razieh, Mahdi, Samira, Farnood, Farnaz, Sepehr, Amirali, Mahdi, Hedyeh, and

Hamed. I also greatly thank all my old friends; Mohsen Abbasi, Milad Bashiri, Amir Lajevardi, Rouhallah Khosroshahi, Mahdi Kazemi, Mostafa Khoramabadi Arani, Hessam Mohammadian, Sadegh Kazemi, Bijan Ghahremani, Zeinab Mapar, Bahar Asgari, Mahyar Najibi, Sina Salehi, Hamid Azimi, Nastaran Mahmoudyar, Javid Esmaeeli, Shiva Shahrokhi, Amir Ebrahimifard, and Mehrdad Forouzandeh, who I know for a long time. I would also like to especially thank Hamed and Lida for all the great moments we shared.

There is no greater source of support than that of my Family. They are my ultimate inspiration and source of strength. My sincerest thanks to my parents, my siblings, Alireza, Reza, Ahmadreza, Amir, Zahra, and Somayeh, my nephews and nieces, whose love and guidance are with me in whatever I pursue.

I am most grateful for the daily love and encouragement from my wife, Mahnaz. Words cannot describe how lucky I am to have her in my life. She has always been unconditionally supportive, pushing me to greater success. Her love for science and dedication to becoming a better researcher is contagious. Her touch, influence, and elegant thoughts are in many, if not all, of the works in this dissertation. Thank you.

Curriculum Vitæ
Mohammad Javad Amiri

Education

2020	Ph.D. in Computer Science, University of California, Santa Barbara.
2013	M.S. in Software Engineering, Iran University of Science and Technology.
2011	B.S. in Computer Engineering, Iran University of Science and Technology.

Publications

[1] **Mohammad Javad Amiri**, Joris Duguépéroux, Tristan Allard, Divyakant Agrawal, Amr El Abbadi, *SEPAR: Towards Regulating Future of Work Multi-Platform Crowdworking Environments with Privacy Guarantees*, The 30th Web *Conf.* (*WWW*), pp. 1891-1903, Ljubljana, Slovenia, 2021.

[2] **Mohammad Javad Amiri**, Divyakant Agrawal, Amr El Abbadi, *SharPer: Sharding Permissioned Blockchains Over Network Clusters*, ACM SIGMOD *Int. Conf.* on Management of Data, pp. 76-88, Xi'an, Shaanxi, China, 2021.

[3] **Mohammad Javad Amiri**, Divyakant Agrawal, Amr El Abbadi *Permissioned Blockchains: Properties, Techniques and Applications* [Tutorial], ACM SIGMOD *Int. Conf.* on Management of Data, pp. 2813-2820, Xi'an, Shaanxi, China, 2021.

[4] **Mohammad Javad Amiri**, Sujaya Maiyya, Divyakant Agrawal, Amr El Abbadi, *SeeMoRe: A Fault-Tolerant Protocol for Hybrid Cloud Environments*, The 36th IEEE *Int. Conf.* on Data Engineering (ICDE), pp. 1345-1356, Dallas, 2020.

[5] **Mohammad Javad Amiri**, Divyakant Agrawal, Amr El Abbadi, *Modern Large-Scale Data Management Systems after 40 Years of Consensus* [Tutorial], The 36th IEEE *Int. Conf.* on Data Engineering (ICDE), pp. 1794-1797, Dallas, 2020.

[6] Divyakant Agrawal, Amr El Abbadi, **Mohammad Javad Amiri**, Sujaya Maiyya, Victor Zakhary, *Blockchains and Databases: Opportunities and Challenges for the Permissioned and the Permissionless*, 24th European *Conf.* on Advances in Databases and Information Systems (ADBIS), LNCS 12245, pp. 3-7, Lyon, France, 2020. [Invited Paper]

[7] **Mohammad Javad Amiri**, Divyakant Agrawal, Amr El Abbadi, *CAPER: A Cross-Application Permissioned Blockchain*, The 45th *Int. Conf.* on Very Large Data Bases (VLDB), PVLDB 12(11), pp. 1385-1398, Los Angeles, 2019.

[8] **Mohammad Javad Amiri**, Divyakant Agrawal, Amr El Abbadi, *ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems*, The 39th IEEE *Int. Conf.* on Distributed Computing Systems (ICDCS), pp. 1337-1347, Dallas, 2019.

[9] **Mohammad Javad Amiri**, Divyakant Agrawal, *VIEW: An Incremental Approach to Verify Evolving Workflows*, The 34th ACM/SIGAPP Symposium on Applied Computing (SAC), pp. 85-93, Cyprus, 2019.

[10] **Mohammad Javad Amiri**, Divyakant Agrawal, Amr El Abbadi, *On Sharding Permissioned Blockchains* [Short Paper] The Second IEEE *Int. Conf.* on Blockchain, pp. 282-285, Atlanta, 2019.

[11] Vaibhav Arora, **Mohammad Javad Amiri**, Divyakant Agrawal, Amr El Abbadi, *M-DB: A Continuous Data Processing and Monitoring Framework for IoT Applications* The 12th IEEE *Int. Conf.* on Internet of Things (iThings), pp. 1096-1105, Atlanta, 2019.

[12] Victor Zakhary, **Mohammad Javad Amiri**, Sujaya Maiyya, Divyakant Agrawal, Amr El Abbadi, *Towards Global Asset Management in Blockchain Systems*, Blockchain and Distributed Ledger Workshop (BCDL), in conjunction with VLDB, Los Angeles, 2019.

[13] Sujaya Maiyya, Victor Zakhary, **Mohammad Javad Amiri**, Divyakant Agrawal, Amr El Abbadi, *Database and Distributed Computing Foundations of Blockchains* [Tutorial], ACM SIGMOD *Int. Conf.* on Management of Data, pp. 2036-2041, The Netherlands, 2019.

[14] **Mohammad Javad Amiri**, Mahnaz Koupaee, Divyakant Agrawal, *On Similarity of Object-Aware Workflows*, The 13th IEEE *Int. Conf.* on Service-Oriented System Engineering (SOSE), pp. 84-89, San Francisco, 2019.

[15] Arezoo Yazdani, **Mohammad Javad Amiri**, Saeed Parsa, Mahnaz Koupaee, *Automatic Test Case Generation from Business Process Models*, Journal of Requirements Engineering, 24(1), pp. 119-132, 2019.

[16] **Mohammad Javad Amiri**, *Object-aware Identification of Microservices* [Short Paper], The 15th IEEE *Int. Conf.* on Services Computing (SCC), pp. 253-256, San Francisco, 2018.

[17] **Mohammad Javad Amiri**, Mahnaz Koupaee, *Data-driven Business Process Similarity*, Journal of IET Software 11(6), pp 309-318, 2017.

[18] Mahnaz Koupaee, Mohammad Reza Kangavari, **Mohammad Javad Amiri**, *Scalable Structure-free Data Fusion on Wireless Sensor Networks*, Journal of Supercomputing 73(12), pp 5105-5124, 2017.

[19] **Mohammad Javad Amiri**, Saeed Parsa, Amir Mohammad Zade Lajevardi, *Multifaceted Service Identification: Process, Requirement and Data*, Computer Science and Information Systems 13(2), pp 335-358, 2016.

Abstract

Large-Scale Data Management Using

Permissioned Blockchains

by

Mohammad Javad Amiri

The unique features of blockchain such as transparency, provenance, and authenticity are used by many large-scale data management systems to deploy a wide range of distributed applications including supply chain management, healthcare, and crowdsourcing in a permissioned setting. Unlike permissionless settings, e.g., Bitcoin, where the network is public, and anyone can participate without a specific identity, a permissioned blockchain consists of a set of known, identified nodes that might not fully trust each other. While the characteristics of permissioned blockchains are appealing to a wide range of large-scale data management systems, these systems, have to deal with five important challenges: *confidentiality*, *verifiability*, *performance*, *scalability*, and *fault tolerance*. Confidentiality of data is required in many collaborative large-scale data management applications where collaboration between enterprises, e.g., cross-enterprise transactions, should be *visible* to all enterprises, however, the internal data of each enterprise, e.g, internal transactions, might be *confidential*. Besides confidentiality, in many multi-enterprise systems, e.g., crowdworking environments, participants need to verify transactions that are initiated by other enterprises to ensure some predefined global constraints on the entire system. Thus, the system needs to support *verifiability* while preserving the confidentiality of transactions. Verifiability will gain in importance as crowdworking applications increase in popularity, and the need for regulation will arise. Large-scale data management applications also require high *performance* in terms of throughput and latency. *Scalability*

is one of the main obstacles to business adoption of blockchain systems. To support a large-scale data management application, a blockchain system should be able to scale efficiently by adding more resources to the system. Finally, large-scale data management systems must provide *fault tolerance*. Fault-tolerant protocols are the main building block of large-scale data management systems. However, in spite of years of intensive research, existing fault-tolerant protocols, do not adequately address hybrid environments consisting of trusted and untrusted servers which are widely used by enterprises. In this dissertation, we propose several techniques and develop different systems to address all five main challenges of large-scale data management using permissioned blockchains. We have developed systems, called *CAPER*, *SEPAR*, *ParBlockchain*, *SharPer*, and *SeeMoRe* to deal with the *confidentiality*, *verifiability*, *performance*, *scalability*, and *fault tolerance* requirements of large-scale data management respectively.

Contents

List of Figures

Chapter 1

Introduction

Bitcoin [22] as the first successful global scale peer-to-peer cryptocurrency, allows financial transactions to be transacted among participants without the need for a trusted third party, e.g., banks, credit card companies, or PayPal. Bitcoin eliminates the need for such a trusted third party by replacing it with a distributed data structure (i.e., ledger) that is fully replicated among all participants in the cryptocurrency system [13]. This distributed data structure for recording transactions, which is maintained by several nodes without a central authority, is referred to *blockchain* [23]. In a blockchain system, nodes agree on their shared states across a large network of possibly *untrusted* participants. While blockchain was originally devised for Bitcoin cryptocurrency [22], recently, large-scale data management systems have focused on the features of blockchains such as transparency, provenance, and authenticity to support a wide range of applications. Bitcoin and other cryptocurrencies are *permissionless* blockchains. In a permissionless blockchain, the network is public, and anyone can participate without a specific identity. Many other distributed applications, such as supply chain management and healthcare, are deployed on *permissioned* blockchains consisting of a set of known, identified nodes that still might not fully trust each other. Large-scale data manage-

ment systems deal with five main challenges, *confidentiality*, *verifiability*, *performance*, *scalability*, and *fault tolerance* that need to be supported by permissioned blockchain systems.

Confidentiality. Confidentiality of data is required in many collaborative distributed applications, e.g., supply chain management, where multiple enterprises collaborate with each other following Service Level Agreements (SLAs) to provide different services. To deploy distributed applications across different collaborating enterprises, a blockchain system needs to support the internal transactions of each enterprise as well as cross-enterprise transactions that represent the collaboration between enterprises. While the data accessed by cross-enterprise transactions should be *visible* to all enterprises, the internal data of each enterprise, which are accessed by internal transactions, might be *confidential*. In particular, in collaborative distributed applications where a set of enterprises collaborate with each other, each enterprise can maintain its own independent disjoint blockchain and use techniques such as atomic cross-chain transactions [24] [25] and Interledger protocol [26] to support cross-enterprise collaboration. Such techniques are often costly, complex, and mainly designed for permissionless blockchains. Techniques that support collaborating applications on a single blockchain, on the other hand, either do not support internal transactions of enterprises (results in data integration issues), or suffer from confidentiality issues since the entire ledger is visible to all enterprises, e.g., single-channel Fabric [27], or require a trusted channel among enterprises, e.g., multi-channel Fabric [28]. While cryptographic techniques can be used to achieve confidentiality, the considerable overhead of such techniques makes them impractical [27]. Hyperledger Fabric [27] ensures the confidentiality of data using Private Data Collections [29]. Private Data Collections manage confidential data that two or more enterprises on a single channel want to keep private from other enterprises on that channel. However, the

blockchain ledger is still maintained by the nodes of every enterprise within a channel. To address this problem, we present a permissioned blockchain system, Caper [7]. In Caper, transactions are either internal, which are maintained by a single enterprise, or cross-enterprise, which are maintained by all enterprises. Each enterprise also maintains two types of private and public data. Caper supports both internal and cross-enterprise transactions of collaborating distributed applications. In Caper, each enterprise orders and executes its internal transactions locally while cross-enterprise transactions are public and visible to every node. In addition, the blockchain ledger of Caper is a directed acyclic graph that includes the internal transactions of every enterprise and all cross-enterprise transactions. Nonetheless, for the sake of confidentiality, the blockchain ledger is not maintained by any node. In fact, each enterprise maintains its own *local view* of the ledger including its internal and all cross-enterprise transactions. Since ordering cross-enterprise transactions requires global agreement among all enterprises, Caper introduces different consensus protocols to globally order cross-enterprise transactions.

Verifiability. Besides confidentiality, in many cross-enterprise systems, e.g., crowdsourcing applications, participants need to verify transactions that are initiated by other enterprises to ensure some predefined global constraints on the entire system. Thus, the system needs to support *verifiability* while preserving the confidentiality of transactions. Zero-knowledge proof is used to provide verifiability in ZebraLancer [30] (in the context of crowdsourcing) and Quorum [31]. In cryptography, a zero-knowledge proof is a method by which one party (the prover) can prove to another party (the verifier) that they know a value x, without conveying any information apart from the fact that they know the value x. While the zero-knowledge proof technique satisfies the verifiability property in Quorum and ZebraLancer, first, zero-knowledge proof has a considerable overhead [27], and second, replicating all transactions on every enterprise is costly especially since all

transactions need to be to verified.

The confidentiality technique of CAPER [7] cannot also support all the requirements of such systems, especially since internal transactions might need to be verified as well. For instance, crowdworking platforms need to integrate within society and in particular to interface with legal and social institutions. Global regulations must be enforced, such as minimal and maximal work hours that participants can spend on crowdworking platforms. crowdworking platforms are also naturally distributed and need to collaborate with each other to process complex tasks, resulting in the rise of *multi-platform crowdworking systems*. Moreover, while collaborating to enforce global regulations requires the *transparent* sharing of information about the tasks, the system needs to preserve the *privacy (confidentiality)* of all participants. In this thesis, we present SEPAR [1], a blockchain-based *multi-platform* crowdworking system that enforces global *constraints* on distributed independent entities. In SEPAR, *Privacy* is ensured using *lightweight and anonymous tokens*, while *transparency* is achieved using a *permissioned blockchain* shared across multiple platforms. To support fault tolerance and collaboration among platforms, SEPAR provides a suite of distributed consensus protocols.

Performance. In addition to confidentiality and verifiability, distributed applications, e.g., financial application, require high performance in terms of throughput and latency, e.g., while the Visa payment service is able to handle more than 10000 transactions per second, Multichain [32] can handle at most 200 transactions per second. The order-execute architecture is widely used in different permissioned blockchains such as Tendermint [33], Quorum [31], and Multichain [32]. In order-execute permissioned blockchains, a set of nodes (might be all of them) agrees on a total order for the transactions using Byzantine, e.g., PBFT [21], crash, e.g., Paxos [20], or hybrid, e.g., SeeMoRe [4], fault-tolerant protocols, generates blocks and multicasts them to all the nodes. Each node then

executes the transactions sequentially in the same order and updates its own copy of the ledger. The order-execute architecture, however, suffers from performance issues because of the *sequential execution* of transactions on all nodes. Hyperledger Fabric [27], presents XOV architecture by switching the order of the execution and ordering phases. In Hyperledger Fabric, transactions of different enterprises are first executed in parallel and then ordered using a pluggable consensus protocol. While Fabric improves performance by executing transactions in parallel and supports non-deterministic execution of transactions, in the presence of any contention, i.e., conflicting transactions, in the workload (which is common in distributed applications), it has to disregard the effects of conflicting transactions which negatively impacts the performance of the blockchain. To address this issue, we present ParBlockchain [8]. ParBlockchain introduces the OXII architecture to support contentious workloads. In ParBlockchain, a disjoint set of nodes (orderers) establishes agreement on the order of the transactions of different enterprises, constructs the blocks of transactions, and generates a *dependency graph* for the transactions within a block. A dependency graph gives a partial order based on the conflicts between transactions and enables the parallel execution of non-conflicting transactions. The transactions are then executed following the generated dependency graph. While ParBlockchain supports contentious workloads, any non-deterministic execution of transactions will decrease its performance.

Scalability. Scalability is one of the main obstacles to business adoption of blockchain systems. To support a large-scale data management system, a blockchain system should be able to scale efficiently by adding more resources to the system. The scalability of blockchain systems has been addressed in several studies using different on-chain, e.g., increasing the block size, and off-chain, e.g., Lightning Networks [34], techniques. Partitioning the data into multiple shards that are maintained by different subsets of

nodes is a proven approach to enhance the scalability of databases [35]. In such an approach, the performance of the database scales linearly with the number of nodes. While database systems use the sharding technique to improve the scalability of databases [35] in a network of crash-only nodes, the technique cannot easily be utilized by blockchain systems due to the existence of malicious nodes in the network. In this thesis, we present SharPer [2] [10], a permissioned blockchain system that improves scalability by clustering (partitioning) the nodes and assigning different data shards to different clusters where each data shard is replicated on the nodes of a cluster. SharPer supports both intra-shard and cross-shard transactions and processes intra-shard transactions of different clusters as well as cross-shard transactions with non-overlapping clusters simultaneously. In SharPer, the blockchain ledger is formed as a directed acyclic graph where each cluster maintains *only* a view of the ledger. SharPer also incorporates a *flattened* protocol to establish consensus among clusters on the order of cross-shard transactions.

Fault Tolerance. Finally, large-scale data management systems must provide fault tolerance. Fault-tolerant protocols are the main building block of permissioned blockchain systems and have been extensively used in the distributed database infrastructure of large enterprises such as Google's Spanner [35], Amazon's Dynamo [36], and Facebook's Tao [37]. However, and in spite of years of intensive research, existing fault-tolerant protocols do not adequately address hybrid environments consisting of trusted and untrusted clusters, e.g., clouds, which are widely used by enterprises. On one hand, the existing Byzantine fault-tolerant protocols [21] [38] [39] [40] [41] [42] [43] do not distinguish between crash and malicious failures, and consider all failures as malicious, thus incurring a higher cost in terms of performance. On the other hand, the hybrid protocols [44] [45] that have been designed to tolerate both crash and malicious failures, make no assumption on *where* the crash or malicious failures may occur. As a result, using

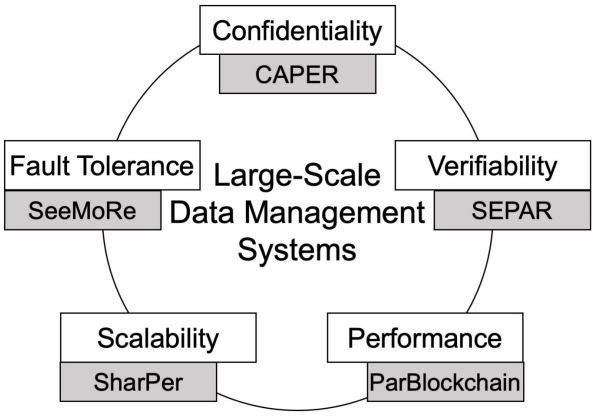


Figure 1.1: Five main challenges and the proposed permissioned blockchain systems

these protocols in a hybrid environment, where all machines in the private environment are known to be trusted while machines in the public environment could be compromised and hence malicious, results in an unnecessary performance overhead. we consider a trusted (i.e., private) environments consisting of non-malicious nodes (crash-only failures) and an untrusted (i.e., public) environments with possible malicious failures. We introduce SeeMoRe [4][1], a hybrid State Machine Replication protocol that uses the knowledge of *where* crash and malicious failures may occur in a trusted/untrusted environment to improve overall performance. SeeMoRe has three different modes that can be used depending on the trusted environment load and the communication latency between the untrusted and trusted environments. SeeMoRe can dynamically transition from one mode to another.

The key contribution of this dissertation is to show how permissioned blockchains can be used to address the main challenges of large-scale data management systems. This dissertation aims to provide an understanding of the fundamental challenges of large-scale data management and the techniques that will help in building efficient large-scale

[1]SeeMoRe is derived from Seemorq, a benevolent, mythical bird in Persian mythology which appears as a peacock with the head of a dog and the claws of a lion. Seemorq in Persian literature also refers to a group of birds who flew together to achieve a common goal.

data management systems. We demonstrate the practicality of the techniques in real-life applications and develop different systems to implement these techniques. Figure 1.1 shows the five main challenges of large-scale data management and the proposed systems to address these challenges.

The rest of the dissertation is organized as follows. Chapter 2 introduces the preliminary concepts of data management and blockchains. Then, the dissertation addresses the five challenges of large-scale data management systems, confidentiality, verifiability, performance, scalability, and fault tolerance in chapters 3, 4, 5, 6, and 7 respectively. Chapter 8 presents the related literature on large-scale data management systems and blockchains, and finally, Chapter 9 summarizes the dissertation and discusses future direction in large-scale data management and blockchains.

Chapter 2

Preliminaries

Recent large-scale data management systems have focused on the unique features of blockchain such as immutability, transparency, provenance, and authenticity to support a wide range of distributed applications. In this chapter, we introduce blockchain, an infrastructure for blockchain systems, and show how consensus works in such systems.

2.1 Blockchain

Bitcoin [22] is an example of a global scale peer-to-peer cryptocurrency that integrates many techniques and protocols from cryptography, distributed systems, and databases. In a blockchain, nodes agree on their shared states across a large network of possibly *untrusted* participants. Bitcoin and other cryptocurrencies use *permissionless* blockchains. *Permissionless* blockchains are public and computing nodes without a priori known identities can join or leave the blockchain network at any time. On the other hand, a *permissioned* blockchain uses a network of a priori known and identified computing nodes to manage the blockchain. The main underlying data structure in blockchain systems is the *blockchain ledger*, a scalable fully replicated structure that is shared among all parti-

cipants and guarantees a consistent view of all user transactions by all participants in the system. The blockchain ledger is an append-only data structure recording transactions in the form of a hash chain where each block contains a batch of transactions (could be a single transaction).

2.1.1 Transaction Model

Two main transaction models are used in blockchain systems: UTXO (Unspent Transaction Output) and Account-based. In the UTXO model, which is adopted by Bitcoin [22] and many other cryptocurrencies, each transaction spends output from prior transactions and generates new outputs that can be spent by transactions in the future. For each transaction in the UTXO model, three conditions need to be satisfied: first, the sum of the inputs must be equal or greater than the sum of the outputs, second, every input must be valid and not yet spent, and third, every input requires a valid signature of its owner. UTXO provides a higher level of privacy by allowing users to use new addresses for each transaction.

The Account-based model, which is adopted by Ethereum [46], is similar to the record keeping in a bank. The bank tracks how much money each account has, and when users want to spend money, the bank makes sure that they have enough balance in their account before approving the transaction. The account-based model is more efficient since the system only needs to validate that the account has enough balance to pay for the transaction.

2.1.2 Smart Contract

The blockchain model is similar to an object-oriented programming language (OOPL). Similar to the primitive data types, user-defined functions, and classes in an OOPL, each

blockchain also has primitive data types (e.g., an asset, asset ownership, etc) and primitive functions operating on these primitive data types (e.g., transactions that move currency units from one user identity to another). Classes and complex functionalities are implemented in the blockchain using smart contracts. A *smart contract*, as exemplified by Ethereum [47], is a computer program that self-executes once it is established and deployed. A smart contract can be seen as a class in an object-oriented programming language where assets are the objects of that class and transactions update the state (ownership) of the objects. The state transformation of a smart contract is made persistent in the blockchain by ensuring that every state change appears as a record in the blockchain. Smart contracts have the advantages of supporting real-time updates, accurate execution, and little human intervention. Smart contracts can be written in different languages such as Solidity and Vyper. A smart contract can also handle automatic conditional payments from escrow. When a payment function is triggered, the smart contract automatically checks the defined conditions, and transfers the money according to the defined rules [48].

While permissionless blockchains only support cryptocurrency assets, smart contracts are more generic and can support any type of asset. Indeed, a smart contract, like a class in the object-oriented programming, could potentially have different attributes and functions. Once a smart contract is written, it can be deployed on a blockchain and different transactions can call the functions of the smart contract to change its attributes or even destroy the contract (using a destructor function), making it void [12].

2.2 Infrastructure

The Blockchain Architecture consists of a set of nodes in an asynchronous distributed system. Nodes in the system might crash (follow a crash failure model) or behave

maliciously (follow a Byzantine failure model). In a crash failure model, nodes operate at arbitrary speed, may fail by stopping, and may restart, however they may not collude, lie, or otherwise attempt to subvert the protocol. Whereas, in a Byzantine failure model, faulty nodes may exhibit arbitrary, potentially malicious, behavior.

Nodes are connected by point-to-point bi-directional communication channels. Network channels are pairwise authenticated, which guarantees that a malicious node cannot forge a message from a correct node, i.e., if node i receives a message m in the incoming link from node j, then node j must have sent message m to i beforehand.

Furthermore, messages may contain public-key signatures and message digests [21]. A *message digest* is a numeric representation of the contents of a message produced by collision-resistant hash functions. Message digests are used to protect the integrity of a message and detect changes and alterations to any part of the message. We denote a message m signed by replica r as $\langle m \rangle_{\sigma_r}$ and the digest of a message m by $D(m)$. For signature verification, we assume that all machines have access to the public keys of all other machines. We assume that a strong adversary can coordinate malicious nodes and delay communication to compromise the replicated service. However, the adversary cannot subvert standard cryptographic assumptions about collision-resistant hashes, encryption, and signatures, e.g., the adversary cannot produce a valid signature of a non-faulty node.

2.3 Consensus

A fundamental problem in distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a number of faulty processes. This requires consensus among a number of processes (or agents) for a single data value. Some of the processes (agents) may fail or be unreliable in other ways, so consensus

protocols must be fault tolerant or resilient. *Synchronous* distributed systems assume known bounds on message delays and process speeds [49]. In synchronous systems, all communication proceeds in rounds. In one round, a process may send all the messages it requires, while receiving all messages from other processes. In this manner, no message from one round may influence any messages sent within the same round. On the other hand, in *asynchronous* distributed systems, there are no bounds on the amount of time a node might take to complete its work and then respond with a message [49]. In such systems, there is no global clock nor consistent clock rate, each node processes independently of others, and coordination is achieved via events such as message arrival [5].

In a permissioned blockchain system, nodes establish consensus on a unique order in which entries are appended to the blockchain ledger using asynchronous fault-tolerant protocols. Fault-tolerant protocols use the State Machine Replication (SMR) algorithm [50] where nodes agree on an ordering of incoming requests. SMR regulates the deterministic execution of client requests on multiple copies of a server, called replicas, such that every non-faulty replica must execute every request in the same order [51] [50]. The SMR algorithm has to satisfy four main properties [52]: (1) *Agreement:* every correct node must agree on the same value, (2) *Validity (integrity):* if a correct node commits a value, then the value must have been proposed by some correct node, (3) *Consistency (total order):* all correct nodes commit the same value in the same order, and (4) *Termination:* eventually every node commits some value. The first three properties are known as *safety* and the termination property is known as *liveness*. As shown by Fischer et al. [53], in an asynchronous system, where nodes can fail, consensus has no solution that is both safe and live. Based on that impossibility result, in most fault-tolerant protocols, safety is ensured in an asynchronous network that can drop, delay, corrupt, duplicate, or reorder messages. However, a synchrony assumption, i.e., a finite but possibly unknown bound on message delivery time is needed to ensure liveness.

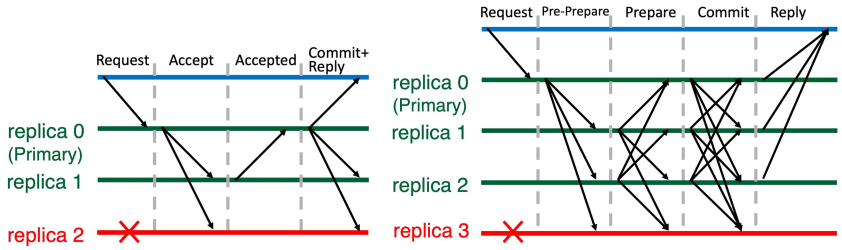


Figure 2.1: Normal case operation in (a) Paxos [20] and (b) PBFT [21]

Crash fault-tolerant protocols, e.g., Paxos [20], guarantee safety in an asynchronous network using $2f+1$ nodes to overcome the simultaneous crash failure of any f nodes while in Byzantine fault-tolerant protocols, e.g., PBFT [54], $3f+1$ nodes are needed to provide safety in the presence of f malicious nodes [55].

We now briefly introduce Paxos [20] and PBFT [21] as two well-known crash and Byzantine fault-tolerant protocols.

In Paxos [20], which guarantees safety in an asynchronous network using $2f+1$ nodes, upon receiving a request from a client, the primary (assuming it is already elected) initiates a consensus protocol among the agents of the enterprises by multicasting an accept message including the transaction. Once an agent receives an accept message, it sends an accepted message to the primary. The primary waits for f accepted messages from different agents (plus itself becomes $f+1$), multicasts a commit message to all the agents, and sends a reply to the client. Upon receiving a commit message from the primary, each agent executes the transaction. Figure 2.1(a) shows the normal case operation of Paxos protocol.

In the presence of malicious nodes, PBFT [21] can be used. In PBFT, which guarantees safety in an asynchronous network using $3f+1$ nodes, during a normal case execution, a client sends a request to a (primary) replica, and the primary broadcasts a pre-prepare message to all replicas. Once a replica receives a valid pre-prepare message, it broadcasts a prepare message to all other replicas. Upon collecting $2f$ valid matching prepare messages

(including its own message) that are also matched to the pre-prepare message sent by the primary, each replica broadcasts a commit message. In this stage, each replica knows that all non-faulty replicas agree on the contents of the message sent by the primary. Once a replica receives $2f + 1$ valid matching commit messages (including its own message), it executes the request and sends the response back to the client. Finally, the client waits for $f+1$ valid matching responses from different replicas to make sure at least one correct replica executed its request. PBFT also has a view change routine that provides liveness by allowing the system to make progress when the primary fails. Figure 2.1(b) shows the normal case operation of PBFT protocol.

Chapter 3

CAPER: On Confidentiality of Permissioned Blockchains

3.1 Introduction

Confidentiality of data is required in many collaborative distributed applications, e.g., supply chain management, where multiple enterprises collaborate following Service Level Agreements (SLAs), which are agreed upon by all involved enterprises, to provide different services. SLAs can be written as self executing computer programs, called *smart contracts* [46]. The collaboration is realized by means of cross-enterprise transactions that are *visible* to every enterprise. During the execution of cross-enterprise transactions, agreement on the shared state of the collaborating enterprises is needed without trusting a central authority or any particular participant. While cross-enterprise collaborations and the involved data are *visible* to every enterprise, the internal data of each enterprise, i.e., the enterprise logic, internal transactions, and their data, might be *confidential*. Hence, it is desirable to restrict access to such data. Although cryptographic techniques can be used to achieve confidentiality, the high overhead of such techniques makes them

impractical [27].

The unique features of blockchain such as transparency, provenance, fault tolerance, and authenticity are used by many systems to deploy a wide range of large-scale data management applications such as supply chain management [56] and healthcare [57] in a permissioned setting. Existing permissioned blockchains, however mostly suffer from confidentiality issues since a single blockchain ledger with all transactions is maintained at every node. While Hyperledger Fabric [27] addresses the confidentiality leaks by restricting access to the blockchain state (using private data collections) and *smart contracts*. which include the enterprise logic, the blockchain ledger is still maintained by every node. To provide confidentiality, different enterprises could have independent disjoint blockchains. However, to support collaboration between enterprises on distinct blockchains, techniques such as atomic cross-chain transaction [24] [25] and Interledger protocol [26] are needed to exchange (transfer) assets or information between the blockchains (*Interoperability*). Such techniques are often costly, complex, and mainly designed for permissionless blockchains. Techniques that support collaborating enterprises on a single blockchain either do not support internal transactions of enterprises [48] (results in data integration issues), or suffer from confidentiality issues since the entire ledger is visible to all enterprises, e.g., single-channel Fabric [27], or require a trusted channel among participants, e.g., multi-channel Fabric [28].

In this chapter, we present *CAPER*: a permissioned blockchain system that supports both internal and cross-enterprise transactions of collaborating distributed enterprises. In CAPER, each enterprise orders and executes its internal transactions locally while cross-enterprise transactions are public and visible to every enterprise. In addition, the blockchain ledger of CAPER is a directed acyclic graph that includes the internal transactions of every enterprise and all cross-enterprise transactions. Nonetheless, for the sake of confidentiality, the blockchain ledger is *not maintained* by any node. In fact,

each enterprise maintains its own *local view* of the ledger including its internal and all cross-enterprise transactions. Since ordering cross-enterprise transactions requires *global* agreement among all enterprises, CAPER introduces different consensus protocols to globally order cross-enterprise transactions.

Lack of trust is an important problem in collaboration between enterprises. Lack of trust has two main origins: first, a *physical node* (server) might fail, thus behave maliciously, and second, an *enterprise* could behave maliciously in the communication with other enterprises for its benefits. To address both types of behavior, CAPER distinguishes between trust at the node level and trust at the enterprise level, e.g., while the nodes of an enterprise might behave non-maliciously within the enterprise, the enterprise (as a collection of nodes) might still behave maliciously in communication with other enterprises.

A key objective of this chapter is to demonstrate how internal and cross-enterprise transactions of a set of collaborating distributed enterprises which do not trust each other can be processed efficiently by a blockchain system while both confidentiality of internal transactions and transparency of cross-enterprise transactions are met. The contributions of this chapter are three-fold:

- Introducing *Blockchain views* where each enterprise maintains *only* its own *view* of the ledger including its internal and all cross-enterprise transactions.
- CAPER, a permissioned blockchain that supports collaborating distributed enterprises. CAPER supports both internal and cross-enterprise transactions.
- Three different consensus protocols for globally ordering cross-enterprise transactions among enterprises with different local consensus protocols.

The rest of this chapter is organized as follows. Section 3.2 briefly describes the limitations of current blockchain systems and motivates the problem. The CAPER model

and architecture are introduced in Section 3.3 and Section 3.4. Section 3.5 and Section 3.6 present consensus in CAPER. Section 3.7 presents a performance evaluation of CAPER, and Section 3.8 concludes the chapter.

3.2 Background and Motivation

In this section, we discuss the confidentiality limitations of permissioned blockchains and provide a motivating example to explain the limitations.

3.2.1 Confidentiality Limitations of Permissioned Blockchain

Data confidentiality is required in many permissioned blockchains. A blockchain might need to restrict access to smart contracts (which include the logic of enterprises), blockchain ledger, and transaction data. While cryptographic techniques can be used to achieve confidentiality, the considerable overhead of such techniques makes them impractical [27]. Hyperledger Fabric [27] ensures the confidentiality of data using Private Data Collections [29]. Private Data Collections manage confidential data that two or more enterprises on a single channel want to keep private from other enterprises on that channel. The confidentiality of smart contracts in Fabric is also ensured by storing the smart contracts of different enterprises on different sets of nodes, called *endorsers*. The endorsers of an enterprise are responsible for executing the transactions of the enterprise independent of the other enterprises. However, the blockchain ledger is still maintained by the agents of every enterprise within a channel. Enterprises need to collaborate with each other to provide different services. Distributed applications are often designed and implemented in different blockchain systems, each of which processes and stores data independently [58]. In this case, cross-enterprise collaboration could be performed as atomic cross-chain transactions [24] [25] or using the Interledger protocol [26] where two enter-

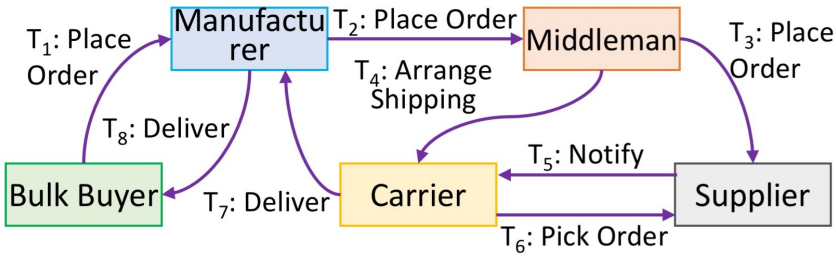


Figure 3.1: A supply chain scenario

prises (parties) exchange (transfer) assets or information across their blockchains. Atomic cross-chain transactions are also addressed between permissioned blockchains (channels) in Fabric by either assuming the existence of a trusted channel among the participants or using an atomic commit protocol [28]. In general, supporting cross-enterprise transactions is an expensive and challenging task. Nonetheless, for collaborating enterprises in a single permissioned blockchain, since the participants of cross-enterprise transactions are known beforehand and the transactions follow service level agreements, we might be able to find a solution that preserves both confidentiality and performance.

3.2.2 A Motivating Example: Supply Chain

We now consider collaborative workflows as a use-case to illustrate the aforementioned limitations. In a collaborative workflow, different parties need to communicate across enterprises to provide services. However, the lack of trust between parties is problematic.

Figure 3.1 shows a Supply Chain scenario (reported in [48]). The workflow involves five participants (enterprises): Supplier, Manufacturer, Bulk Buyer, Carrier, and Middleman where each of the participants might have multiple trusted or untrusted nodes that perform different internal tasks (transactions). For example, the production process in the Manufacturer involves Financial, Marketing, and Purchasing departments and includes different internal tasks such as assembly, painting, drying, testing, and packaging.

Participants also need to communicate with each other to provide different services. The Bulk Buyer communicates with the Manufacturer to place an order, the Manufacturer places an order for materials via a Middleman, the Middleman forwards the order to a Supplier and arranges transportation by a Carrier. Once the materials are acquired, the Supplier informs the Carrier and the Carrier picks them up and delivers them to the Manufacturer. As soon as the goods become available, the Manufacturer delivers them to the Bulk Buyer. These collaborations are defined in service level agreements which are agreed upon by all participants.

Now the Manufacturer might receive the materials later than agreed upon or might receive something different from what they agreed on. In this case, the Supplier might argue that this is exactly what was ordered by the Middleman while the Middleman would blame the Supplier. The situation is complicated for the Carrier since the Manufacturer might refuse to accept the delivery. The Carrier is now eligible for compensation from either the Supplier or the Middleman depending on who is responsible for the fault.

To tackle such an issue, *permissioned* blockchain systems can be used among all the different participants to ensure agreement on the shared state of the collaborating parties without trusting a central authority or any particular participant [48]. The blockchain basically *monitors* the execution of the collaborative process and *checks conformance* between the process execution and SLAs. Any blockchain-based solution has to address the following concerns.

First, the blockchain system should support both cross-enterprise and internal transactions. For example, in the Supply Chain scenario, in addition to cross-enterprise transactions, the Manufacturer might want to use the blockchain for its internal transactions, e.g., calculating materials demand or testing the product, to benefit from the the unique features of blockchain. Second, in contrast to the cross-enterprise transactions which are public and can be accessed by all participants, the internal transactions of each enterprise

and their data should only be accessed by the nodes of the enterprise to preserve confidentiality, e.g., the internal transactions of the Manufacturer show its internal process for producing a product which the Manufacturer might intend to keep as a secret. Third, the solution has to address the performance aspect as well.

One possible solution is to implement all enterprises within a single blockchain where all the transactions are maintained in a single blockchain ledger which is replicated among all the nodes in the blockchain. This solution handles the cross-enterprise transactions efficiently because every node accesses all the data. However, since the ledger is replicated among all the nodes and every transaction is visible to all enterprises, the confidentiality of data is not preserved.

Another solution is to implement each enterprise on a separate blockchain. In that way, participants can perform their internal transactions in parallel resulting in higher performance and since their data is maintained on different blockchains, the confidentiality of data is preserved. To perform communication between different enterprises, one approach is to use atomic cross-chain operation, which, as discussed earlier, is expensive. An alternative approach is to use a new blockchain to maintain public transactions. However, since public transactions use data which is provided by internal transactions, e.g., to place an order, the Manufacturer needs to calculate demand internally, and these two types of transactions are stored in different blockchains, *data integration* becomes an expensive and challenging task.

In this chapter, we present a new approach that not only addresses the performance and confidentiality issues, but also handles cross-enterprise transactions efficiently.

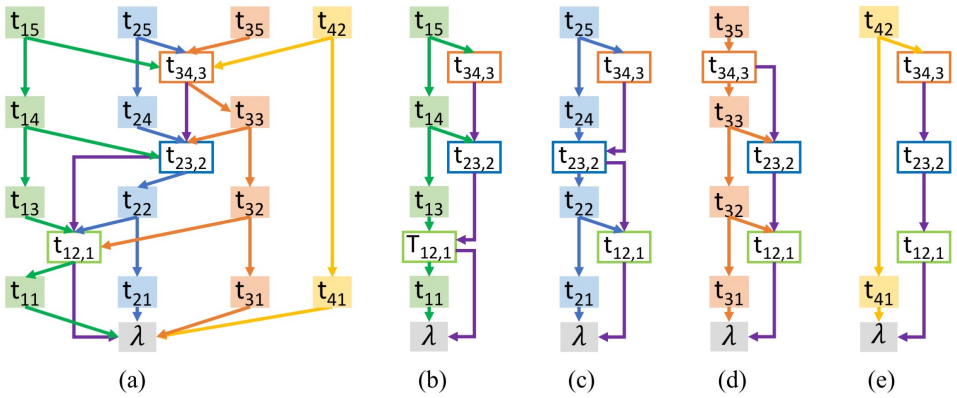


Figure 3.2: (a): A blockchain ledger, (b)-(e): Different views of the blockchain

3.3 The CAPER Model

In this section, the CAPER model is introduced. We first present distributed applications and the blockchain ledger, and then show how the distributed applications are deployed in the blockchain.

3.3.1 Distributed Applications

CAPER is a blockchain system designed to support distributed applications consisting of a set of collaborating enterprises which might not trust each other. Each enterprise maintains two sets of *private* and *public* records. The private records of an enterprise are accessible only to the enterprise whereas the public records are replicated on all enterprises.

CAPER supports internal and cross-enterprise transactions. *Internal* transactions are performed within an enterprise following the logic of the enterprise, e.g., in the Supply Chain scenario, the Manufacturer calculates materials demand internally. Internal transactions of an enterprise can read and write its private records, however they can only read (and not write) the public records. *cross-enterprise (public)* transactions, on the other

hand, involve multiple enterprises and are visible to all enterprises, e.g., the Manufacturer places an order for materials via a Middleman. cross-enterprise transactions follow the service level agreements (SLAs) between the involved enterprises. SLAs present the flow of communication between the enterprises and indicate different aspects of the services, e.g., quality, availability, and responsibilities, that should be provided by different enterprises. For example, in the Supply Chain scenario, the Carrier is responsible for delivering requested materials to the Manufacturer in two business days from the date it is informed by the Supplier. Public records can only be updated via cross-enterprise transactions.

3.3.2 Blockchain Ledger

The blockchain ledger in CAPER consists of both internal transactions of all enterprises and all cross-enterprise transactions in the system where each block consists of a single transaction. To support both types of transactions, we generalize the notion of a blockchain ledger from a linear chain to a *directed acyclic graph (DAG)* where *nodes* of the graph are transactions and *edges* enforce the order of transactions.

Within an enterprise, since transactions have access to the same datastore, a total order between the transactions that are initiated by the enterprise is enforced to ensure consistency. To present the total order of transactions in the blockchain ledger, transactions are *chained* together, i.e., each transaction includes the cryptographic hash of the previous transaction. In addition, since cross-enterprise transactions update data which is replicated on all the enterprises, to ensure consistency, cross-enterprise transactions are totally ordered as well. Furthermore, internal transactions of enterprises might use data that is provided by cross-enterprise transactions, e.g., the Manufacturer calculates materials demand based on the place-order cross-enterprise transaction of the Bulk Buyer.

To show such data dependencies, an internal transaction includes the cryptographic hash of a cross-enterprise transaction in the ledger.

In summary, the blockchain ledger has three properties: (1) There is a total order between all transactions (internal as well as cross-enterprise) that are initiated by an enterprise, (2) There is a total order between cross-enterprise transactions, and (3) An internal transaction of an enterprise might include the cryptographic hash of a cross-enterprise transaction (that is initiated by another enterprise).

In addition to internal and cross-enterprise transactions, a unique initialization transaction (block), called *genesis* transaction, is considered for the blockchain. Function $H(.)$ also denotes the cryptographic hash function. For simplicity, to show that transaction t includes $H(t')$ (i.e. t is ordered immediately after t' as explained in properties 1 and 2 or has data dependency to t' as explained in property 3) we include an *edge* from t to t' in the DAG representation of the blockchain ledger.

Fig. 3.2(a) shows a CAPER blockchain ledger consisting of four enterprises α_1, α_2, α_3, and α_4. In this figure, λ is the genesis transaction. Internal and cross-enterprise transactions of each enterprise are also specified. For example, t_{11}, t_{13}, t_{14}, and t_{15} are the internal transactions of enterprise α_1, and $t_{12,1}$, $t_{23,2}$, and $t_{34,3}$ are the cross-enterprise transactions initiated by α_1, α_2, and α_3 respectively. Note that each cross-enterprise transaction is labeled with $t_{i,j}$ where i indicates the order of the transaction among the transactions that are initiated by its initiator enterprise and j presents the order of the transaction among all cross-enterprise transactions. As can be seen, transactions (both internal and cross-enterprise) that are initiated by an enterprise are chained together (property 1), e.g. t_{31}, t_{32}, t_{33}, $t_{34,3}$, and t_{35}. In addition, cross-enterprise transactions are chained together (property 2), i.e., $t_{12,1}$, $t_{23,2}$, and $t_{34,3}$. Finally, an internal transaction might include the hash of a cross-enterprise transaction that shows the data dependency of the internal transaction to the cross-enterprise transaction (property 3), e.g., internal

transactions t_{22} of α_2 has edge to cross-enterprise transaction $t_{12,1}$.

Note that, since an edge from transaction t to transaction t' indicates that t occurs after t' (t includes the hash of t', thus t' has to be appended to the ledger earlier), it is easy to show that the resulting graph is acyclic.

In contrast to the cross-enterprise transactions that are visible to and maintained by all enterprises, the internal transactions of an enterprise present confidential data about the enterprise, e.g., its business logic. For example, in the Supply Chain scenario, the internal transactions of the Manufacturer show its internal process for producing a product which the Manufacturer might intend to keep as a secret. The presented blockchain ledger, however, is at odds with confidentiality because every enterprise has access to every transaction. For the sake of confidentiality, we want to prohibit an enterprise from observing the internal transactions of other enterprises. To achieve this, in CAPER, the entire blockchain ledger is *not maintained* by any enterprise. In fact, each enterprise only maintains its own *view* of the blockchain ledger that includes its internal transactions and all the cross-enterprise transactions. The blockchain ledger is indeed the union of all these physical views.

Fig. 3.2(b)-(e) show the views of the blockchain ledger for enterprises α_1, α_2, α_3, and α_4 respectively where each enterprise maintains only the part of the ledger consisting of its internal and all the cross-enterprise transactions.

3.3.3 Application Deployment on a Blockchain

Each enterprise in CAPER, in addition to the datastore and its view of the blockchain ledger, maintains a "private smart contract" to implement the enterprise logic, and a "public smart contract" to implement the logic of cross-enterprise transactions.

Each enterprise in CAPER has its own *private smart contract* which includes the logic

of the internal transactions. In addition, a *public smart contract* is written to include the logic of cross-enterprise transactions which is determined by the service level agreements between the enterprises. As discussed in Section 3.2, to execute a collaborative process, participants agree on SLAs which are then written in the public smart contracts. The public smart contract runs on every enterprise to check if the cross-enterprise transactions are *conforming* to the SLAs, and enforce the conditions defined in the cross-enterprise transactions. In contrast to most existing blockchains where every smart contract runs on all nodes, which is at odds with confidentiality, in CAPER, each private smart contract runs only on its enterprise. To support cross-enterprise transactions, however, the public smart contract runs on every enterprise. Furthermore, both private and public smart contracts can be written in domain-specific languages, e.g., Solidity, to ensure the deterministic execution of transactions.

In the Supply Chain scenario in Figure 3.1, each of the five involving enterprises, i.e., Supplier, Manufacturer, Bulk Buyer, Carrier, and Middleman, executes its internal transactions following the enterprise logic, which is implemented in its private smart contract. Once a cross-enterprise transaction is requested, e.g., the Bulk Buyer places an order with the Manufacturer, every enterprise executes and appends the transaction to its view of the ledger. To execute the cross-enterprise transactions, the public smart contract is used which includes the SLAs (that are agreed upon by all enterprises). Hence, if the requested transaction does not conform to the SLAs, it will be detected. For each cross-enterprise transaction the SLA defines several conditions to check. SLA also includes actions for non-conforming transactions, e.g., if the Carrier delivers the materials later than agreed upon, it will be penalized as specified in the SLA or if the delivered materials are something different from what they agreed on, the transaction will be aborted.

3.4 The CAPER Architecture

CAPER consists of a set of nodes in an asynchronous distributed system where each enterprise runs on a (non-empty) disjoint subset of nodes called the *agents* of the enterprise. We use N and A to denote the set of nodes and enterprises. In addition, N_α indicates the set of agents of enterprise $\alpha \in A$ where for each pair of enterprises α_1 and α_2 in A, $N_{\alpha_1} \cap N_{\alpha_2} = \varnothing$.

Nodes in CAPER might crash, behave maliciously, or be reliable. In addition, we assume that enterprises do not trust each other, thus we model enterprise failures as Byzantine failures. In fact, we define two levels of behavior in the system. First, at the node level, each agent might be a crash-only, a Byzantine, or a reliable node. In a crash failure model, nodes operate at arbitrary speed, may fail by stopping, and may restart. Whereas, in a Byzantine failure model, faulty nodes may exhibit arbitrary, potentially malicious, behavior. A reliable node, on the other hand, never fails. Second, at the enterprise level, an enterprise (as a group of agents) might behave maliciously. Note that these two levels of behavior are independent of each other. Thus, even if the agents of an enterprise are crash-only nodes, the enterprise might still behave maliciously.

Ordering the transactions within each enterprise needs consensus among the agents of the enterprise. To establish consensus among the agents of an enterprise, Paxos [20] and PBFT [21] can be used. To establish consensus for cross-enterprise transactions, since enterprises may not trust each other, a Byzantine fault-tolerant protocol among enterprises is needed. To provide both safety and liveness for consensus at the enterprise level, we assume that at most $\lfloor \frac{|A|-1}{3} \rfloor$ enterprises might be malicious. As a result, to commit a cross-enterprise transaction, by a similar argument as in PBFT [21], at least *two-thirds* ($\lfloor \frac{2|A|}{3} \rfloor + 1$) of the enterprises *including* the initiator enterprise of each cross-enterprise transaction must agree on the order of the transaction. We need agreement

from agents of the initiator enterprise to ensure that the cross-enterprise transaction is consistent with the internal transactions of the initiator enterprise. The conformance between a cross-enterprise transaction and the SLAs is checked by every enterprise during the execution of the transaction, thus, if the initiator enterprise initiates a transaction that does not conform to the SLAs, it will be detected by other enterprises during the execution.

3.5 Local Consensus

In this section, we show how internal transactions are ordered and executed in CAPER. CAPER employs *local* consensus within an enterprise to order transactions where the agents of an enterprise, *independent* of other nodes in the network, agree on the order of transactions.

The local consensus protocols in CAPER are pluggable. Depending on the failure model of nodes (agents), the enterprise can use a crash fault-tolerant protocol, e.g., Paxos [20], or a Byzantine fault-tolerant protocol, e.g., PBFT [54], as the local consensus protocol. The enterprise might not even use a consensus protocol and rely on a single non-faulty reliable node to order the transactions. The number of required agents is also determined by the protocol and the maximum number of simultaneous failures in the network.

The local consensus protocol to order the internal transactions of an enterprise is initiated by one of the agents, called *the primary*. The normal case operation for CAPER to execute an internal transaction proceeds as follows. A client c requests an internal transaction tx for an enterprise by sending a message $\langle \textsf{REQUEST}, tx, \tau_c, c \rangle_{\sigma_c}$ to the agent p of the enterprise it believes to be the primary. Here, τ_c is the client's timestamp and the entire message is signed with signature σ_c. The timestamps of clients are used to totally

order the requests of each client and to ensure exactly-once semantics for the execution of client requests.

When the primary p receives a request from a client, it first checks the signature to ensure it is valid, and then initiates a local consensus algorithm by multicasting a message, e.g., accept message in Paxos or pre-prepare message in PBFT, including the requested transaction to other agents. To provide a total order between transactions, the primary also includes $H(t)$ in the message where $H(.)$ denotes the cryptographic hash function and t is the previous transaction that is ordered by the enterprise. If the transaction has a data dependency to a cross-enterprise transaction (as discussed in Section 3.3.2), the primary includes the cryptographic hash of the cross-enterprise transaction as well.

The agents then establish agreement on a total order of transactions using the utilized consensus protocol, execute the transaction, and append it to the blockchain ledger. Finally, either the primary or every agent node (depending on the local consensus protocol) sends a reply message $\langle \text{REPLY}, \tau_c, u \rangle_{\sigma_o}$ to client c where ts_c is the timestamp of the corresponding request and u is the execution result.

3.6 Global Consensus

In this section, we show how cross-enterprise transactions are ordered and executed in CAPER using *global* consensus among enterprises. We introduce three ordering approaches for achieving global consensus in CAPER.

Ordering transactions using a disjoint set of nodes was introduced by Hyperledger [27] to enhance the scalability of the system and to add flexibility for implementing the consensus protocol, i.e., different protocols can be used to establish consensus. Similarly, in the first approach of CAPER, a disjoint set of nodes, called *orderers*, which are not the agents of any enterprise, are used to globally order cross-enterprise transactions. The

global consensus protocol among orderers is pluggable and depending on the specifications of the system, a crash, a Byzantine, or any other fault-tolerant protocol can be used.

In the absence of such orderer nodes, and in the second approach, CAPER relies on the enterprises to order cross-enterprise transactions in a hierarchical way. To distinguish between trust at the node level and trust at the enterprise level, agreement is established in two levels: a local level among the agents of each enterprise, and a global level among the enterprises of the system.

Although the second approach does not require a set of orderers to order transactions, the hierarchical nature of the algorithm, which needs local consensus within each enterprise for each step of global ordering, makes the protocol expensive. In the third approach, similar to the second approach, the agents of all enterprises order cross-enterprise transactions, however, agreement is established in one level.

In all three approaches, when agreement is achieved, the agents execute the transaction and append it to their local views of the ledger. Note that if the transaction does not follow the service level agreements, which are implemented in the public smart contract, it will be detected during the execution of the transaction (as discussed in Section 3.3.3).

3.6.1 Global Consensus using a Separate Set of Orderers

Using a separate set of nodes to order transactions adds flexibility to the system by allowing the global consensus implementation to be tailored to the trust assumption of a particular deployment. In addition, since the orderers are decoupled from the agents that execute transactions and maintain the blockchain ledger, using orderers enhances the scalability of the system.

In the first approach, CAPER orders the cross-enterprise transactions using a disjoint set of *orderers* O where for each enterprise α in A, $O \cap N_\alpha = \varnothing$. As discussed

Algorithm 1 Global Consensus using Orderers

```
 1: init():
 2:    r := node_id
 3:    O := the set of orderer nodes
 4:    p := the primary agent of the initiator enterprise
 5:    o := the primary agent of the orderers

 6: upon receiving m=⟨REQUEST, tx, τc, c⟩σc and (r == p):
 7:    if m is valid then
 8:       initiate local consensus
 9:       if agreement is achieved then
10:          send ⟨⟨ORDER, hL, d, r⟩σr, m⟩ to o {either primary or every agent r of the initiator enterprise}

11: upon receiving valid ⟨⟨ORDER, hL, d, r⟩σr, m⟩ from the sufficient number of agents and node is the primary orderer o
12:       initiate a global consensus among orderers O
13:       if agreement is achieved then
14:          multicast ⟨SYNC, hL, hG, d, o⟩σo, m⟩ {o or every orderer }

15: upon receiving ⟨SYNC, hL, hG, d, o⟩σo, m⟩ from the primary orderer (or a sufficient number of orderers)
16:    execute and append the transaction to the ledger
```

earlier, a cross-enterprise transaction is ordered *locally* among the transactions that are initiated by the initiator enterprise and *globally* among all cross-enterprise transactions, thus both local and global orderings are needed. Since orderers are not involved in the local consensus and enterprise agents do not participate in the global consensus, these two orderings are separated from each other. As a result, cross-enterprise transactions are first, similar to the internal transactions, ordered *locally* within each enterprise using the enterprise consensus protocol and then ordered *globally* among all cross-enterprise transactions using orderers. Note that the cross-enterprise transactions are ordered first locally and then globally to prevent the case where the agents of the initiator enterprise do not agree on the local order of a transaction that has already been ordered globally. Once orderers agree on the global order of the transaction, they multicast the transaction to all the enterprises, thus, every agent of every enterprise executes and appends the transaction to its ledger.

The normal case operation of *global consensus using orderers* to execute a cross-enterprise transaction is presented in Algorithm 1. Although not explicitly mentioned, every sent and received message is logged by the nodes. As indicated in lines 1-5 of

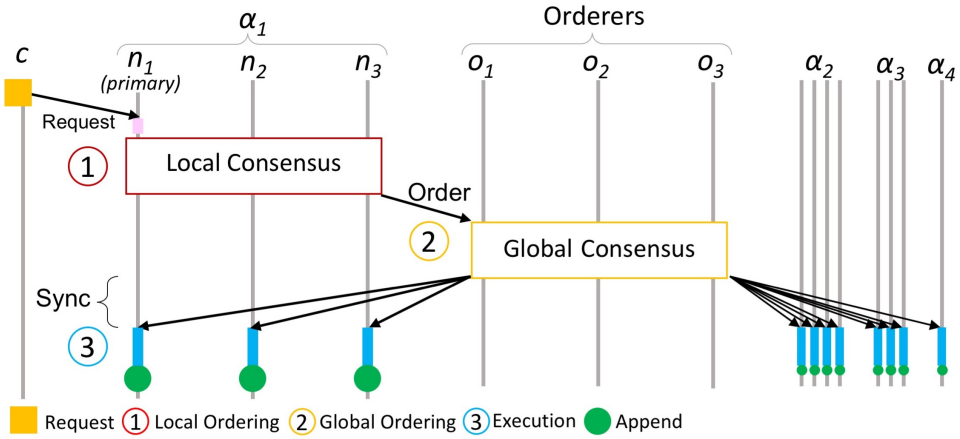


Figure 3.3: Global consensus using a set of orderers

the algorithm, nodes p and o are the primary agent of the initiator enterprise and the primary orderer respectively and O is the set of orderers.

As shown in lines 6-10 of the algorithm, when primary agent p receives a valid request $\langle \text{REQUEST}, tx, \tau_c, c \rangle_{\sigma_c}$ from an authorized client c (with timestamp τ_c) to execute a cross-enterprise transaction tx, it initiates the local consensus protocol to establish agreement on the order of the cross-enterprise transaction among all transactions that are initiated by the enterprise. Once agreement is achieved, depending on the local consensus protocol, either the primary or every agent r of the initiator enterprise sends a signed order message $\langle \langle \text{ORDER}, h_L, d, r \rangle_{\sigma_r}, m \rangle$ including the client's request message m to the primary orderer node where d is m's digest and $h_L = H(t)$ where t is the previous transaction that is ordered by the enterprise. Note that h_L is included in the order messages to let orderers know that the agents of the enterprise agree on the local order of the transaction.

When the order message is sent to the primary orderer, the primary of the enterprise waits for the corresponding reply (sync) message from the orderers before initiating any other transactions. In addition, when local agreement is established, nodes wait for global agreement before appending the transaction to the blockchain ledger.

As shown in lines 11-14, once primary orderer o receives a sufficient number (determ-

ined by the local consensus protocol of the initiator enterprise, i.e., 1 for crash fault-tolerant and $f + 1$ for Byzantine fault-tolerant) of valid matching order messages (with valid signature, matching h_L, and matching message digest), primary orderer o initiates the (global) consensus protocol by multicasting the transaction to other orderers to establish a total order on cross-enterprise transactions. Once the orderers reach agreement on the order of the transaction, depending on the global consensus protocol, either the primary or every orderer node o multicasts a sync message $\langle \text{SYNC}, h_L, h_G, d, o \rangle_{\sigma_o}, m \rangle$ to all the agent of every enterprise where d is the digest of m, h_L is copied from order message of the initiator enterprise, and $h_G = H(t)$ such that t is the previous cross-enterprise transaction.

Upon receiving a sync message, the agents of each enterprise log the message. Once an agent receives a *sufficient* number of matching sync messages (determined by the utilized global consensus protocol), the agent executes the transaction and appends the transaction to its blockchain ledger (as can be seen in lines 15 and 16). Note that the agents of the initiator enterprise consider both h_L and h_G hashes to append the transaction to the ledger while the agents of the other enterprises only consider the hash of the previous cross-enterprise transaction (h_G). Finally, depending on the utilized local consensus protocol, either the primary or every agent node r of the initiator enterprise sends a reply message $\langle \text{REPLY}, \tau_c, u \rangle_{\sigma_r}$ to client c where ts_c is the timestamp of the corresponding request and u is the result of executing the request.

The flow of cross-enterprise transactions using a set of orderers in a blockchain consisting of four enterprises α_1, α_2, α_3, and α_4 can be seen in Figure 3.3. Here enterprise α_1 initiates the cross-enterprise transaction. In addition, o_1, o_2, and o_3 are the orderer nodes. Upon receiving a cross-enterprise transaction from a client, the primary node n_1 of α_1 validates the transaction and similar to internal transactions, initiates a local consensus algorithm to order the transaction within the enterprise. Once the transac-

tion is internally ordered, an order message is sent to the primary orderer node. The orderers use a global consensus protocol, e.g., a crash fault-tolerant protocol with $f = 1$ in Figure 3.3, to agree on the global order of the transaction and then since here the orderers are crash-only nodes, the primary orderer (node o_1) multicasts sync messages including the transaction to every agent of every enterprise. Each agent then validates the transaction, executes the transaction, and appends it to the ledger.

Safety and Liveness. Since the global ordering protocol is pluggable, its safety and liveness are implied due to Paxos [20] and PBFT [21]. The order of cross-enterprise transactions on different enterprises is unique because cross-enterprise transactions are ordered sequentially by orderers and the agents of every enterprise follows the order that is provided by orderers. Note that if the agents of an enterprise do not follow the provided order, the enterprise might not be able to initiate cross-enterprise transactions in the future (since the initiated transactions might not conform to SLAs). To ensure a total order between transactions that are initiated by the same enterprise, once the primary of an enterprise sends a cross-enterprise transaction to be ordered by the orderers, the primary stops initiating any other transactions and waits for the reply from the orderers.

3.6.2 Hierarchical Global Consensus

While using a separate set of orderer nodes makes the agreement routine simple and modular, it comes with an extra cost of adding orderers to the system. In the absence of such orderer nodes, reaching consensus on the order of the cross-enterprise transactions needs the participation of all enterprises. To distinguish between trust at the node level and trust at the enterprise level, CAPER uses an asynchronous Byzantine fault-tolerant protocol for the global consensus where for each cross-enterprise transaction and in each phase of the global consensus, every enterprise runs its local consensus protocol between

Algorithm 2 Hierarchical Global Consensus

```
1: init():
2:    r := node_id
3:    α := the enterprise that initiates the consensus
4:    P := the set of primary agents of all enterprises
5:    p := the primary of α

6: upon receiving transaction m and (r == p)
7:    if m is valid then
8:       multicast ⟨⟨PROPOSE, h_L, h_G, d⟩_σp, m⟩ to P

9: if (r == p) OR (upon receiving ⟨⟨PROPOSE, h_L, h_G, d⟩_σp, m⟩ from initiator primary p and r ∈ P)
10:   initiate local consensus
11:   if agreement is achieved then
12:      multicast ⟨ACCEPT, h_L, h_G, d, r⟩_σr to P {r or all agents}

13: upon receiving matching ⟨ACCEPT, h_L, h_G, d, q⟩_σq from two-thirds of the enterprises including α and r ∈ P
14:   initiate a local consensus
15:   if agreement is achieved then
16:      multicast ⟨COMMIT, h_L, h_G, d, r⟩_σr {either r or all agents}

17: upon receiving matching ⟨COMMIT, h_L, h_G, d, r⟩_σr from two-thirds of the enterprises including α
18:   execute and append the transaction to the ledger
```

its agents to internally decide on the enterprise vote in that phase.

In addition, since the agents of the initiator enterprise participate in the global consensus, in contrast to the first approach, the local and global orderings are merged together. However for each step of the global ordering the protocol ensures that the initiator enterprise agrees with the ordering. Hence the transaction is ordered correctly with respect to the transactions that are initiated by the initiator enterprise.

Furthermore, in each step of the global ordering and within each enterprise, once agreement is established, depending on the utilized local consensus protocol, either the primary, e.g., in Paxos, or every agent, e.g., in PBFT, sends the vote to other enterprises. For example, if the local consensus protocol is Paxos, only the primary sends the vote while in PBFT, every agent sends the vote. Indeed, if an enterprise uses a crash fault-tolerant protocol, the primary reply is counted as the enterprise vote by other enterprises whereas if an enterprise uses a Byzantine fault-tolerant protocol, other enterprises wait for $f + 1$ matching replies from the agents of the enterprise to count it as the enterprise vote. It is needed because in a crash fault-tolerant protocol, the primary is non-malicious,

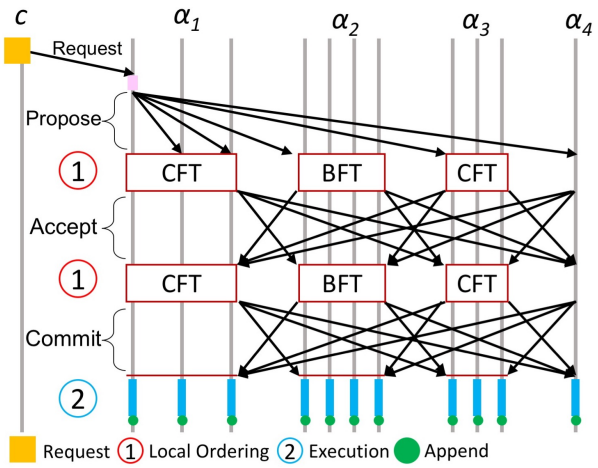


Figure 3.4: Hierarchical global consensus

whereas in a Byzantine fault-tolerant f nodes (including primary) might be malicious, thus other enterprises wait for $f + 1$ matching replies to ensure it is valid.

Algorithm 2 presents the *hierarchical global consensus*. Same as before, every sent and received message is logged by the agents. As presented in lines 1-5 of the algorithm, P is the set of primary agents of all enterprises and p is the primary agent of the initiator enterprise α.

Once the primary agent p of the initiator enterprise receives a valid cross-enterprise transaction, as indicated in lines 6-8, it multicasts a signed $\langle\langle \text{PROPOSE}, h_L, h_G, d\rangle_{\sigma_p}, m\rangle$ message to the primary agents of every enterprise where d is the digest of m, $h_L = H(t)$ such that t is the previous transaction that is initiated by the enterprise, and $h_G = H(t')$ such that t' is the previous cross-enterprise transaction. Note that hash h_L is only used by the agents of the initiator enterprise to ensure that the new transaction is ordered correctly with respect to the transactions that are initiated by the enterprise. The agents of the other enterprises ignore h_L and do not include it in their future messages.

As can be seen in lines 9-12, upon receiving a propose message from an enterprise, the primary agent of every enterprise checks the signature, hash h_G, and message di-

gest to ensure the message is valid. Then, every primary agent (including the primary of the initiator enterprise) *internally* initiates the local consensus protocol to establish agreement on the order of the requested transaction. If agreement is achieved, depending on the utilized local consensus protocol, either the primary or every agent multicasts an accept message to the primary agents of all other enterprises. Note that local consensus is needed to ensures that non-faulty agents agree with the received propose message. Hence, if agreement achieved, they just log the messages and do not append the transaction to their copies of the ledger.

As shown in lines 13-16, each enterprise waits for valid accept messages from *two-thirds* of the enterprises *including* the initiator enterprise which are matched with the accept message that is sent by the enterprise. Note that a valid accept message from initiator enterprise is needed to ensure that the transaction is consistent with the transactions that are initiated by that enterprise. Upon receiving a sufficient number of accept messages, the primary agent of each enterprise initiates the local consensus protocol to establish agreement on the received accept messages. Once agreement is achieved, the enterprise (either the primary or every agent) multicasts a commit message to every agent of all other enterprises.

The propose and accept phases of the global consensus, similar to `pre-prepare` and `prepare` phases of PBFT [21], guarantee that non-malicious enterprises agree on a total order for the transactions. Indeed, they ensure that no fork happens in the blockchain, i.e., it is not possible to have two different transactions with the same hash h_G. This is true because at least *two-thirds* ($\lfloor \frac{2|A|}{3} \rfloor + 1$) of the enterprises agreed with the order of each transactions, thus any two quorums of enterprises intersect in at least $\lfloor \frac{|A|-1}{3} \rfloor + 1$ enterprises. Since at most $\lfloor \frac{|A|-1}{3} \rfloor$ enterprises might be malicious, there is at least *one non-malicious* enterprise in the intersection of any two quorums.

Finally, in lines 17 and 18, similar to the previous phase, if an enterprise receives

Algorithm 3 One-Level Global Consensus

init():
  r := *node_id*
  α := the enterprise that initiates the consensus
  p := the primary agent of α

upon receiving transaction m and $(r == p)$
  if m is *valid* then
    broadcast $\langle\langle \mathsf{PROPOSE}, h_L, h_G, d\rangle_{\sigma_p}, m\rangle$ to every agents

upon receiving $\langle\langle \mathsf{PROPOSE}, h_L, h_G, d\rangle_{\sigma_p}, m\rangle$ from *primary p*
  if the message is *valid* then
    broadcast $\langle \mathsf{ACCEPT}, h_L, h_G, d, r\rangle_{\sigma_r}$

upon receiving matching $\langle \mathsf{ACCEPT}, h_L, h_G, d, r\rangle_{\sigma_r}$ from **local-majority** of **two-thirds** of the enterprises including α
  if the message is *valid* then
    broadcast $\langle \mathsf{COMMIT}, h_L, h_G, d, r\rangle_{\sigma_r}$

upon receiving matching $\langle \mathsf{COMMIT}, h_L, h_G, d, r\rangle_{\sigma_r}$ from **local-majority** of **two-thirds** of the enterprises including α
  execute and append the transaction to the ledger

matching commit messages from *two-thirds* of the enterprises including the initiator one that match the enterprise's commit message, its agents execute the transaction and append it to their ledgers.

Figure 3.4 shows the hierarchical consensus with four enterprises (similar to Figure 3.3) where enterprises α_1 and α_3 use a crash fault-tolerant (CFT) protocol and enterprise α_2 uses a Byzantine-fault-tolerant (BFT) protocol as their local consensus protocol. Here, the primary of enterprise α_1 initiates the consensus.

As an optimization, for a system with a high percentage of cross-enterprise transactions and to prevent the initiation of concurrent cross-enterprise transactions, the primary node of one of the enterprises can be designated as a *super primary* where every enterprise sends its cross-enterprise transaction to the super primary and the super primary initiates the protocol.

Safety and Liveness. In the hierarchical consensus, as discussed earlier, since at least $\lfloor \frac{2|A|}{3} \rfloor + 1$ of the enterprises must agree with the order of a transaction and at most $\lfloor \frac{|A|-1}{3} \rfloor$ enterprises might be malicious, safety is ensured. Indeed, if two or more concurrent transactions are initiated, at most one of them collects the required number of messages

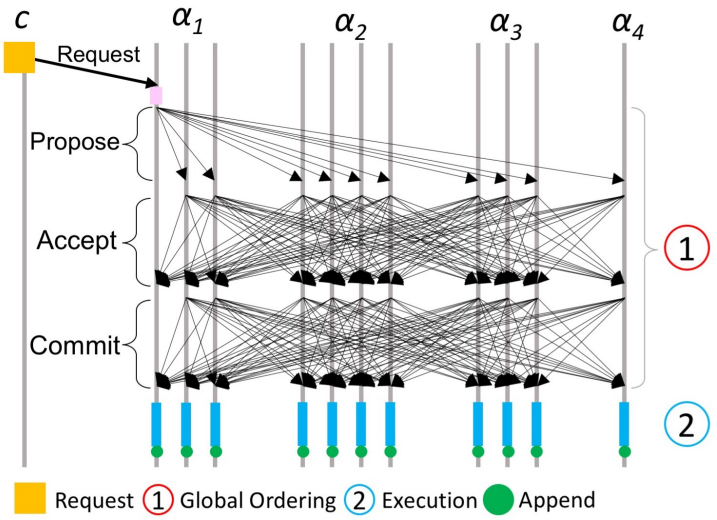


Figure 3.5: One-level global consensus

(two-thirds of the enterprises) , i.e., it is not possible for more than one of them to be ordered with the same hash h_G. If none of the concurrent transactions collects enough votes, all initiator enterprises try to send their transactions again. In such a situation and to ensure liveness, CAPER assigns a timer to each transaction and delays the transactions to prevent concurrent re-initiation of the transactions.

3.6.3 One-Level Global Consensus

While hierarchical consensus eliminates the need for having an extra set of orderer nodes and also distinguishes between trust at the node level and trust at the enterprise level, it requires an expensive two-level consensus protocol where each step of the global consensus needs the entire local consensus protocol to be run within each enterprise. In this section, we introduce a *one-level* global consensus protocol where for each cross-enterprise transaction, the agents of all enterprises participate to achieve consensus on the order of the transaction.

Since the number of agents of each enterprise depends on the utilized consensus

protocol within the enterprise, the required number of matching replies to ensure that the majority of agents of an enterprise agree on the order of the transaction is different from enterprise to enterprise. Therefore, we define *local-majority* as the required number of matching messages from the agents of an enterprise. If the agents of an enterprise are crash-only nodes, local-majority for the enterprise is equal to $f+1$ (from the total $2f+1$ agents), and if the agents of an enterprise might behave maliciously, local-majority for the enterprise is equal to $2f+1$ (from the total $3f+1$ agents). For an enterprise that has only a single reliable agent, local-majority is one.

Algorithm 3 presents the *one-level global consensus*. Variable p indicates the primary agent of the initiator enterprise α. As shown in lines 5-7, the primary of the initiator enterprise broadcasts a signed propose message including the transaction, the hash of the previous transaction that is initiated by the enterprise (h_L), and the hash of the previous cross-enterprise transaction (h_G) to the agents of every enterprise. Same as before, h_L is only used by the agents of the initiator enterprise.

Once an agent receives a propose message, it checks the signature, message digest, and hash h_G to ensure the message is valid. If the agent belongs to the initiator enterprise, it also checks hash h_L. Once the message is validated, the agent broadcasts an accept message to every agent of every enterprise, as indicated in lines 8-10.

As presented in lines 11-13, upon receiving valid accept messages from the local-majority of two-thirds of the enterprises *including* the initiator enterprise that match the accept message which is sent by the agent, each agent broadcasts a commit message to every agent of every enterprise. The propose and accept phases of the algorithm, similar to `pre-prepare` and `prepare` phases of PBFT [21], guarantee that non-faulty agents agree on a order for the transactions.

Finally, as shown in lines 14 and 15, once an agent receives valid commit messages from local-majority of two-thirds of the enterprises including the initiator enterprise that

matches its commit message, the agent considers the transaction as committed, thus, executes the transaction and appends the transaction to the ledger.

Figure 3.5 shows the one-level consensus in CAPER for four enterprises with different failure modes.

Safety and Liveness. To ensure safety in one-level consensus, in each of the accept and commit phases, matching messages from the local majority of two-thirds of the enterprise is required. Local majority of each enterprise is needed to ensure that any two quorums intersect in at least one non-faulty node within the enterprise and two-thirds of the enterprises is needed to ensure that any two quorums intersect in at least one non-malicious enterprise (since at most $\lfloor \frac{|A|-1}{3} \rfloor$ enterprises might be malicious). Thus, similar to the hierarchical ordering, even if two or more concurrent transactions are initiated, at most one of them collects required number of matching accept and commit messages. To ensure liveness, similar to the hierarchical consensus, CAPER assigns timers to delay concurrent transactions and also as an optimization uses a super primary for systems with a high percentage of cross-enterprise transactions.

3.7 Experimental Evaluations

In this section, we conduct several experiments to evaluate CAPER. As explained earlier, CAPER supports distributed applications consisting of a set of collaborating enterprises where each enterprise maintains its data in a datastore consisting of private and public records. The private records of the datastore, which are replicated across the agents of enterprise, include the data of internal transactions. The public records, on the other hand, are replicated across every agent of every enterprise and include the data of cross-enterprise transactions. For the purpose of this evaluation, enterprises are implemented as simple accounting applications where clients can initiate transactions to

transfer assets from one or more of their accounts to other accounts.

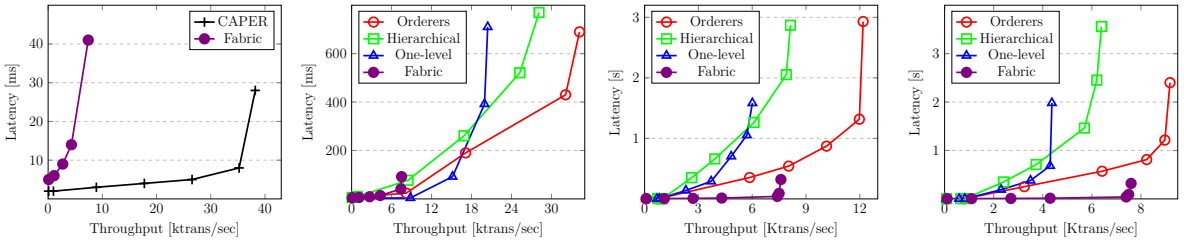
In addition to CAPER, we also implemented a permissioned blockchain system specifically designed in the execute-order-validate architecture introduced by Fabric [27] where the transactions (internal as well as cross-enterprise) of different enterprises are executed by the agents (endorsers) of their enterprises in parallel, ordered by a separate set of orderers, and then validated by every agent of every enterprise. Each block also consists of a single transaction (as in [59]). Note that in the case of Fabric, all internal as well as cross-enterprise transactions are ordered by orderers and all the enterprises maintain the same blockchain ledger (i.e., the confidentiality of data is not preserved).

The experiments are conducted on the Amazon EC2 platform. Each VM is Compute Optimized c4.2xlarge instance with 8 vCPUs and 15GB RAM, Intel Xeon E5-2666 v3 processor clocked at 3.50 GHz. In each experiment, we increase the total number of transactions per second from 100 to 100000 (by increasing the number of clients running on a single VM) and measure the end-to-end throughput (x-axis) and latency (y-axis) of the system. The load is equally distributed among the enterprises.

When reporting throughput measurements, we use an increasing number of clients running on a single VM, until the end-to-end throughput is saturated, and state the throughput just below saturation. Throughput numbers are reported as the average measured during the steady state of an experiment.

3.7.1 Workloads with Cross-Enterprise transactions

In the first set of experiments, we measure the performance of CAPER for workloads with different percentage of cross-enterprise transactions, i.e., 0%, 20%, 80%, and 100%. We consider four enterprises where each enterprise has three agents and uses a Paxos protocol with $f = 1$ to establish consensus on its internal transactions. To process



(a) 0% Cross-app. (b) 20% Cross-app. (c) 80% Cross-app. (d) 100% cross-app.

Figure 3.6: Performance with different percentage of cross-application transactions

cross-enterprise transactions we implement all three approaches, using a set of orderers (we refer to this approach as *orderers*), hierarchical, and one-level, which are explained in Section 3.6. Orderers are implemented using a typical Kafka orderer setup with 3 ZooKeeper nodes, 4 Kafka brokers and 3 orderers (similar to the ordering service of Fabric [27]). The results are shown in Figure 3.6(a)-(d).

When all transactions are internal (Figure 3.6(a)), each enterprise processes its transactions independent of other enterprises. In such a situation, CAPER is able to process upto 36000 transactions (9000 transactions per enterprise) with very low latency (∼10 ms). Here, since there is no dependency between the transactions of different enterprises and the blockchain views of enterprises are constructed in parallel, the throughput of the entire system increases linearly by increasing the number of enterprises. With the same latency as CAPER (∼10 ms), Fabric processes 3000 transactions. Fabric is able to process upto 7000 transactions in total with 40 ms latency, however, the end-to-end throughput is saturated beyond 7000 transactions. In Fabric, adding more enterprises only increases the number of parallel execution threads and since every transaction of all enterprises is ordered by the same set of orderers, the performance of Fabric is not significantly improved. Note that since all transactions are internal, if Fabric uses different channels for different enterprises, it can linearly scale as the number of enterprises increases. However, even with that improvement, CAPER still provides higher throughput

(∼29% higher in its peak throughput).

In the second set of experiments, the workload is changed to include 20% cross-enterprise transactions which are equally initiated by different enterprises. As can be seen in Figure 3.6(b), when CAPER is not heavily loaded, the one-level consensus approach has better performance since it involves less number of communication phases. As can be seen, the one-level approach processes ∼16000 transactions with 90 ms latency whereas the hierarchical and orderers approaches have latencies of 220 and 150 ms latency respectively in order to process the same number of transactions.

Once CAPER becomes heavily loaded, cross-enterprise transactions might be initiated in parallel. As a result, the latency of the hierarchical and one-level approaches are dramatically increased (as discussed in Section 3.6). With 400 ms latency, using orderers, CAPER is able to process 30000 transactions whereas the one-level and hierarchical approaches process 18000 and 19000 transactions (resp.).

The performance of Fabric, however, is not affected by increasing the percentage of cross-enterprise transactions (since all transactions are ordered by the same set of orderers) and Fabric still processes upto 7000 transactions in total with 40 ms latency.

Increasing the percentage of cross-enterprise transactions to 80% (Figure 3.6(c)) decreases the performance of all three approaches. In this case, using orderers, CAPER is able to process ∼10000 transaction with sub-second latency whereas the hierarchical and one-level approaches process upto 6000 transactions with the same latency. This is expected because in the hierarchical and one-level approaches when the system is heavily loaded, different nodes receive concurrent transactions in different orders.

When all transactions are cross-enterprise, the one-level consensus can only process ∼4000 transactions per second with 800 ms latency whereas using the orderers, CAPER is able to process ∼9000 transactions with the same latency. Fabric, same as before, is able to process upto 7000 transactions in total with 40 ms latency which is better than

both hierarchical and one-level approaches.

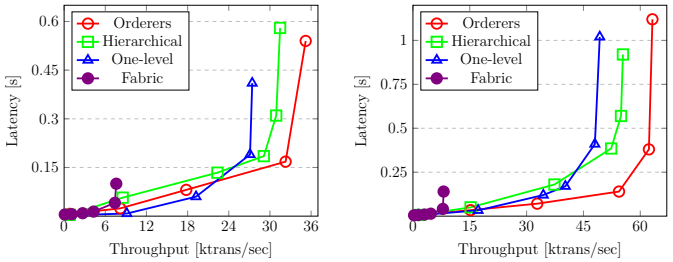
As mentioned before, since Fabric orders all internal as well as cross-enterprise transactions by the same set of orderers, the performance of Fabric is not affected by increasing the percentage of cross-enterprise transactions. As a result, in workloads with 80% and 100% cross-enterprise transactions, Fabric performs better than all other approaches in terms of latency. In fact, for cross-enterprise transactions and to achieve consensus, CAPER uses either multiple rounds of consensus (in the orderers and hierarchical approaches) or a Byzantine fault-tolerant protocol with a large number of participants (in the one-level approach). This is in contrast to Fabric that relies on a single crash fault-tolerant protocol (with only three nodes) to order the transactions which results in a lower latency. However, Fabric does not ensure confidentiality of data in this settings.

Note that even with 100% cross-enterprise transactions, the throughput of CAPER using the orderers approach is slightly higher than Fabric (9% higher in its peak throughput) because the throughput of Fabric is affected by conflicting transactions, i.e., transactions that access the same records, due to its execute-order-validate architecture [8] (in the experiments, ∼10% of the transactions are conflicting).

3.7.2 Performance with Multiple Enterprises

In the next set of experiments, we measure the performance of CAPER in two deployments with 4 and 8 enterprises where each enterprise has three agents and uses a Paxos protocol with $f = 1$ to establish consensus on its internal transactions. For each deployment, we consider workloads with 90% internal and 10% cross-enterprise transactions (the typical settings in partitioned databases [60] [61]).

For the deployment with four enterprises (Figure 3.7(a)), the performance of CAPER is close to the scenario in Figure 3.6(b) where in the workloads with less than ∼20000



(a) 4 Enterprises (b) 8 Enterprises

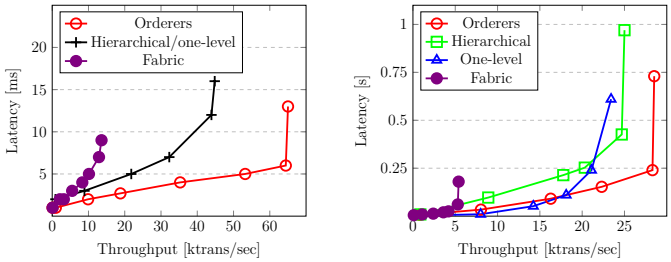
Figure 3.7: Performance with 4 and 8 enterprises

transactions per second, the one-level consensus has better performance and beyond that, using orderers is more beneficial. Increasing the number of enterprises (Figure 3.7(b)), however, results in higher latency for the same throughput in both one-level and hierarchical consensus due to the increasing number of nodes which increases the chance of conflict between concurrent transactions. Note that since 90% of the transactions are internal, increasing the number of enterprises improves the overall throughput of CAPER near-linearly (using orderers, CAPER processes upto ∼32000 and ∼59000 transactions per second with four and eight enterprises (respectively) with 30 ms latency). As mentioned earlier, in Fabric, increasing the number of enterprises does not significantly improve the performance.

3.7.3 Different Failure Models Performances

Finally, in the last set of experiments, we change the failure model of nodes (agents as well as orderers in the first approach of global consensus). We assume a deployment with four enterprises and a workload with 90% internal and 10% cross-enterprise transactions. Since the crash-only nodes are considered in the previous experiments, in this set of experiments, we only consider reliable and Byzantine nodes.

In the first settings, all nodes are reliable, thus each enterprise has a *single* agent, i.e., the internal transactions need no consensus and get committed as soon as received by the



(a) Reliable nodes (b) Byzantine nodes

Figure 3.8: Performance with different failure models

agents. In addition, in the first approach of global consensus, cross-enterprise transactions are ordered using a single reliable orderer. Note that since each enterprise has only a single agent, the hierarchical and one-level consensus protocols become identical and they both become similar to PBFT. The only difference is that here every agent can initiate a cross-enterprise transaction whereas in PBFT only the primary can.

The results are shown in Figure 3.8(a). By relying on an orderer node, CAPER orders cross-enterprise transactions in only two phases: order and sync, thus the system is able to process more than 60000 transactions per second with very low latency ($\sim$ 5 ms). It should be mention that although reliable nodes show a significant performance, the assumption is that they never fail which seems to be unrealistic.

In the second settings, nodes are Byzantine. As a result, each enterprise relies on four agents ($f = 1$) to order internal transactions. Similarly, there are four orderers to order cross-enterprise transactions in the first approach of global consensus and in Fabric. The results can be seen in Figure 3.8(b) where, as expected, the performance of the system reduces in comparison to crash-only and reliable nodes due to the increasing number of nodes, messages, and communications phases (recall than Byzantine fault-tolerant protocols establish consensus in three phases with a quadratic number of message exchanges). It is noteworthy to mention that even when all agents and orderers follow the Byzantine failure model, CAPER is still able to process more than 20000 (using orderers,

$\sim$ 28000) transactions per second with 250 ms latency.

Note that different enterprises and also the set of orderers might have different failure models, thus, many possible combinations can be considered (5^3 combinations when the system has only four enterprises). Here, we only measured the settings where every node (agent as well as orderer) follows the same failure model.

3.8 Summary

In this chapter, we proposed CAPER, a permissioned blockchain system that supports both internal and cross-enterprise transactions of collaborating enterprises. CAPER targets both performance and confidentiality aspects of blockchain systems. To achieve better performance, CAPER orders and executes internal transactions of different enterprises simultaneously. In addition, to achieve confidentiality, the blockchain ledger is *not maintained* by any node and each enterprise maintains its own local view of the ledger including its internal and all cross-enterprise transactions. CAPER also distinguishes between trust at the node level and trust at the enterprise level and allows an enterprise to behave maliciously for its benefit while its nodes are non-malicious. Furthermore, CAPER introduces three consensus protocols to globally order cross-enterprise transactions: using a separate set of orderers, hierarchical consensus, and one-level consensus. Our experiments show that for lightly loaded enterprises one-level consensus shows better performance whereas using a set of orderers is more beneficial for heavily loaded enterprises. In the absence of extra resources for orderers, the hierarchical approach can provide better performance in heavily loaded enterprises. CAPER is able to process around 10000 transactions per second per enterprise in workloads with a reasonable percentage (10%) of cross-enterprise transactions.

Chapter 4

SEPAR: On Verifiability of Permissioned Blockchains

4.1 Introduction

The rise of the *platform economy* [62, 63] is reshaping work all around the world. Crowdsourcing platforms dedicated to work (also called *crowdworking platforms* [64]) are online intermediaries between *requesters* and *workers*, where requesters propose *tasks* while *workers* propose skills and time. By providing requesters (resp. workers) 24/7 access to a worldwide workforce (resp. worlwide task market), crowdworking platforms have grown in numbers, diversity, and adoption[1]. Today, crowdworkers come from countries spread all over the world, and work on several, possibly competing, platforms [64]. The use of crowdworking platforms is expected to continue growing [65], and in fact they are envisioned as key technological components of the future of work [66].

[1]See for example : Amazon Mechanical Turk (`https://www.mturk.com/`), Wirk (`https://www.wirk.io/`), or Appen (`https://appen.com/`) for micro-tasks, Uber (`https://www.uber.com/`) or Lyft (`https://www.lyft.com/`) for rides, TaskRabbit (`https://www.taskrabbit.com/`) for home maintainance, Kicklox (`https://www.kicklox.com/`) for collaborative engineering.

Crowdworking platforms, however, challenge national boundaries, weaken the formal relationships between workers and requesters, and are often not considered legal as employers. Guaranteeing the compliance of crowdworking platforms with national or regional labour laws is hard[2] [65] despite the stringent need for regulating work. For example, the preamble of the 1919 constitution of the International Labour Organization [67], written in the ruins of World War I, states that: *"Whereas universal and lasting peace can be established only if it is based upon social justice; (...) an improvement of those conditions is urgently required; as, for example, by the regulation of the hours of work, including the establishment of a maximum working day and week, the regulation of the labour supply (...)".* The global regulation of the work hours represents the *minimal* and *maximal* number of hours that participants, i.e., worker, requester, and platform, can spend on crowdworking platforms. While legal tools are currently being investigated, e.g., a *Universal Labour Guarantee* [65], there is a stringent need for technical tools allowing official institutions to enforce regulations.

Most current crowdworking platforms are independent of each other. However, the emergence of more complex tasks and novel requirements for both workers and requesters, on one hand, and the enforcement of legal regulations, on the other hand, highlights the need for collaboration between crowdworking platforms, thus resulting in *multi-platform crowdworking systems*. For example, many drivers work for both Uber and Lyft concurrently[3], while requesters may also request multiple drivers from both Uber and Lyft concurrently. The observation holds also for microtask platforms [64], where a common combination among workers is *Amazon Mechanical Turk* and *Prolific*, or for on-demand services[4]. Participants in a crowdworking task may also behave maliciously or act as

[2]See, e.g., the `Otey V Crowdflower` class action against a famous microtask platform for *"substandard wages and oppressive working hours"* (`https://casetext.com/case/otey-v-crowdflower-1`).

[3]For example, `ridesharapps.com` provides tutorials to help drivers manage apps to optimize their earnings `https://rideshareapps.com/drive-for-uber-and-lyft-at-the-same-time/`.

[4]See, e.g., `https://tinyurl.com/nytgigmult`.

adversaries for their benefits, e.g., violate the privacy of participants or the regulations. Therefore, to check the enforcement of legal regulations in a multi-platform crowdworking environment, we need to reconcile *transparency* with *privacy*. Indeed, while enforcing limits on the hours of work over several crowdworking platforms requires the *transparent* sharing of information about the crowdworking tasks performed by each platform, without any *privacy protection* measures, this may lead to out-of-control disclosures about the participants. Transparent and privacy-preserving collaboration between multiple platforms might also be needed to address *complex cross-platform* tasks. If a requester submits a task with a specified number of requested solutions to multiple platforms, the involved platforms need to collaborate with each other in order to assign workers and provide the specified number of solutions. As a result, a multi-platform system needs to establish consensus between platforms to enable them either to enforce legal regulations or to process cross-platform tasks.

In this chapter, we present *SEPAR*, a technical solution to the problem of imposing global *constraints* on distributed independent entities in the context of multi-platform crowdworking systems. The problem is non-trivial because of the complexity of the conjunction of the required properties:

1. **Expressibility**: The constraints need to be expressed in a *simple and non-ambiguous* manner.
2. **Transparent and Privacy-preserving Constraint Enforcement**: Crowdworking platforms need to *share information* about the tasks performed without jeopardizing the privacy of participants in order to allow both the enforcement of the global constraints and the collaborative processing of cross-platform tasks.
3. **Distributed Collaboration**: Crowdworking platforms are naturally distributed and need to collaborate through distributed consensus algorithms.

SEPAR proposes a privacy-preserving token-based system where global constraints

are modeled using *lightweight and anonymous tokens* distributed to workers, platforms, and requesters. Our system formally guarantees that global constraints are satisfied *by construction* and limits the information shared among platforms and participants to the minimum necessary for performing the tasks against *adversarial participants acting as covert adversaries*. We extend our token-based system to allow participants to prove to external entities (e.g., social security agencies) their involvement in crowdworking tasks. The resulting proofs are called *certificates*. To provide transparency across multiple platforms, SEPAR proposes a *blockchain-based distributed ledger* shared across platforms. Nonetheless, for the sake of privacy and to improve performance, the blockchain ledger is *not maintained* by any single platform and each platform maintains only a view of the ledger. We then design a suite of distributed consensus algorithms across platforms for coping with the concurrency issues inherent to a multi-platform context and formally prove their correctness. Salient features of SEPAR include the simplicity of its building blocks (e.g., usual asymmetric encryption scheme) and its compatibility with today's platforms (e.g., it does not jeopardize their privacy requirements about requesters and workers for enforcing the regulation).

In a nutshell, the contributions of this chapter are as follows:

1. A privacy model stating formally the privacy requirements of a multi-platform regulated crowdworking system based on the well-known simulatability paradigm,
2. A simple language for expressing *global constraints*, e.g., limits on the number of work hours, and mapping them to `SQL` constraints to ensure semantic clarity,
3. SEPAR, a *privacy-preserving transparent* multi-platform crowdworking system that enforces a given set of constraints. (1) **Privacy** is ensured using *lightweight and anonymous tokens*, while (2) **transparency** is achieved using a *blockchain* shared across platforms. The token-based system is extended to allow participants to prove to external entities their involvement in crowdworking tasks.

4. A suite of distributed consensus protocols for coping with the concurrency issues inherent to a multi-platform context, and
5. A formal security analysis and thorough experimental evaluation.

The chapter is organized as follows. Section 4.2 defines the problem that SEPAR addresses. The language for expressing constraints is expressed in Section 4.3. The token-based system for enforcing constraints and the extended system for certificates are designed in Section 4.4. The blockchain ledger and consensus protocols are presented in Section 4.5. Section 4.6 details our thorough experimental evaluation, and finally, Section 4.7 concludes the chapter.

4.2 Problem Formulation

In this section, we provide a motivating example to illustrates the challenges of crowdworking systems and then formulate the problem. Next, We explain the security model.

4.2.1 Motivating Example

Multi-platform crowdworking systems face two main privacy preserving challenges: enforcing multi-platform regulations and supporting cross-platform tasks. We consider constraining the number of work hours in a ridesharing use-case to illustrate the challenge of enforcing privacy preserving multi-platform regulations. In ridesharing scenarios, a set of workers (i.e., drivers) gives rides to a set of requesters (i.e., travelers) through a set of platforms, e.g., Uber, Lyft, Curb, and Juno, where each driver (resp. traveler) registers to one or more platforms. Regulations on the hours of work often specify minimal and maximal number of work hours that can be performed by the participants. For instance, (1) the total work hours of a driver per week may not exceed 40 hours to follow the

Fair Labor Standards Act[5] (FLSA), (2) a driver has to work at least 5 hours per week to be eligible for insurance coverage, and (3) the total work hours of all drivers on a platform should be at least 1000 hours per week to enable the platform to fill for a tax refund. A multi-platform crowdworking system needs to express and enforce such regulations while preserving the privacy of participants. Indeed, the system needs to (1) provide a technical tool to enable official institutions expressing the regulations, (2) support transparent sharing of information about the crowdworking tasks performed by each platform to enable them checking the enforcement of regulations, and (3) preserve the privacy of participants.

Supporting complex cross-platform tasks that may need multiple contributions from possibly different platforms raises the second set of challenges. For instance, a requester who has registered with *Amazon Mechanical Turk*, *Appen* and other microtask platforms might need hundreds or thousands of contributions at the same time. The requester would like to accept these contributions from workers regardless of the platforms the microtasks are performed on. Since workers from different platforms might want to perform these contributions, the system needs to establish consensus among the various microtask platforms to assign workers and provide the specified number of solutions without revealing any private information about the workers to competing platforms.

4.2.2 Crowdworking Environment

Participants

Today's realistic *crowdworking environments* consist of a set of workers $\mathcal{W}$ interacting with a set of requesters $\mathcal{R}$ through a set of competing platforms $\mathcal{P}$. We call *participants* the workers, platforms, and requesters of a crowdworking environment. Each worker

[5] `https://www.dol.gov/agencies/whd/flsa`

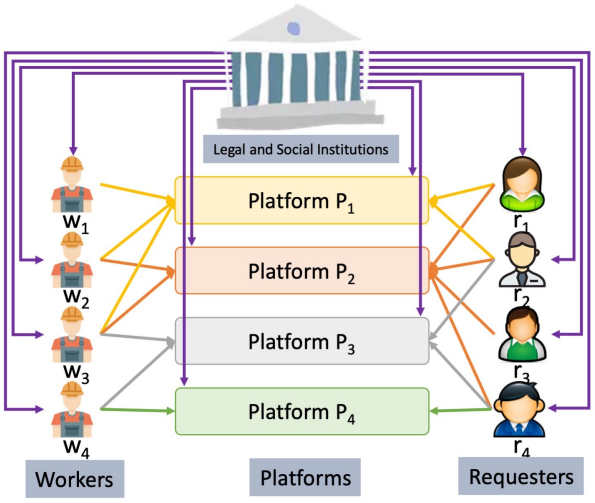


Figure 4.1: A crowdworking infrastructure

$w \in \mathcal{W}$ (1) registers to one or more platforms $\mathcal{P}_w \subset \mathcal{P}$ according to her preferences and, through the latter, (2) accesses the set of tasks available on $\mathcal{P}_w$, (3) submits each *contribution* to the platform $p \in \mathcal{P}_w$ she elects, and (4) obtains a *reward* for her work. On the other side, each requester $r \in \mathcal{R}$ similarly (1) registers to one or more platforms $\mathcal{P}_r \subset \mathcal{P}$, (2) issues a *submission* which contains her tasks $\mathcal{T}_r$ to one or more platforms $p \in \mathcal{P}_r$, (3) receives the contributions of each worker w registered to $\mathcal{P}_r \cap \mathcal{P}_w$ having elected a task $t \in \mathcal{T}_r$, and (4) launches the distribution of rewards. Platforms are thus in charge of facilitating the intermediation between workers and requesters. A *crowdworking process* π connects three parties – a worker w, a platform p, and a requester r – with each other and aims to solve a task $t \in \mathcal{T}_r$ through p, and consists in the steps (2) to (4) above. Figure 4.1 shows a crowdworking infrastructure with four platforms, four workers and four requesters.

In this work, we do not focus on the description of tasks and contributions and consequently model both as arbitrary bitstrings $\{0,1\}^*$ and make no assumption on the

distribution of rewards to workers[6].

Finally, workers, requesters, and platforms are all equipped with the cryptographic material required by SEPAR: a pair of usual public/private asymetric keys (e.g., RSA) and a pair of public/private asymetric *group* keys (e.g., [68]) where the union of all workers forms a group (in the sense of group signatures), similar to the union of all requesters, and to the union of all platforms. Participants acquire them when joining SEPAR (see Section 4.4 for more information).

Interactions with Institutions

Crowdworking environments do not exist in a vacuum but rather are integrated within society as a whole. The participants need in particular to interact with legal institutions (in order to enforce the local labor laws) and with social institutions (in order to enable local social rights). We capture these interactions through less-than *constraints* and greater-than *certificates*.

Constraints. A set of constraints embodies the labor policy that applies to a given crowdworking environment. Essentially, a constraint expresses a limit on the *actions* that can be performed by the participants of the crowdworking environment, e.g., the total working hours of a worker per week must not exceed 40 hours across all platforms. Constraints must be expressed in an intuitive language that is both expressive enough to adapt to a variety of real-world policies and at the same time restrictive enough to guarantee the efficiency of their enforcement.

Certificates. A certificate is a piece of information that participants can provide to third parties to prove that they took part in a given crowdworking process. Contrary to constraints, they are not *a priori* specification: they are made available during the

[6]A task embeds all the information necessary to be performed by a worker (*e.g.,* the precise description of the work that must be performed, a *reward policy* for distributing the reward among contributors, the expected number of contributions).

process to the participants involved and can be provided by participants to other parties on demand after the process. Certificates are well suited to real-world situations such as conforming to legal obligations, suing other parties in court in case of abuse, or legitimizing applications to grants or tax refund depending on local legislation, e.g., a driver has to work at least 5 hours per week to be eligible for insurance coverage or the total work hours of all drivers on a platform should be at least 1000 hours per week to enable the platform to fill for a tax refund.

Distribution of Platforms

We do not make any assumptions on the inner working of platforms, especially on their inner implementation of crowdworking processes (e.g., task assignment algorithm, workers contributions delivery). However, we stress that our approach is compatible with distributed infrastructures, supported by one or more data centers, following today's fault tolerance and performance standards. In particular, we assume each platform consists of a set of nodes in an asynchronous distributed network. Nodes are connected by point-to-point bi-directional communication channels. To guarantee data consistency, a total order among the transactions of each platform is needed. To establish a total order, asynchronous consensus protocols can be used where nodes agree on an ordering of incoming requests using the state machine replication algorithm [50]. Each node has a distributed persistent transparent datastore where transactions are committed to the datastore. In this chapter and due to the unique features of blockchains such as transparency, provenance, and fault tolerance, the datastore is implemented using a *blockchain*. In particular and for a crowdworking system, the *transparency* of blockchains can be used to check integrity constraints, *provenance* enables the system to trace how data is transformed, *fault tolerance* helps to enhance reliability and availability, and finally, *authenticity* guarantees that signatures and transactions are valid.

A crowdworking environment processes *internal*, i.e., submitted to a single platform, and *cross-platform*, i.e., submitted to more than one platform, tasks. Processing a task (either internal or cross-platform) requires agreement from the nodes of the involved platforms. To establish agreement among the nodes, we introduce *local* and *cross-platform* consensus protocols. In addition, we enable all platforms checking the satisfaction of constraints by establishing consensus among every node of all platforms. To do so, a *global* consensus protocol is introduced.

4.2.3 Security Model

We consider that any participant in a crowdworking environment may act as a *covert adversary* [69] that aims at inferring anything that can be inferred from the execution sequence and that is able to deviate from the protocol if no other participant detects it. Adversarial participants may additionally collude.

The privacy definition that we adopt requires that no participant obtains or infers any information about a crowdworking process beyond what is strictly needed for accomplishing its local crowdworking processes and for the distributed enforcement of constraints. This requirement is formalized in [1] by defining the set of *secrets* and by using the well-known simulatability model often used by secure multi-party computation algorithms.

4.3 Expressing Global Regulations

We express global regulations using constraints and certificates. A constraint demonstrates a limit on the actions that can be performed by the participants of the crowdworking environment and a certificate is a piece of information that participants can provide to third parties to prove that they took part in a given crowdworking process. In this Section, we express both constraints and certificates.

4.3.1 Expressing Constraints

Syntax. We define a constraint c as being essentially (1) a triple (w, p, r) that associates a worker w, a platform p, and a requester r, and (2) a threshold θ (an integer) that defines the upper bound of c[7]. Intuitively, a constraint $((w, p, r), \theta)$ states that there must not be more than θ actions between the worker w, the platform p, and the requester r (see below for the detailed semantics). We also allow two wildcards to be written in any position of a triple: $*$ and $\forall$. First, the $*$ wildcard allows to ignore one or more elements of a triple[8]. For example $(*, p, r)$ means that the constraint applies to the couple (p, r). A triple may contain up to three $*$ wildcards. An element of a triple that is not a $*$ wildcard is called a *specified participant* of the constraint. Second, the $\forall$ wildcard factorizes the writing of triples because it allows to express a constraint that must hold for all participants in the same group of participants[9]. For example, $(\forall, p, r)$ represents the following set of triples: $\{(w, p, r)\}$, $\forall w \in \mathcal{W}$. We denote $\mathcal{C}$ the complete set of constraints.

Semantics. We give now a precise definition of the semantics of our constraints by illustrating how they translate to `SQL` constraints. Let assume that there exists a table of actions `A` that records all actions performed between any triple of worker, platform, requester. The attributes of `A` are `WORKER`, `PLATFORM`, `REQUESTER`. For simplicity, we consider a constraint c without any wildcard, i.e., $c \leftarrow ((w, p, r), \theta)$. The semantics of c is the same as the following `SQL` query :

```
ALTER TABLE A ADD CONSTRAINT c CHECK (
  NOT EXISTS (
    SELECT * FROM A
    WHERE WORKER=w AND PLATFORM=p AND REQUESTER=r
```

[7]Extending constraints with, e.g., labels for defining categories of actions (e.g., working time) or validity periods (e.g., "one week", "one month"), is straightforward.

[8]Intuitively, the $*$ wildcard means "whatever".

[9]Intuitively, the $\forall$ wildcard means "for each".

```
    GROUP BY WORKER, PLATFORM, REQUESTER
    HAVING COUNT(*) ≥ θ
  ) );
```

The presence of a $*$ wildcard in the triple simply leads to removing the corresponding attributes in the `WHERE` and `GROUP BY` clauses. The presence of a $\forall$ wildcard leads to expanding it to the set containing all the elements that it represents (e.g., all workers if the $\forall$ wildcard is at the first position in the triple) and to generate the cartesian product between the resulting set and the elements at the two other positions of the triple (that may be $\forall$ wildcards as well). Finally, the semantics of a set of constraints is the conjunction of the constraints contained in the set.

Example. the weekly FLSA limit on the total work hours per worker can easily be expressed as $c_{FLSA} \leftarrow ((\forall, *, *), 40)$.

4.3.2 Expressing Requests for Certificates

Syntax and Semantics. Certificates allow a participant called *prover* (e.g., worker) to prove to an external entity called *verifier* (e.g., social security agency) that a minimal number of hours have been spent on crowdworking platforms (e.g., for applying to insurance coverage). *Requests for certificates* (e.g., from social security agencies) are expressed using the same syntax as the constraints with the following two differences. First, the θ threshold does not represent an upper bound on actions that cannot be exceeded, but a lower bound on actions that have to be proved.

And second, there must always be at least one specified participant in a request for certificates, i.e., typically the prover. This syntax allows verifiers to follow *minimal disclosure principles* by requesting from the prover exactly the information needed about the crowdworking processes performed. There is no need to request the identities of the

participants with whom the prover collaborated. Additionally, it is trivial to connect multiple requests for certificates through conjunctions and disjunctions if needed.

Examples. A social security institution can request each worker w applying for insurance coverage to prove that she worked in total more than 5 hours: $r_1 \leftarrow ((w, *, *), 5)$ is both necessary and sufficient. Similarly, the request $r_2 \leftarrow ((*, p, *), 1000)$ allows a tax institution to ask for each platform p applying for a tax refund to prove that the total work hours of all its workers is at least 1000 hours.

4.4 Enforcing Global Regulations

In this section, we develop our conception of constraints and certificates, how they are built, and how to use them. The correctness and the privacy guarantees of our construction is proved in [1].

4.4.1 Implementing a Token-Based System

In this section, we show how constraints and certificates, which are expressed in Section 4.3, can be enforced and produced respectively. Inspired by e-cash systems, we enforce constraints and produce certificates by managing two *budgets* per participant while preserving both the privacy of participants and the correctness of budgets. Our proposal makes use of a centralized authority, called the *registration authority (RA for short)*. RA registers the participants to the crowdworking environment, sends them the required cryptographic material, receives the set of constraints, and manages the budgets. The required cryptographic material includes a pair of public/private asymetric keys (e.g., RSA) and a pair of public/private asymmetric *group* keys (e.g., [68]) for which the registration authority is the group manager, while the set of constraints may be expressed by the regulators through a dedicated interface. A group signature scheme is a signature

scheme for groups that respects three main properties, as defined first in [70]: (1) only members of the group can sign messages, (2) the receiver of the signature can verify that it is a valid signature of that group, but cannot discover which member of the group made it, and (3) in case of dispute later on, the signature can be "opened" (with or without the help of the group members) to reveal the identity of the signer. A common way to enforce the third property is to rely on a *group manager*, that can add new members to the group, or revoke the anonymity of a signature. Instances of such schemes are proposed in [70], but also in [68, 71]. In this chapter, we use the protocol proposed in [68], and denote $GroupSign(key_{priv,p}, g, m)$ the group signature of participant p (with her private key) of group g, for the message m. The notation $Sign(key_{priv,p}, m)$ may also be used to refer to a simple asymmetric signature of the message m by user p (e.g., RSA). We instantiate the budgets based on labeled, single-use, anonymous *tokens* and use a persistent transparent datastore to guarantee their correct and validated spending by participants. The persistent datastore is implemented using a *blockchains*. To process crowdworking tasks, our token-based system is defined by five functions: `GENERATE` for initializing the budgets and refilling them, `SPEND` for spending portions of the budgets, `PROVE` for providing certificates to a third party, `CHECK` for checking whether a given spending is allowed or not, and `ALERT` for reporting dubious spending.

The GENERATE Function

The registration authority uses the `GENERATE` function to initialize the budgets, i.e., constraint and certificate tokens of all participants (i.e., workers, platforms, requesters) and refill them periodically[10] according to the set of constraints $\mathcal{C}$ to enforce.

Constraint tokens. For each constraint $c \leftarrow ((w, p, r), \theta)$, the registration authority

[10]The refreshment rates of budgets is easily computed from the validity periods of constraints (see Section 4.3).

generates θ tokens and sends a copy of each token to each specified participant of c. A token consists of a public and a private component. The *public component* is a pair made of a *number used only once* (referred to as a *nonce* below) generated by the registration authority and a signature of the nonce produced by the registration authority[11]. The public component will be used later (upon completion of the corresponding task) by other platforms to check the validity of tokens. The *private component* is an index allowing the participants involved in a crowdworking process to select the correct set of tokens given the other specified participants involved in the process. We implement this private component as a list containing the public keys of the specified participants (in the corresponding constraint)[12]. Let $tk^{\dagger}$ be a constraint token, $tk^{\dagger}_{pub}$ be its public component, $tk^{\dagger}_{priv}$ be its private component, N be a nonce, and *pubs* the list of public keys. The constraint token is thus the couple $(tk^{\dagger}_{pub}, tk^{\dagger}_{priv})$ where $tk^{\dagger}_{pub} = (N, Sign(key_{priv,RA}, (N)))$ and $tk^{\dagger}_{priv} = pubs$.

Certificate Tokens. Certificate tokens are generated initially by the registration authority for all participants. For each crowdworking process, a single certificate token is linked to a fully specified triplet of participants (w, p, r). The number of certificate tokens produced is decided initially, but their quantity is not as easy to decide as for constraint tokens since it is not capped by a θ threshold. For simplicity, we assume that there is at least one constraint in the system, and the smallest threshold for all constraints is θ_{min}. Then, θ_{min} is a sufficient upper bound of the number of crowdworking processes in which any given triplet of participants is involved. It is therefore enough to produce $\theta_{min} \times |\mathcal{W}| \times |\mathcal{P}| \times |\mathcal{R}|$ certificate tokens. In practice, the number of tokens produced can

[11]Extending tokens with labels and/or timestamps for supporting the validity periods of constraints is straightforward.

[12]The use of a public key generated by the registration authority is important here because (1) it can be shared among participants without disclosing their identities, i.e., it is a pseudonym, (2) the corresponding private key can be used by participants for mutual authentication in order to guarantee the correctness of the index and consequently of the choice of tokens.

be drastically reduced in a straightforward manner by letting participants declare to RA the subset of participants they may work with (*e.g.,* selecting a subset of platforms, or domains of interest).

As stated above, a certificate token always relates to a fully specified triplet (w, p, r) and to its owner o (i.e., one of the participants in the triplet). Similar to a constraint token, it consists of a public and a private component. The public component consists of a nonce as well as the signature of the nonce produced by the registration authority. The private component, on the other hand, is a triplet in which each element certifies (i.e., signs) the association between the owner o and another specified participant. More formally, let tk^* be a certificate token, tk^*_{pub} be its public part, tk^*_{priv} be its private part, N be a nonce, o be the identity of the participant owner of the token, and (w, p, r) be the related triplet. The certificate token is thus the pair $(tk^*_{pub}, tk^*_{priv})$ where $tk^*_{pub} = (N, Sign(key_{priv,RA}, (N)))$ and $tk^*_{priv} = (Sign(key_{priv,RA}, (N, o, w)), Sign(key_{priv,RA}, (N, o, p)), Sign(key_{priv,RA}, (N, o, r)))$.

The SPEND Function

Requesters create and send their tasks to a platform and the platform submits the tasks in either its own datastore (for local tasks) or all involved platforms (for cross-platform tasks). Once the task is published, the workers can indicate their intent to perform the task by sending a *contribution intent* to their platforms. If a contribution is still needed for the task, the `SPEND` function is performed as follow. First, for a given constraint $c \in \mathcal{C}$, the platform requests the public component of a constraint token corresponding to c from the *initiator* (one of the specified participants in constraint c). For certificates, the platform is the initiator. The platform includes the task, an identifier for the contribution (e.g., a nonce generated by the platform), and a signature of the identifier concatenated to the task in its request message. Therefore, workers and

requesters will be able to prove that they were asked for tokens, even if the platform fails. The initiator then chooses a token to spend and sends it to the platform. Once the platform receives the required token, it sends the public component of the constraint token to all the specified participants. For certificates, the platform sends the public part of the certificate tokens to all participants of the process. The platform also requires them to send back two signatures: (1) the group signature of the token (which will be later verified by all platforms, together with the token, when it is shared with all platforms), and (2) the group signature of the pair consisting of a token and a task. Note that the second signature while it does not reveal the task by itself, can be used by participants to verify that tokens are used on the task they are intended to be used on. Again, this demand is associated with the task and the identifier of the contribution, and is signed by the platform.

Finally, for each task, a transaction consisting of all spent constraint tokens from each specified participant (all spent certificate tokens from every participant in the case of certificates) is committed to the datastore of all platforms. For each token, the transaction includes first, the public component of each token, second, the group signature of the public component of each token (i.e., for a constraint token $tk^{\dagger}$: $GroupSign(key_{priv,part}, Group, tk^{\dagger}_{pub})$), and third, the group signature of the public part of each token together with the associated task t (i.e., for a constraint token $tk^{\dagger}$: $GroupSign(key_{priv,part}, Group, (t, tk^{\dagger}_{pub}))$).

The PROVE function

The `PROVE` function is used by participants to provide certificates to a third party. The use of certificate tokens is relatively straightforward. During the crowdworking processes, participants store the private components of certificate tokens which will be used later to deliver certificates on demand. A participant indeed initiates the `PROVE` function by

sending the related subpart(s) of the private component of the corresponding tokens to the verifier. As an example, for a $((w, *, *), 5)$ request for certificates, the worker w sends the subparts containing w from the private parts of all 5 certificate tokens. The verifier, first, checks the signature of the registration authority to verify that the participant was involved in the task, and then, checks the nonce stored in the datastore to ensure that the token has been shared and validated by all platforms.

The CHECK and ALERT Functions

The `CHECK` and `ALERT` functions are used to detect and report either the malicious behavior of participants resulting in an invalid consumption of tokens or the failure of a platform. The complete set of verifications protects against (1) the forgery of tokens (verification of the signatures), (2) the replay of tokens (verification of the absence of double-spending), (3) the relay of tokens (verification of the absence of usurpation), and (4) the illegitimate invalidation of tokens (timeout against malicious platform failures). The first two verifications are straightforward and performed during the global consensus (when all platforms can access tokens and signatures). We explain the last two verifications.

Usurpation. When a token is appended to the datastore of all platforms, anyone (whether involved in the corresponding crowdworking process or not) can `CHECK` its nonce. If a participant detects a nonce that was received from the registration authority but not spent[13], she `ALERT`s the registration authority. The registration authority will de-anonymize the group-signature of the corresponding participant (e.g., the worker's group signature if the alert comes from a worker), and checks whether it has been signed by the same participant that sent the token. Similarly, if a participant detects that a

[13]For example, a platform p can collude with a worker w_1 to spend a token dedicated to a $(w_2, p, *)$ constraints.

token has not been spent on the right task, she `ALERT`s the registration authority. After an alert, the registration authority has to act either against the target participant of the alert (true positive) or against the participant originating the alert (false positive). The possible actions (e.g., ban the participant) depend on the context.

Platform failure. If a platform fails after it requested tokens or signatures and does not recover (e.g., tokens are not appended to the datastore), the tokens revealed to the platform are lost: they cannot be used in any other crowdworking process because they are not anonymous anymore (i.e., the platform knows the association between them and the corresponding participants), and they are not spent either. In that case, workers or requesters send an `ALERT` to the registration authority including (1) the identifier of the platform, (2) the identifier of the task, and (3) all the requests received from the platform. The registration authority then checks whether the number of requests sent by the platform for the given task matches the corresponding number of messages committed in the datastore. If there are more requests, the registration authority sets a timeout (e.g., to let unfinished transactions end or the platform recover from a failure). When the timeout is over, the registration authority can act against the platform.

4.4.2 Task Processing Sequence

In summary, five main phases exist during the processing of a crowdworking task are: (1) initialization, (2) publication, (3) assertion, (4) verification, and (5) execution.

Initialization. The registration authority provides all parties with their keys and tokens.

Publication. Requesters create and send their tasks to platforms. If a requester wants to publish its task on more that one platform (i.e., a cross-platform task), the involved platforms collaborate with each other to create a common instance of the task. The involved platforms then publish the tasks on their datastores through submission transac-

tions and inform their workers in their preferred manner for accessing tasks.

Assertion. After a worker has retrieved a task, the worker sends a *contribution intent* message to the platform without revealing the actual contribution. The platform then updates the number of required contributions for the task and publishes the contribution intent in its datastore through a claim transaction. For cross-platform tasks, the platform informs other involved platforms about the received contribution intent, so that all involved platforms agree with the number (and order) of the received contribution intents (i.e., claim transactions). If the desired number of contribution for the task has been achieved, the process is aborted. Note that while the requester does not choose the workers, it is possible to enforce a selection with *a priori* criteria, passed through the platform. Another straight-forward enhancement would be to add a communication step, by forwarding the contribution intent, together with the worker's identity to the requester, and letting her approve of it or not. This communication, however, requires the disclosure of the worker's identity to requesters even before a contribution is accepted.

Verification. Once the contribution intent has been accepted by the platform(s), the platform asks the corresponding requester and worker to send the required tokens and signatures, through the `SPEND` function, developed in Section 4.4.1. Upon receiving all tokens and signatures, the platform shares them with all platforms and the tokens and their signatures are published to the datastores through verification transactions. From this point, anyone can check the validity of requirements with the `CHECK` function (and `ALERT` if required), as developed in Section 4.4.1.

Execution. Once all parties have checked the validity of the task, its tokens and group signatures, the actual contribution can be given to the requester and reward to the worker through the platform.

A sequence chart of this protocol is provided in Figure 4.2. The privacy of SEPAR against covert adversaries is shown in [1].

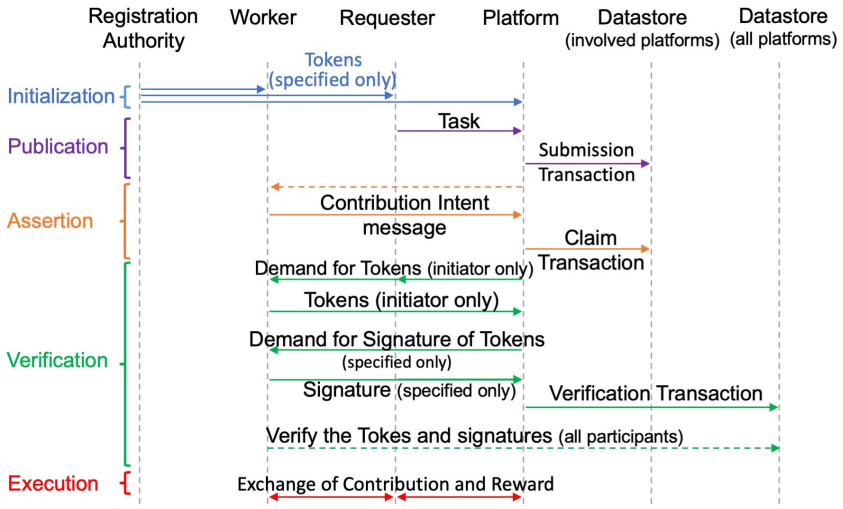


Figure 4.2: Sequence chart of SEPAR
(references to specified participants include all participants for certificate tokens)

4.5 Coping with Distribution

SEPAR is a multi-platform crowdworking system where multiple globally distributed platforms collaborate with each other to process crowdworking tasks. To realize such distributed collaborations and due to the unique features of permissioned blockchains such as transparency and provenance, which are needed by crowdworking applications, SEPAR is deployed on a permissioned blockchain to implement the persistent datastore. In this section, we first present the distributed blockchain ledger of SEPAR and then, show how SEPAR establishes consensus on the order of transactions within and across different platforms.

4.5.1 Blockchain Ledger

The blockchain ledger in SEPAR includes all submission, claim, and verification transactions of all internal as well as cross-platform tasks. To ensure data consistency, an ordering among transactions in which a platform is involved is needed. The total or-

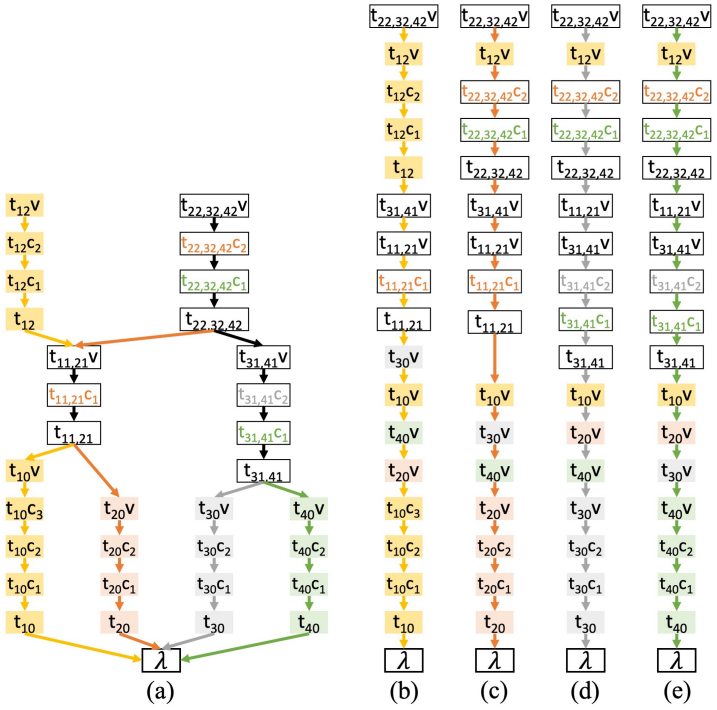


Figure 4.3: (a): A blockchain ledger, (b)-(e): Different views of the blockchain

der of transactions in the blockchain ledger is captured by *chaining* the transactions (blocks) together, i.e. each transaction block includes a sequence number or the cryptographic hash of the previous transaction block. Since SEPAR supports both internal and cross-platform tasks and more than one platform are involved in each cross-platform transaction, similar to CAPER, the ledger is formed as a *directed acyclic graph (DAG)* where the *nodes* of the graph are transaction blocks (each block includes a single transaction) and *edges* enforce the order among transaction blocks. In addition to submission, claim, and verification transactions, a unique initialization transaction (block), called the *genesis* transaction is also included in the ledger.

Fig. 4.3(a) shows a blockchain ledger created in the SEPAR model for a blockchain infrastructure consisting of four platforms p_1, p_2, p_3, and p_4. In this figure, λ is the genesis block of the blockchain, t_i's are submission transactions, $t_i c_j$ is the j-th claim transaction

of task t_i, and t_iv is the verification transaction of task t_i. In Fig. 4.3(a), t_{10}, t_{20}, t_{30}, and t_{40} are internal submission transactions of different platforms. In SEPAR, as can be seen, the internal transactions of different platforms can be appended to the ledger in parallel. $t_{10}c_1$, $t_{10}c_2$, ..., and $t_{40}c_2$ are the corresponding claim transactions. As shown, t_{10} requires 3 contributions (thus 3 claim transactions) whereas each of t_{20}, t_{30}, and t_{40} needs two contributions. $t_{10}v$, $t_{20}v$, $t_{30}v$, and $t_{40}v$ are also the verification transactions. $t_{11,21}$ is a cross-platform submission among platforms p_1 and p_2. Similarly, $t_{31,41}$ is a cross-platform submission among platforms p_3 and p_4. Here, $t_{11,21}$ needs a single contribution and $t_{31,41}$ requires two contributions. Note that the claim transactions of a cross-platform task might be initiated by different platforms and as mentioned earlier, the order of these claim transactions is important (to recognize the n first claims). Finally, $t_{22,32,42}$ is a cross-platform task among platforms p_2, p_3, and p_4 that is processed in parallel to the internal task t_{12} of platform p_1.

The introduced blockchain ledger includes all transactions of internal as well as cross-platform tasks initiated by all platforms. However, due to the data privacy requirement, each platform must access only a subset of these transactions, i.e., the transactions in which the platform is involved. One way to achieve data privacy is to encrypt *all* transactions using cryptographic techniques and keep an identical blockchain ledger on every platform. However, the considerable overhead of such techniques results in low performance [27], and in addition, each platform will store many transactions that are not relevant to the platform. As a result and for the sake of performance, in SEPAR, similar to CAPER, the entire blockchain ledger is *not maintained* by any platform and each platform only maintains its own *view* of the blockchain ledger including (1) all submission and claim transactions of its internal tasks, (2) all submission and claim transactions of the cross-platform tasks that the platform is involved in them, and (3) verification transactions of all tasks. Note that verification transactions are replicated on every platform to enable

all platforms to check the satisfaction of constraints. The blockchain ledger is indeed the union of all these physical views.

Fig. 4.3(b)-(e) show the views of the blockchain ledger for platforms p_1, p_2, p_3, and p_4 respectively. As can be seen, each platform p_i maintains only submission and claim transactions of all internal tasks as well as cross-platform tasks that p_i is involved in them and verification transactions of all tasks. For example and as shown in Fig. 4.3(b), platform p_1 maintains all transactions of its two internal tasks t_{10} and t_{12}. These are either submission transactions, i.e., t_{10} and t_{12}, or claim transactions, i.e., $t_{10}c_1$, $t_{10}c_2$, $t_{10}c_3$, $t_{12}c_1$, $t_{12}c_2$, or verification transactions, i.e., $t_{10}v$, $t_{12}v$. Platform p_1 also maintains cross-platform transactions that p_1 is involved in, i.e., $t_{11,21}$, $t_{11,21}c_1$, and $t_{11,21}v$. Finally, p_1 maintains the verification transactions of all other tasks within the system, i.e. $t_{20}v$, $t_{30}v$, $t_{40}v$, $t_{31,41}v$, and $t_{22,32,42}v$. Note that, since there is no data dependency between the verification transactions of the tasks that a platform is not involved in and the transactions of the tasks that a platform is involved in, the verification transactions might be appended to the ledgers in different orders, e.g., $t_{20}v$ (of platform p_2) and $t_{40}v$ (of platform p_4) are appended to the ledger of platforms p_1 and p_3 in two different orders.

4.5.2 Consensus in SEPAR

In SEPAR, each platform consists of a (disjoint) set of nodes (i.e., replicas) where the platform replicates its own view of the blockchain ledger on those nodes to achieve fault tolerance where depending on the failure model of nodes, asynchronous crash or Byzantine fault-tolerant protocols can be used. Figure 4.4 shows the crowdworking infrastructure of Figure 4.4 where each platform consists of 4 replicas (assuming Byzantine failure model and $f = 1$) and replicas use a blockchain to store data.

Completion of a crowdworking task, as discussed earlier, requires a single submission,

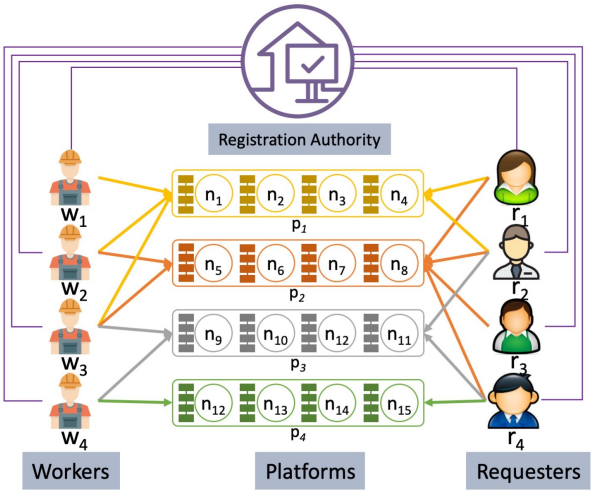


Figure 4.4: SEPAR infrastructure

one or more claim, and a verification transaction. For an internal task of a platform, submission and claim transactions are replicated only on the nodes of the platform, hence, *local consensus* among nodes of the platform on the order of the transaction is needed. For a cross-platform task, on the other hand, submission and claim transactions are replicated on every node of all (and only) involved platforms. As a result, *cross-platform consensus* among the nodes of all *involved* platforms is needed. Finally, verification transactions will be appended to the blockchain of all platforms, therefore, all nodes of *every* platform participate in a *global consensus* protocol. In this section, we show how local, cross-platform, and global consensus are established in the presence of crash-only or Byzantine nodes.

Local Consensus

Processing a submission or a claim transaction of an internal task requires local consensus where nodes of a single platform, *independent* of other platforms, establish agreement on the order of the transaction. The local consensus protocol in SEPAR is pluggable

and depending on the failure model of nodes, i.e., crash-only or Byzantine, a platform uses a crash fault-tolerant protocol, e.g., Paxos [20], or a Byzantine fault-tolerant protocol, e.g., PBFT [54].

The local consensus protocol is initiated by a pre-elected node of the platform, called *the primary*. When the primary p receives a valid internal transaction (either submission or claim), it initiates a local consensus algorithm by multicasting a message, e.g., accept message in Paxos or pre-prepare message in PBFT, including the requested transaction to other nodes of the platform. To provide a total order among transactions, the primary also assigns a sequence number to the request. Instead of a sequence number, the primary can also include the cryptographic hash of the previous transaction block in the message. If the transaction is a claim transaction, the primary includes the cryptographic hash of the corresponding submission transaction and any previously received claim transactions for that particular task (if any). The nodes of the platform then establish agreement on a total order of transactions using the utilized consensus protocol and append the transaction to the blockchain ledger.

Cross-Platform Consensus

Submission and claim transactions of a cross-platform task must be appended to the blockchains of *all* involved platforms in the same order to ensure data consistency. To process such transactions, therefore, consensus among the nodes of all (and only) involved platforms is needed. SEPAR addresses the lack of trust in the collaboration between platforms, by using an asynchronous Byzantine fault-tolerant protocol to establish consensus on the order of cross-platform transactions. Since the number of nodes of each platform depends on the utilized consensus protocol within the platform (i.e. crash fault-tolerant protocols require $2f+1$ whereas Byzantine fault-tolerant protocols require $3f+1$ nodes), the required number of matching replies from each platform, i.e., the quorum size, to en-

Algorithm 4 Cross-Platform Consensus

1: **init():**
2: $r :=$ *node_id*
3: $p_i :=$ the platform that initiates the consensus
4: $\pi(p) :=$ the primary node of cluster p
5: $P :=$ the set of involved platforms
6: $\pi(P) :=$ the primary nodes of clusters in P

7: **upon receiving** valid transaction m and $(r == \pi(p_i))$
8: **multicast** $\langle\langle \mathsf{PREPARE}, h_i, d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ to $\pi(P)$
9: **multicast** $\langle\langle \mathsf{PROPOSE}, h_i, d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ to all nodes of p_i

10: **upon receiving** valid $\mu = \langle\langle \mathsf{PREPARE}, h_i, d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ and $r == \pi(p_j)$
11: if r is not involved in any uncommitted request m' where m and m' intersect in some other platform p_k
12: **multicast** $\langle\langle \mathsf{PROPOSE}, h_j, d, r\rangle_{\sigma_{\pi(p_j)}}, \mu\rangle$ to all nodes of p_j
13: **multicast** $\langle \mathsf{ACCEPT}, h_i, h_j, d, r\rangle_{\sigma_{\pi(p_j)}}$ to P

14: **upon receiving** valid $\langle\langle \mathsf{PROPOSE}, h_i, d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ and $r \in p_i$
15: **multicast** $\langle \mathsf{ACCEPT}, h_i, d, r\rangle_{\sigma_r}$ to P

16: **upon receiving** valid $\langle\langle \mathsf{PROPOSE}, h_j, d, r\rangle_{\sigma_{\pi(p_j)}}, \mu\rangle$ and $r \in p_j$
17: **multicast** $\langle \mathsf{ACCEPT}, h_i, h_j, d, r\rangle_{\sigma_r}$ to P

18: **upon receiving** valid matching $\langle \mathsf{ACCEPT}, h_i, h_j, d, r\rangle_{\sigma_r}$ from *local-majority* of every platform p_j in P
19: **multicast** $\langle \mathsf{COMMIT}, h_i, h_j, ..., h_k, d, r\rangle_{\sigma_r}$ to P

20: **upon receiving** valid $\langle \mathsf{COMMIT}, h_i, h_j, ..., h_k, d, r\rangle_{\sigma_r}$ from *local-majority* of every platform in P
21: **append** the transaction block to the ledger

sure the safety of protocol depends on the failure model of nodes of the platform. We define *local-majority* as the required number of matching replies from the nodes of a platform. For a platform with crash-only nodes, local-majority is $f + 1$ (from the total $2f + 1$ nodes), whereas for a platform with Byzantine nodes, local-majority is $2f + 1$ (from the total $3f + 1$ nodes).

SEPAR processes cross-platform transactions in four phases: prepare, propose, accept, and commit. Upon receiving a cross-platform (submission or claim) transaction, the (pre-elected) primary node of the (recipient) platform initiates the consensus protocol by multicasting a prepare message to the primary node of all involved platforms. Each primary node then assigns a sequence number to the request and multicasts a propose message to every node of its platform. During the accept and commit phases, all nodes of every involved platform communicate to each other to reach agreement on the order of the cross-platform transaction.

Algorithm 4 presents the normal case of *cross-platform consensus* in SEPAR. Although not explicitly mentioned, every sent and received message is logged by nodes. As shown in lines 1-6 of the algorithm, p_i is the platform that initiates the transaction, $\pi(p)$ represents the primary node of platform p, P is the set of involved platforms in the transaction where $\pi(P)$ represents their current primary nodes (one node per platform).

Once the primary $\pi(p_i)$ of the initiator platform p_i receives a valid submission or claim transaction, as presented in lines 7-8, the primary node assigns sequence number h_i to the request and multicasts a *signed* prepare message $\langle\langle \textsf{PREPARE}, h_i, d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ to the primary nodes of all involved platforms where m is the received message (either submission or claim) and $d = D(m)$ is the digest of m. The sequence number h_i represents the correct order of the transaction block in the initiator platform p_i. If the transaction is a claim transaction, the primary includes the cryptographic hash of the corresponding submission transaction as well. As shown in line 9, the primary node also multicasts a *signed* propose message $\langle\langle \textsf{PROPOSE}, h_i, d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ to the nodes of its platform where $d = D(m)$ is the digest of m.

As indicated in lines 10-12, once the primary node of some platform p_j receives a prepare message μ from the primary node of the initiator platform, it first validates the message. If node r is currently waiting for a commit message of some cross-platform transaction m' where the involved platforms of the two requests m and m' intersect, the node does not process the new transaction m before the earlier transaction m' gets committed. This ensures that requests are committed in the same order on different platforms. Otherwise, it assigns sequence number h_j to the message and multicasts a *signed* propose message $\langle\langle \textsf{PROPOSE}, h_j, d\rangle_{\sigma_{\pi(p_j)}}, \mu\rangle$ to the nodes of its platform. The primary node $\pi(p_j)$ also piggybacks the prepare message μ to its propose message to enable the node to access the request and validate the propose message. The primary node $\pi(p_j)$,

as presented in line 13, multicasts a signed accept message $\langle \text{ACCEPT}, h_i, h_j, d \rangle_{\sigma_{\pi(p_j)}}$ to *every* node of *all* involved platforms.

Upon receiving a propose message Once a node r of an involved platform p_j receives a propose message, as indicated in lines 8-10, it validates the signature and message digest (if the node belongs to the initiator platform ($i = j$), it also checks h_i to be valid (within a certain range)) since a malicious primary might multicast a request with an invalid sequence number. In addition, if the node is currently involved in an uncommitted cross-platform request m' where the involved platforms of two requests m and m' overlap in some other platform, the node does not process the new request m before the earlier request m' is processed. This is needed to ensure requests are committed in the same order on different platforms. The node then multicasts a signed accept message including the corresponding sequence number h_j (that represents the order of m in platform p_j), and the digest $d = D(m)$ to *every* node of *all* involved platforms.

As presented in lines 18-19, each node waits for valid matching accept messages from a local majority (i.e., either $f + 1$ or $2f + 1$ depending on the failure model) of *every* involved platform with h_i and d that matches the propose message which was sent by primary $\pi(p_i)$. We define the predicate $\text{accepted-local}_{p_j}(m, h_i, h_j, r)$ to be true if and only if node r has received the request m, a propose for m with sequence number h_i from the initiator platform p_i and accept messages from a local majority of an involved platform p_j that match the propose message. The predicate $\text{accepted}(m, h, r)$ where $h = [h_i, h_j, ..., h_k]$ is then defined to be true on node r if and only if $\text{accepted-local}_{p_j}$ is true for *every* involved platform p_j in cross-platform request m. The order of sequence numbers in the predicate is an ascending order determined by their platform ids. The propose and accept phases of the algorithm basically guarantee that non-faulty nodes agree on a total order for the transactions. When $\text{accepted}(m, h, v, r)$ becomes true, node r multicasts a signed commit message $\langle \text{COMMIT}, h, d, r \rangle_{\sigma_r}$ to all nodes of every involved platforms.

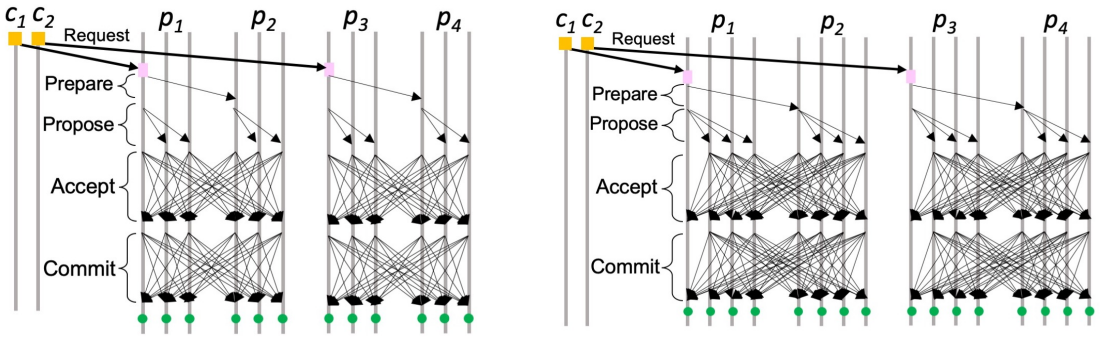


Figure 4.5: Cross-platform transactions with (a) crash-only and (b) Byzantine nodes

Finally, as shown in lines 20-21, node r waits for valid matching commit messages from a local majority of *every* involved platform that matches its commit message. The predicate committed-local$_{p_j}(m, h, r)$ is defined to be true on node r if and only if accepted(m, h, r) is true and node r has accepted valid matching commit messages from a local majority of platform p_j that match the propose message for cross-platform transaction m. The predicate committed(m, h, v, r) is then defined to be true on node r if and only if committed-local$_{p_j}$ is true for *every* involved platform p_j in cross-platform transaction m. The committed predicate indeed shows that at least $f + 1$ nodes of each involved platform have multicast valid commit messages. When the committed predicate becomes true, the node considers the transaction as committed. If all transactions with lower sequence numbers than h_j have already been committed, the node appends a transaction block including the transaction as well as the corresponding commit message to its copy of the ledger.

Figure 4.5 shows the normal case operation for SEPAR to execute two concurrent cross-platform transactions in the presence of (a) crash-only and (b) Byzantine nodes where each transaction accesses two disjoint platforms. The network consists of four platforms where each platform includes either three or four nodes ($f = 1$).

In addition to the normal case operation, SEPAR has to deal with two other scenarios. First, when the primary node fails. Second, when nodes have not received a quorum of *matching* accept messages from the local-majority of every involved platform due to con-

Algorithm 5 Global Consensus

1: **init():**
2: r := *node_id*
3: p_i := the platform that initiates the consensus
4: $\pi(p)$:= the primary node of cluster p

5: **upon receiving** valid transaction m and $(r == \pi(p_i))$
6: **multicast** $\langle\langle \mathsf{PREPARE}, h_i, d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ to the primary node of every cluster
7: **multicast** $\langle\langle \mathsf{PROPOSE}, h_i, d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ to all nodes of p_i

8: **upon receiving** valid μ= $\langle\langle \mathsf{PREPARE}, h_i, d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ and r==$\pi(p_j)$
9: if r is not involved in any uncommitted request m' where m and m' intersect in some other platform p_k
10: **multicast** $\langle\langle \mathsf{PROPOSE}, h_j, d, r\rangle_{\sigma_{\pi(p_j)}}, \mu\rangle$ to all nodes of p_j
11: **multicast** $\langle \mathsf{ACCEPT}, h_i, h_j, d, r\rangle_{\sigma_{\pi(p_j)}}$ to all nodes

12: **upon receiving** valid $\langle\langle \mathsf{PROPOSE}, h_i, d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ and $r \in p_i$
13: **multicast** $\langle \mathsf{ACCEPT}, h_i, d, r\rangle_{\sigma_r}$ to all nodes

14: **upon receiving** valid $\langle\langle \mathsf{PROPOSE}, h_j, d, r\rangle_{\sigma_{\pi(p_j)}}, \mu\rangle$ and $r \in p_j$
15: **multicast** $\langle \mathsf{ACCEPT}, h_i, h_j, d, r\rangle_{\sigma_r}$ to all nodes

16: **upon receiving** valid matching $\langle \mathsf{ACCEPT}, h_i, h_j, d, r\rangle_{\sigma_r}$ from *local-majority* of **two-thirds** of platforms
17: **multicast** $\langle \mathsf{COMMIT}, h_i, h_j, ..., h_k, d, r\rangle_{\sigma_r}$ to all nodes

18: **upon receiving** valid $\langle \mathsf{COMMIT}, h_i, h_j, ..., h_k, d, r\rangle_{\sigma_r}$ from *local-majority* of **two-thirds** of platforms
19: **append** the transaction block to the ledger

flicting accept messages. Indeed, the primary nodes of different platforms might multicast their propose messages in parallel, hence, different overlapping platforms might receive the messages in different order. Furthermore, nodes might assign inconsistent sequence numbers since they have not necessarily received the latest propose message from the primary of their own platform. We use the similar techniques as SharPer to address these two situations (will be explained in Sections 6.3 and 6.4).

Global Consensus

The verification transactions include group signatures and all tokens that are consumed by different participants to perform a particular task. In SEPAR and in order to enable all platforms to check constraints, verification transactions are appended to the blockchains of all platforms. To do so, a Byzantine fault-tolerant protocol is run among all nodes of every platform where the protocol needs agreement from the *local majority* of the nodes of *two-thirds* of the platforms. The local majority, similar to cross-platform con-

sensus, is defined based on the utilized consensus protocol within each platform. However, there are two main differences between cross-platform consensus and global consensus. First, in cross-platform consensus only the involved platforms participate, whereas, in global consensus, every platform verifies transactions by checking the group signatures and consumed tokens. Second, cross-platform consensus requires agreement from every platform, whereas, in global consensus, agreement from only two-thirds of platforms is needed. In fact, in cross-platform consensus, there might be some dependency between cross-platform transactions and internal ones, thus, to ensure data consistency, every involved platform must agree on the order of the cross-platform transaction. However, in global consensus, the goal is to verify the correctness of the transaction and as soon as two-thirds of platforms verify that (assuming at most one-third of platforms might behave maliciously), the transaction can be appended to the blockchain ledger.

Algorithm 5 shows the normal case of *global consensus* in SEPAR where a Byzantine protocol is run among all nodes of every platform (in contrast to cross-platform consensus where only the involved platforms participate). The protocol, similar to cross-platform consensus, process a transaction in four phases of prepare (lines 5-6), propose (lines 7-10), accept (lines 11-15), and commit (lines 16-19), however, each node waits for matching accept and commit messages from the local majority of only *two-thirds* of the platforms (as shown in lines 16 and 18).

Figure 4.6 presents the normal case operation of global consensus in SEPAR. Here all platforms include crash-only nodes where $f = 1$ and the network consists of four platforms.

Similar to cross-consensus, global consensus also addresses primary failure and conflicting transactions in a similar way as SharPer (the details will be explained in Sections 6.3 and 6.4).

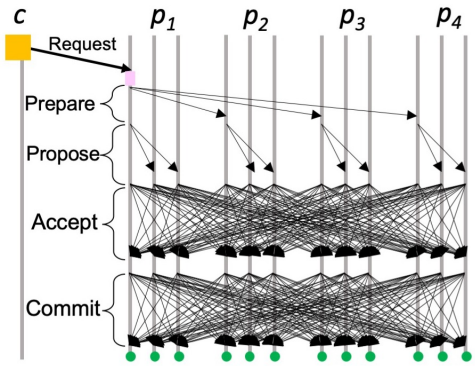


Figure 4.6: Global consensus in SEPAR

Correctness Arguments

A consensus protocol, as mentioned before, has to satisfy four main properties [52]: (1) *agreement:* every correct node must agree on the same value (Lemma 4.5.1), (2) *Validity (integrity):* if a correct node commits a value, then the value must have been proposed by some correct node (Lemma 4.5.2), (3) *Consistency (total order):* all correct nodes commit the same value in the same order (Lemma 4.5.3), and (4) *termination:* eventually every node commits some value (Lemma 4.5.4). The first three properties are known as *safety* and the termination property is known as *liveness*. In an asynchronous system, where nodes can fail, as shown by Fischer et al. [53], consensus has no solution that is both safe and live. Therefore, SEPAR guarantees safety in an asynchronous network, however, similar to most fault-tolerant protocols, deals with termination (liveness) only during periods of synchrony using timers.

Lemma 4.5.1 (*Agreement*) If node r commits request m with sequence number h, no other correct node commits request m' ($m \neq m'$) with the same sequence number h.

Proof: The propose and accept phases of both cross-platform and global consensus protocols guarantee that correct nodes agree on a total order of requests. Indeed, if the accepted(m, h, r) predicate where $h = [h_i, h_j, ..., h_k]$ is true, then accepted(m', h, q)

is false for any non-faulty node q (including $r = q$) and any m' such that $m \neq m'$. This is true because (m, h, r) implies that $\mathsf{accepted\text{-}local}_{p_j}(m, h_i, h_j, r)$ is true for each involved platform p_j and a local majority ($f+1$ crash-only or $2f+1$ Byzantine node) of platform p_j have sent accept (or propose) messages for request m with sequence number h_j. As a result, for $\mathsf{accepted}(m', h, q)$ to be true, at least one non-faulty nodes needs to have sent two conflicting accept messages with the same sequence number but different message digest. This condition guarantees that first, a malicious primary cannot violate the safety and second, at most one of the concurrent *conflicting* transactions, i.e., transactions that overlap in at least one platform, can collect the required number of messages from each overlapping platform. ■

Lemma 4.5.2 (*Validity*) If a correct node r commits m, then m must have been proposed by some correct node π.

Proof: In the presence of crash-only nodes, validity is ensured since crash-only nodes do not send fictitious messages. In the presence of Byzantine nodes, however, validity is guaranteed mainly based on standard cryptographic assumptions about collision-resistant hashes, encryption, and signatures which the adversary cannot subvert them. Since the request as well as all messages are signed and either the request or its digest is included in each message (to prevent changes and alterations to any part of the message), and in each step $2f + 1$ matching messages (from each Byzantine platform) are required, if a request is committed, the same request must have been proposed earlier. ■

Lemma 4.5.3 (*Consistency*) Let P_μ denote the set of involved platforms for a request μ. For any two committed requests m and m' and any two nodes r_1 and r_2 such that $r_1 \in p_i$, $r_2 \in p_j$, and $\{p_i, p_j\} \in P_m \cap P_{m'}$, if m is committed before m' in r_1, then m is committed before m' in r_2.

Proof: once a node r_1 of some platform p_i receives a propose message for some transaction m, in both cross-platform and global consensuses, if the node is involved in some other uncommitted transaction m' where m and m' overlap, node r_1 does not send an accept message for transaction m before m' gets committed. In this way, since committing request m requires accept messages from a local majority of *every* (involved) platform, m cannot be committed until m' is committed. As a result the order of committing messages is the same in all involved platforms. ■

It should be noted that in such a case we might face a deadlock where different platforms might need to re-initiate their transactions after some predefined time. To prevent any further deadlock, SEPAR defines different waiting times for different platforms.

Lemma 4.5.4 (*Termination*) A request m issued by a correct client eventually completes.

Proof: SEPAR deals with termination (liveness) only during periods of synchrony using timers. To do so, three scenarios need to be addressed. If the primary is non-faulty and accept messages are non-conflicting, following the normal case operation of the protocol, request m completes. If the primary is non-faulty, but accept messages are conflicting, the request will be re-initiated. Finally, SEPAR includes a routine to handle primary failures. SEPAR, as explained before, Handles conflicting messages and primary failures in a similar way as SharPer [2] (will be explained in Sections 6.3 and 6.4). ■

4.6 Experimental Evaluations

In this section, we conduct several experiments to evaluate SEPAR. We have implemented a blockchain-based multi-platform crowdworking system. For the purpose of this evaluation, and as explained earlier, we do not focus on the description of tasks and

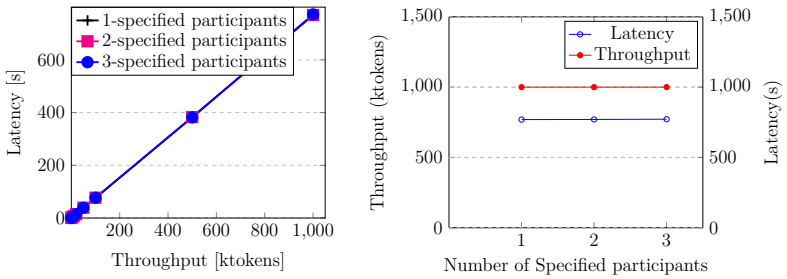
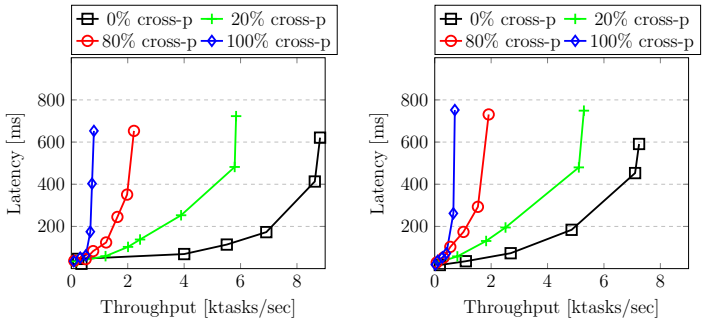


Figure 4.7: Token generation performance

contributions (both are modeled as arbitrary bitstrings). In addition, certificate tokens, as explained earlier, are very similar to $((w, p, r), \theta)$ tokens except for the private part that has no significant impact on the performance and the number of interaction phases which is even less than constraint tokens. Therefore, we only focus on constraint tokens in the experiments. To implement group signatures, we use the protocol proposed in [68]. The experiments were conducted on the Amazon EC2 platform. Each VM is c4.2xlarge instance with 8 vCPUs and 15GB RAM, Intel Xeon E5-2666 v3 processor clocked at 3.50 GHz. When reporting throughput measurements, we use an increasing number of tasks submitted by requesters running on a single VM, until the end-to-end throughput is saturated, and state the throughput and latency just below saturation.

4.6.1 Token Generation

In the first set of experiments, we measure the performance of token generation in SEPAR for different types of constraints. We consider constraints with a single specified participant (e.g., $((w, *, *), \theta)$), two specified participants (e.g., $((*, p, r), \theta)$), and three specified participants (e.g., $((w, p, r), \theta)$). As shown in Figure 4.7(a), SEPAR is able to generate tokens in linear time. SEPAR generates each token in $0.7ms$, hence, generating 1 million tokens in 12 minutes. This is an acceptable amount of time since token generation is executed periodically, e.g., every week or every month. Note that since



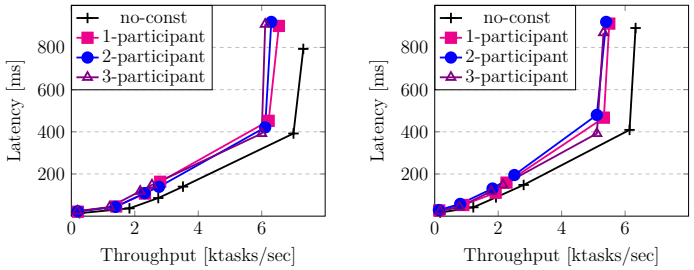
(a) Crash-only nodes (b) Byzantine nodes

Figure 4.8: Performance with different percentage of cross-platform tasks

tokens of different constraints can be generated in parallel, SEPAR can easily parallelize the token generation routine in order to improve the throughput. As can be seen in Figure 4.7(b), the type of constraints, i.e., the number of specified participants, also does not affect the performance and the token generation throughput and latency is constant in terms of the number of participant. However, it should be noted that a more complicated constraint, i.e., a constraint with more specified participants, requires more tokens to be generated.

4.6.2 Performance with different Percentage of Cross-Platform Tasks

In the second set of experiments, we measure the performance of SEPAR for workloads with different percentages of cross-platform tasks. We consider four different workloads with (1) no cross-platform tasks, (2) 20% cross-platform tasks, (3) 80% cross-platform tasks, and (4) 100% cross-platform tasks. We also assume that two (randomly chosen) platforms are involved in each cross-platform tasks and completion of each task requires a contribution coming from a randomly chosen worker. The system includes four platforms and each task has to satisfy two randomly chosen constraints. We consider two different networks with crash-only and Byzantine nodes. When all nodes follow crash-only nodes,



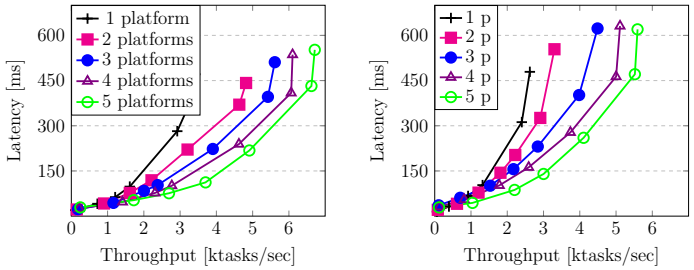
(a) Crash-only nodes (b) Byzantine nodes

Figure 4.9: Performance with different types of constraints

as presented in Figure 4.8(a), SEPAR is able to process 8600 tasks with 400 ms latency before the end-to-end throughput is saturated (the penultimate point), if all tasks are local. Note that even when all tasks are local, the verification transaction of each task still needs global consensus among all platforms. Increasing the percentage of cross-platform tasks to 20%, reduces the overall throughput to 5800 (67%) with 400 ms latency since processing cross-platform tasks requires cross-platform consensus. By increasing the percentage of cross-platform tasks to 80% and then 100%, the throughput of SEPAR will reduce to 1900 and 700 with the same (400 ms) latency. This is expected because when most tasks are cross-platform ones, more nodes are involved in processing a task and more messages are exchanged. In addition, the possibility of parallel processing of tasks will be significantly reduced. In the presence of Byzantine nodes, as shown in Figure 4.8(b), SEPAR demonstrates the similar behavior as the previous case (crash-only nodes). When all tasks are local, SEPAR processes 7100 tasks with 450 ms latency. Increasing the percentage of cross-platform tasks to 20% and 80% will reduce the throughput to 4900 and 1700 tasks with the same (450 ms) latency respectively. Finally, when all tasks are cross-platform, SEPAR is able to process 700 tasks with 450 ms latency.

4.6.3 Performance with different Types of Constraints

In the next set of experiments, we measure the performance of SEPAR with different types of constraints. We consider four different scenarios where each task has to satisfy (1) no constraints (i.e., basic scenario), (2) a one-specified constraint, (3) a two-specified constraint, and (4) a three-specified constraint. The system consists of four platforms and the workload includes 90% intra- and 10% cross-platform tasks (the typical settings in partitioned databases [60,61]) where two (randomly chosen) platforms are involved in each cross-platform tasks. As before, completion of each task requires a single contribution. To measure the overhead of group signatures and tokens, we compare the results with the basic scenario where there is no constraints in the system, thus, there is no need to exchange and validate tokens and signatures. When nodes follow the crash failure model and the system has no constraints, as can be seen in Figure 4.9(a), SEPAR is able to process 7000 tasks with 390 ms latency before the end-to-end throughput is saturated (the penultimate point). Adding constraints to the tasks results in more phases of communication between different participants to exchange tokens and signatures, however, SEPAR is still able to process 6200 tasks (the penultimate point) with 450 ms latency (only 11% and 15% overhead in terms of the throughput and latency respectively). The number of participants in each constraint, on the other hand, does not significantly affect the performance of SEPAR. This is expected, because more participants results in only increasing the number of (parallel) tokens and signature exchanges and the consensus protocols and other communication phases are not affected. Similarly, in the presence of Byzantine nodes and as shown in Figure 4.9(b), SEPAR is able to process 6140 tasks with 409 ms latency with no constraints and 5331 tasks (13% overhead) with 467 ms (14% overhead) latency with one-specified constraint. As before, the number of participants does not significantly affect the performance.



(a) Crash-only nodes (b) Byzantine nodes

Figure 4.10: Performance with different number of platforms

It should be noted that increasing the number of constraints, SEPAR still demonstrates similar performance as shown in this experiment (increasing the number of participants in each constraints). Indeed, adding more constraints results in adding more tokens and possibly more participants and signatures, however, it does not affect the consensus protocols and other communication phases.

4.6.4 Performance with Different Number of Platforms

In the last set of experiments, we measure the scalability of SEPAR in crowdsourcing systems with different number of platforms. We measure the performance of SEPAR in networks including 1 to 5 platforms for both crash-only and Byzantine nodes (assuming $f = 1$ in each platform). Each task has to satisfy on average two randomly chosen constraints, two (randomly chosen) platforms are involved in each cross-platform tasks, completion of each task requires a single contribution, and the workloads include 90% intra- and 10% cross-platform tasks. Note that in the scenario with a single platform, all tasks are intra-platform. As shown in Figure 4.10(a), in the presence of crash-only nodes, the performance of the system improves by adding more platforms, e.g., with five platform, SEPAR processes 6600 tasks with 400 ms latency whereas in a single platform setting, SEPAR processes 3300 task with the same latency. While adding more platforms improves the performance of SEPAR, the relation between the increased number

of platforms and the improved throughput is non-linear (the number of platforms has been increased 5 times while the throughput doubled). This is expected because adding more platforms while increases the possibility of parallel processing of local tasks, makes the global consensus algorithm (which is needed for every single task) more expensive. In the presence of Byzantine nodes, SEPAR demonstrates similar behavior, e.g., processes 5500 tasks with 470 ms latency with 5 platforms.

4.7 Summary

In this chapter, we introduce SEPAR, a multi-platform crowdworking system that enforces global regulations in a privacy-preserving and transparent manner. SEPAR consists of two main components. First, a token-based system that enables official institutions to express legal regulations in simple and unambiguous terms, guarantees the satisfaction of global constraints by construction, and allows participants to prove to external entities their involvement in crowdworking tasks, all in a privacy-preserving manner. Second, a permissioned blockchain that provides transparency using distributed ledgers shared across multiple platforms and enables collaboration among platforms through a suite of distributed consensus protocols. To the best of our knowledge, SEPAR is the first to address the problem of enforcing global regulation over multi-crowdworking platforms. We prove the privacy requirements of the token-based system as well as the correctness of the consensus protocols and conduct an extensive experimental evaluation to measure the performance and scalability of SEPAR.

Chapter 5

ParBlockchain: On Performance of Permissioned Blockchains

5.1 Introduction

Large-scale data management systems require high performance in terms of throughput and latency, e.g., a financial application needs to process tens of thousands of requests every second with very low latency. Large-scale data management applications might also have workloads with high-contention, i.e., conflicting transactions. Under these workloads, several transactions simultaneously perform conflicting operations on a few popular records. These conflicting transactions might belong to a single application or even a set of applications using a shared datastore. While the sequential execution of transactions prevents any possible inconsistency, it adversely impacts performance and scalability.

Existing permissioned blockchains, e.g., Tendermint [33] and Multichain [32], mostly employ an *order-execute* paradigm where nodes agree on a total order of the blocks of transactions using a consensus protocol and then the transactions are executed in the

same order on all nodes sequentially. Such a paradigm suffers from performance issues because of the sequential execution of transactions on all nodes. Hyperledger Fabric [27], on the other hand, presents a new paradigm for permissioned blockchains by switching the order of the execution and ordering phases. In Hyperledger Fabric, transactions of different applications are first executed in parallel and then an ordering service consisting of a set of nodes uses a consensus protocol to establish agreement on a total order of all transactions. Fabric allows the non-deterministic execution of transactions by switching the order of the ordering and execution phases, and improves performance by executing transactions in parallel. However, in the presence of any contention in the workload, it has to disregard the effects of conflicting transactions which negatively impacts the performance of the blockchain.

In this chapter, we present *OXII*: an order-execute paradigm for permissioned blockchains. OXII is mainly introduced to support distributed applications processing workloads with *some degree of contention*. OXII consists of *orderer* and *agent* nodes. Orderers establish agreement on the order of the transactions of different applications, construct the blocks of transactions, and generate a *dependency graph* for the transactions within a block. A dependency graph, on one hand, gives a partial order based on the conflicts between transactions, and, on the other hand, enables higher concurrency by allowing the parallel execution of non-conflicting transactions. A group of agents of each application called executors are then responsible for executing the transactions of that application.

We then present *ParBlockchain*, a permissioned blockchain system designed specifically in the OXII paradigm. ParBlockchain processes transactions in the ordering and execution phases. In the ordering phase, transactions are ordered in a dependency graph and put in blocks. In the execution phase, the executors of each application execute the transactions of the corresponding application following the dependency graph. As long as the partial order of transactions in the dependency graph is preserved, the transactions

of different applications can be executed in parallel.

A key contribution of this chapter is to show how workloads with conflicting transactions can be handled efficiently by a blockchain system without rolling back (aborting) the processed transactions or executing all transactions sequentially. This chapter makes the following contributions:

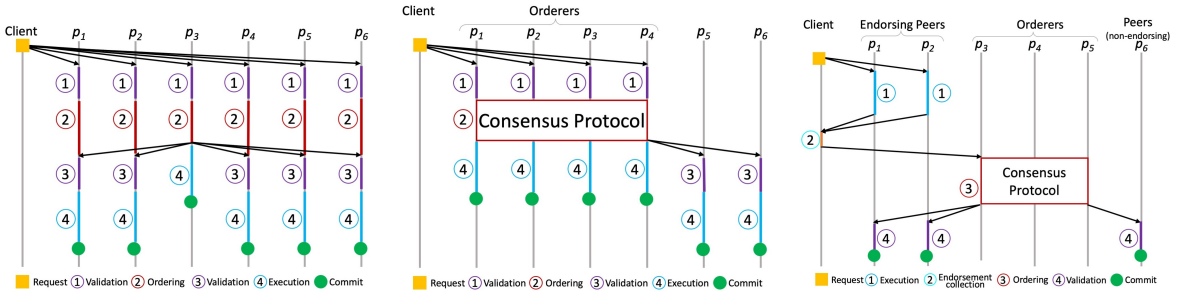
- *OXII*, a new paradigm for permissioned blockchains to support distributed applications that execute concurrently. OXII uses a dependency graph based concurrency control technique to detect possible conflicts between transactions and to ensure the valid execution of transactions while still allowing non-conflicting transactions to be executed in parallel.
- *ParBlockchain*, a permissioned blockchain system designed specifically in the OXII paradigm. The experiments show that workloads with any degree of contention will benefit from ParBlockchain.

The rest of this chapter is organized as follows. Section 5.2 briefly describes current blockchain paradigms and their limitations. The OXII paradigm is introduced in Section 5.3. Section 5.4 presents ParBlockchain, a permissioned blockchain system designed specifically in the OXII paradigm. Section 5.5 shows the performance evaluation, and Section 5.6 concludes the chapter.

5.2 Background

Ordering and *execution* are the two main phases of any fault-tolerant protocol. Fault-tolerant protocols mainly follow an *order-execute* paradigm where the network first, orders transactions and then executes them in the same order on all nodes sequentially.

Permissionless blockchains mainly follow the order-execute paradigm where nodes



(a) Order-Execute Paradigm (Permissionless) (b) Order-Execute Paradigm (Permissioned) (c) Execute-Order Paradigm (Permissioned)

Figure 5.1: Existing Paradigms for Blockchains

validate the transactions, put the transactions into blocks, and try to solve some cryptographic puzzle. The lucky peer that solves the puzzle multicasts the block to all nodes. When a node receives a block of transactions, it validates the solution to the puzzle and all transactions in the block. Then, the node executes the transactions within a block sequentially. Such a paradigm requires all nodes to execute every transaction and all transactions to be deterministic.

Figure 5.1(a) shows the transaction flow for a permissionless blockchain. When a peer receives transactions from clients, in step 1, the peer validates the transactions, puts them into a block, and tries to solve the cryptographic puzzle. If the peer is lucky (p_3 in the figure) and solves the puzzle before other peers, it multicasts the block to all the peers. All the nodes then validate the block and its transactions (step 3), execute the transactions sequentially (step 4), and finally, update their respective copies of the ledger. Note that if multiple peers solve the puzzle at the same time, a fork happens in the blockchain. However, once a block is added to either of the fork branches, nodes in the network join the longest chain.

In permissioned blockchains, on the other hand, since the nodes are known and identified, traditional consensus protocols can be used to order the requests [72]. A permissioned blockchain can follow either order-execute or execute-order paradigm. In order-

execute permissioned blockchains, as can be seen in Figure 5.1(b), a set of peers (might be all of them) validate the transactions, agree on a total order for the transactions, put them into blocks and multicast them to all the nodes. Each node then validates the block, executes the transactions using a smart contract, and updates the ledger.

In order-execute permissioned blockchains, similar to order-execute permissionless blockchains, every smart contract runs on every node. The sequential execution of transactions on every node, however, reduces the blockchain performance in terms of throughput and latency.

In contrast to the order-execute paradigm, Hyperledger Fabric [27] presents a new paradigm for permissioned blockchains by switching the order of execution and ordering. The execute-order paradigm was first presented in Eve [73] in the context of Byzantine fault-tolerant SMR. In Eve peers execute transactions concurrently and then verify that they all reach the same output state, using a consensus protocol. In fact, Eve uses an Optimistic Concurrency Control (OCC) [74] by assuming low data contention where conflicts are rare.

Hyperledger Fabric uses a similar strategy; a client sends a request to a subset of peers, called endorsers (the nodes that have access to the smart contract). Each endorser executes the request and sends the result back to the client. When the client receives enough endorsements (specified by some endorsement policy), it assembles a transaction including all the endorsements and sends it to some specified (ordering) peers to establish a *total order* on all transactions. This set of nodes establishes consensus on transactions, creates blocks, and broadcasts them to every node. Finally, each peer validates a transaction within a received block by checking the endorsement policy and read-write conflicts and then updates the ledger. Since a validation phase occurs at the end, the paradigm is called *execute-order-validate*. Figure 5.1(c) presents the flow of transactions in Fabric. Note that in Fabric the consensus protocol is pluggable and the system can use a

crash fault-tolerant protocol, e.g., Paxos [20], a Byzantine fault-tolerant protocol, e.g., PBFT [54], or any other protocol.

While Fabric improves the performance of blockchains by executing the transactions in parallel (instead of sequentially as the order-execute paradigm does), it performs poorly on workloads with high-contention, i.e., many *conflicting transactions* in a block, due to its high abort rate. Two transactions *conflict* if they access the same data and one of them is a write operation. In such a situation, the order of executing the transactions is important, indeed, the later transaction in a block has to wait for the earlier transaction to be executed first. As a result, if two conflicting transactions execute in parallel, the result is invalid. Although Fabric guarantees correctness by checking the conflicts in the validation phase (the last phase) and disregarding the effects of invalid transactions, the performance of the blockchain is significantly reduced by such conflicts.

5.3 The OXII Paradigm

In this section, we introduce OXII, a new order-execute paradigm for permissioned blockchains. OXII is mainly designed to support distributed applications with high-contention workloads.

OXII consists of a set of nodes in an asynchronous distributed network where each node has one of the following roles:

- *Clients* send operations to be executed by the blockchain.
- *Orderers* agree on a total order of all transactions.
- Executors validate and execute transactions.

The set of nodes in OXII is denoted by N where O of them are orderers, and E of them are executors.

OXII supports distributed applications running concurrently on the blockchain. For each application a program code including the logic of that application (*smart contract*) is installed on a (non-empty) subset of executor peers called the *agents* of the application. We use $\mathcal{A} = \{A_1, ..., A_n\}$ to denote the set of applications (ids) and $\Sigma(A_i)$ to specify the non-empty set of agents of each application A_i where $\Sigma : \mathcal{A} \mapsto 2^E - \emptyset$. Every peer in the blockchain knows the agents of each application and the set of orderers.

Each pair of peers is connected with point-to-point bi-directional communication channels. Network links are pairwise authenticated, which guarantees that a Byzantine node cannot forge a message from a correct node, i.e., if node i receives a message m in the incoming link from node j, then node j must have sent message m to i beforehand.

5.3.1 Orderers

Checking accesses, ordering the requests, constructing blocks, generating dependency graphs, and multicasting the blocks are the main services of orderers in the OXII paradigm.

Since multiple applications run on the blockchain and each application might have its own set of clients, orderers act as trusted entities to restrict the processing of requests that are sent by unauthorized clients. If a client is not authorized to perform an operation on the requested application, orderers simply discard that request. Orderers also check the signature of the requests to ensure their validity.

Orderers use an asynchronous fault-tolerant protocol to establish consensus. OXII, similar to Fabric [27], uses a pluggable consensus protocol for ordering, thus resulting in a modular paradigm. Depending on the characteristics of the network and peers OXII might employ a Byzantine, a crash, or a hybrid fault-tolerant protocol. The number of orderers is also determined by the utilized protocol and the maximum number of

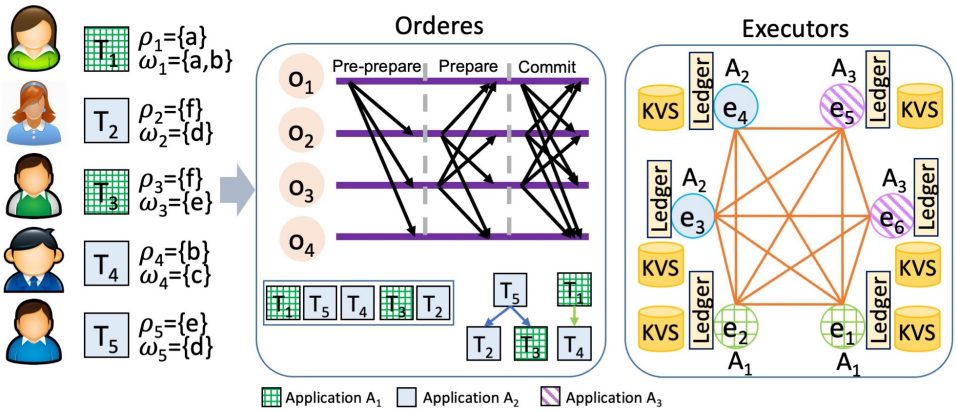


Figure 5.2: The Components of OXII Paradigm

simultaneous failures in the network. Furthermore, orderers do not have access to any smart contract or the application state, nor do they participate in the execution of transactions. This makes orderers independent of the other peers and adaptable to a changing environment.

Orderers batch multiple transactions into blocks. Batching transactions into blocks improves the performance of the blockchain by making data transfers more efficient especially in a geo-distributed setting. It also amortizes the cost of cryptography. The batching process is *deterministic* and therefore produces the same blocks at all orderers.

Figure 5.2 shows the components of the OXII paradigm. As can be seen, clients send requests (transactions) to be executed by different applications. Here, transactions T_1 and T_3 are for some application A_1 and T_2, T_4, and T_5 are for another application A_2. The orderers, i.e., o_1, o_2, o_3, and o_4, then order the transactions and put them into a block. In the figure, orderers use PBFT [54] to order the requests. The resulting block contains five transactions which are ordered as $[T_1, T_5, T_4, T_3, T_2]$.

Next, orderers generate a "dependency graph" for the transactions within a block. In order to generate dependency graphs a priori knowledge of transactions' read- and write-set is needed. Each transaction consists of a sequence of reads and writes, each

accessing a single record. Here we assume that the read-set and write-set are pre-declared or can be obtained from the transactions via a static analysis, e.g., all records involved in a transaction are accessed by their primary keys. Note that even if that assumption does not hold, the system can employ other techniques like speculative execution [75] to obtain the read-set and write-set of each transaction.

Given a transaction T, $\omega(T)$ and $\rho(T)$ are used to represent the set of records written and read, respectively. Each transaction T is also associated with a timestamp $ts(T)$ where for each two transactions T_i and T_j within a block such that T_i appears before T_j, $ts(T_i) < ts(T_j)$.

We define "ordering dependencies" to show possible conflicts between two transactions from the same or different applications. Two transactions conflict if they access the same data and one of them is a write operation.

Definition: Given two transactions T_i and T_j. An *ordering dependency* $T_i \succ T_j$ exists if and only if $ts(j) > ts(i)$ and one of the following hold:

- $\rho(T_i) \cap \omega(T_j) \neq \varnothing$
- $\omega(T_i) \cap \rho(T_j) \neq \varnothing$
- $\omega(T_i) \cap \omega(T_j) \neq \varnothing$

Definition: Given a block of transactions, the *dependency graph* of the block is a directed graph $G = (\mathcal{T}, \mathcal{E})$ where $\mathcal{T}$ is the set of transactions within the block and $\mathcal{E} = \{(T_i, T_j) \mid T_i \succ T_j\}$

We use the example in Figure 5.2 to illustrate the dependency graph construction process. As can be seen the block consists of five transactions which are ordered as $[T_1, T_5, T_4, T_3, T_2]$, i.e., $ts(T_1) < ts(T_5) < ts(T_4) < ts(T_3) < ts(T_2)$. Since T_4 reads data item b which is written by an earlier transaction T_1 there is an ordering dependency $T_1 \succ T_4$, thus (T_1, T_4) is an edge of the dependency graph. Similarly, T_2 writes data item

d which is also written by T_5 ($T_5 \succ T_2$) and T_3 writes data item e which is read by T_5 ($T_5 \succ T_3$). As a result, edges (T_5, T_2) and (T_5, T_3) are also in the graph.

The constructed graph can be used by the executors to manage the execution of transactions. In particular, transactions that are not connected to each other in the dependency graph, e.g., T_1 and T_2, can be processed concurrently by independent execution threads.

The dependency graph generator is an independent module in the OXII paradigm. Therefore, it can also be adapted to a multi-version database system [76]. In a multi-version database, each write creates a new version of a data item, and reads are directed to the correct version based on the position of the corresponding transaction in the block (log). Since writes do not overwrite each other, the system has more flexibility to manage the order of reads and writes. As a result, for any two transactions T_i and T_j within a block where T_i appears before T_j, T_i and T_j can concurrently write the same data item or T_i reads and T_j writes the same data item. However, if T_i wants to write and T_j wants to read the same data item, they cannot be executed in parallel.

It should be noted that in some dependency graph construction approaches, e.g., DGCC [77], transactions are broken down into transaction components, which allows the system to parallelize the execution at the level of operations. The dependency graph generator module in OXII can also be designed in a similar manner.

A dependency graph exposes conflicts between transactions to give a partial order of transactions. Hence, as long as the transactions are executed in an order consistent with the dependency graph, the results are valid. Indeed, using dependency graphs results in higher concurrency by enabling the non-conflicting transactions within a block to be executed in parallel. Such parallelism improves the performance of OXII paradigm in comparison to the traditional order-execute paradigm where transactions are executed sequentially.

When the dependency graph is generated, orderers multicast a message including the block and its dependency graph to all executors. Depending on the employed consensus protocol, either the leader or all the orderers multicast the message.

5.3.2 Executors

Executing and validating transactions, updating the ledger and the blockchain state, and multicasting the blockchain state after executing transactions are the main services of executor peers. Executors in OXII correspond to the endorsers in Hyperledger [27]. Each executor peer maintains three main components: (1) The blockchain ledger, (2) The blockchain state, and (3) Some smart contracts.

When a block of transactions is executed and validated, each executor peer appends the block to its copy of the blockchain ledger. Each executor node is an agent for one or more applications where for each application the smart contract of that application, i.e., a program code that implements the application logic, is installed on the node. When an executor receives a block from the orderers, it checks the application of the transactions within the block. If the executor is an agent for any of the transactions, it executes the transactions on the corresponding smart contract following the dependency graph. In fact, the executor confirms the order of dependent transactions and executes independent transactions in parallel. Finally, it multicasts the execution results (updated blockchain state) to all other peers.

For each transaction within a block where the executor is not an agent of the transaction, the executor waits for matching updates from a specified number of executors, who are the agents of the transaction, before committing the update. This is needed to prevent a malicious executor to commit an invalid result and also tolerate the non-deterministic execution of transactions. The required number of matching results from executors is

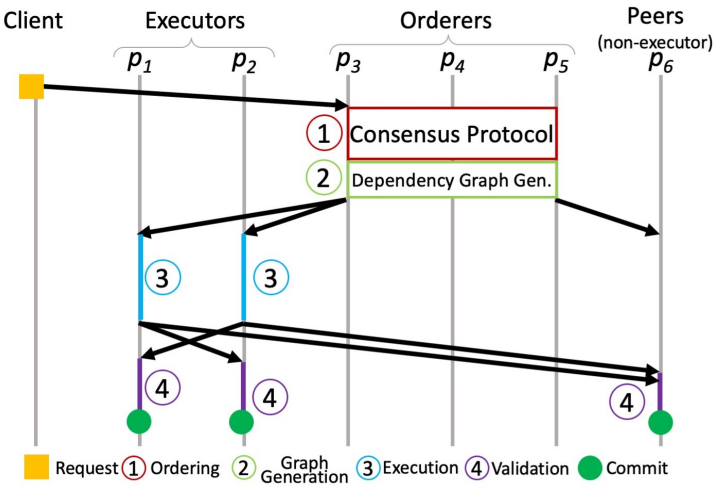


Figure 5.3: The Flow of Transactions in ParBlockchain

decided by the system and known to all executors (similar to endorsement policies in Hyperledger). We use $\tau(A)$ to denote the required number of matching updates for the transactions of application A.

In Figure 5.2, executor nodes e_1 and e_2 are the agents of application A_1 (with transactions T_1 and T_3) and executor nodes e_3 and e_4 are the agents of application A_2 (with transactions T_2, T_4 and T_5).

5.4 ParBlockchain

In this section, we present ParBlockchain, a permissioned blockchain designed specifically in the OXII paradigm. We first give a summary of ParBlockchain and then explain the ordering and execution phases in detail.

5.4.1 ParBlockchain Overview

ParBlockchain is a permissioned blockchain designed in the OXII paradigm to execute distributed applications.

The normal case operation for ParBlockchain to execute transactions proceeds as follows. Clients send requests to the orderers and the orderers run a consensus algorithm among themselves to reach agreement on the order of transactions. Orderers then construct a block of transactions and generate a dependency graph for the transactions within the block.

Once the dependency graph is generated, the block along with the graph is multicast to all the executor nodes. The executors which are the agents of the applications of transactions within the block, execute the corresponding transactions and multicast the results, i.e., updated records in the datastore, to every executor node. Each executor node in the network waits for the required number of matching results from the executors before updating the ledger and blockchain state (datastore). The required number of matching results for each application, which is needed to deal with malicious executors and the non-deterministic execution of transactions, is determined by the system and might be different for different applications.

The flow of transactions in ParBlockchain can be seen in Figure 5.3 where p_3, p_4, and p_5 are the orderer nodes, and p_1, p_2, and p_6 are the executor nodes from which p_1 and p_2 are the agents for the requests. Upon receiving requests from clients, orderers order the requests, put them into a block, generate the dependency graph for the block, and multicast the block along with the graph to all the executor nodes. The agents of the corresponding application (p_1 and p_2) execute the transactions and multicast the updated state of the blockchain to the other executor nodes. Upon receiving the required number of matching messages for each transaction, each executor commits (or aborts) the transaction by updating the blockchain state. The block is also appended to the ledger.

5.4.2 Ordering Phase

The goal of the ordering phase is to establish a total order on all the submitted transactions. A client c requests an operation op for some application A by sending a message $\left\langle \text{REQUEST}, op, A, ts_c, c \right\rangle_{\sigma_c}$ to the orderer p it believes to be the *primary* (an orderer node that initiates the consensus algorithm). Here, ts_c is the client's timestamp and the entire message is signed with signature σ_c. We use timestamps of clients to totally order the requests of each client and to ensure exactly-once semantics for the execution of client requests.

Upon receiving a client request, the primary orderer p checks the signature to ensure it is valid, makes sure the client is allowed to send requests for application A (access control), and then initiates a consensus algorithm by multicasting the request to other orderers. Depending on the utilized consensus protocol, several rounds of communication occurs between orderers to establish a total order on transactions.

Once the orderers agree on the order of a transaction, they put the transaction in a block. Batching multiple transactions into blocks improves the throughput of the broadcast protocol. Blocks have a pre-defined maximal size, maximal number of transactions, and maximal time the block production takes since the first transaction of a new block was received. When any of these three conditions is satisfied, a block is full. Since transactions are received in order, the first two conditions are deterministic. In the third case, to ensure that the produced blocks by all orderers are the same, the primary sends a *cut-block* message in the consensus step of the last request.

When a block is produced, orderers generate a dependency graph for the block as explained in Section 5.3.1. Generating dependency graphs requires a priori knowledge of transactions' read- and write-set. Here, we assume that the requested operations include the read- and write-set.

Algorithm 6 Execution of Transactions on an executor e

Input: A block B and its dependency graph $G(B)$

1: Initiate Set W_e to be empty transaction x in B e is an agent of x's application
2: Add x to W_e W_e in not empty transaction(node) x in W_e all $Pre(x)$ are in $C_e \cup X_e$
3: trigger Execute(x)

When the graph is constructed, each orderer node o multicasts a message $\Big\langle \text{NEWBLOCK}, n, B, G(B), \mathbf{A}, o, h \Big\rangle_{\sigma_o}$ to all executor nodes where n is the sequence number of the block, B is the block consisting of the request messages, $G(B)$ is the dependency graph of B, $\mathbf{A}$ is the set of applications that have transactions in the block, and $h = H(B')$ where $H(.)$ denotes the cryptographic hash function and B' is the block with sequence number $n-1$.

5.4.3 Execution Phase

Each request for an application is executed on the specified set of executors, i.e., agents of that application. Upon receiving a new block message $\Big\langle \text{NEWBLOCK}, n, B, G(B), \mathbf{A}, o, h \Big\rangle_{\sigma_o}$ from some orderer o, executor e checks the signature and the hash to be valid and logs the message. It also checks the set $\mathbf{A}$ to see if the block contains any transaction that needs to be executed by the node, i.e., an application $A_i \in \mathbf{A}$ such that $e \in \Sigma(A_i)$.

When an executor node receives a specified number of matching new block messages, e.g., $f + 1$ messages if the consensus protocol is PBFT, it marks the new block as a valid block and enters the execution phase. The execution phase consists of three procedures that are run concurrently: (1) Executing the transaction following the dependency graph, (2) Multicasting commit messages including the execution results to other executor nodes, and (3) Updating the blockchain state upon receiving commit messages from a sufficient number of executor nodes.

If an executor node is not an agent of any transaction within the block, the node becomes a *passive* node and only the third procedure is run to update the blockchain state. However, if a node is an agent for some transaction's application in the block, it runs

Algorithm 7 Multicasting the Results

1: Initialize set X_e to be empty
2: cut = false
3: Upon obtaining an execution result (x, r)
4: Add pair (x, r) to X_e
5: Remove x from W_e y where (x, y) is an edge in $G(B)$ y's application is different from x's application
6: cut = true
7: break cut = true
8: Multicast $\langle \mathsf{COMMIT}, X_e, e \rangle_{\sigma_e}$ to all executors
9: Clear X_e

all three procedures; executes the corresponding transactions following the dependency graph, multicasts the results, and also updates the blockchain state.

A transaction can be executed only if all of its "predecessors" in the dependency graph are committed. We define functions Pre and Suc to present the set of predecessors and successors of a node in a dependency graph respectively. More formally, Given a dependency graph $G = (\mathcal{T}, \mathcal{E})$, and a node (transaction) x in $\mathcal{T}$, $Pre(x) = \{y \mid (y, x) \in \mathcal{E}\}$ and $Suc(x) = \{y \mid (x, y) \in \mathcal{E}\}$.

The execution procedure on a node e is shows in Algorithm 6. An empty set W_e is initiated to keep all the transactions that will be executed by executor e, i.e., e is an agent for the application of those transactions. Set X_e stores the executed transactions by e and C_e keeps the committed transactions. For each transaction x in W_e, the procedure checks the predecessors of x, if x has no predecessor, or all of its predecessors are executed by e or committed, transaction x is ready to be executed, so an execution thread is triggered.

To multicast the execution results depending on the transactions' applications three different situations could happen. If all the transactions within a block belong to the same application, an agent e executes all of the transactions following the dependency graph and multicast a *commit* message $\left\langle \mathsf{COMMIT}, S, e \right\rangle_{\sigma_e}$ to all other executor nodes. Here, S presents the state of the blockchain and consists of a set of pairs (x, r) where x is a transaction (id) and r is the set of updated records resulting from the execution of x on

the datastore. Note that if a transaction x is not valid, the executor puts (x,"abort") in S.

If the transactions within a block are for different applications but the transactions of each application access a disjoint set of records, the agents still can execute the corresponding transactions independently and multicast a single commit message with all the results to other executor nodes. In this case, the dependency graph is disconnected and can be decomposed to different components where the transactions of each component are for the same application and there is no edge that connects any two components.

However, if there are some dependencies between the transactions of two applications, the agents of those two applications cannot execute the transactions independently. In fact, the agents of one application have to wait for the agents of other applications to execute all their transactions and send the commit message which might result in a deadlock situation.

Figure 5.4 shows three dependency graphs for a block of seven transactions T_1 to T_7. In Figure 5.4(a), all the seven transactions belong to the same application, A_1. Therefore, the agents of application A_1 can execute the transactions following the dependency graph and multicast the results of all transactions together when they all are executed. In Figure 5.4(b) although the transactions belong to different applications (T_2, T_3, T_5, and T_7 are for application A_1 and T_1, T_4 and T_6 are for application A_2), there is no dependency between the transactions of application A_1 and the transactions of application A_2. As a result, the agents can still execute independently and multicast the results once the execution of their transactions is completed. However, in Figure 5.4(c) since there are some dependencies between the transactions of the two applications, the agents cannot execute their transactions independently. For example, to execute transaction T_2, the agents of application A_2 need the execution results of transaction T_5 from the agents of A_1. Similarly, transaction T_4 cannot be executed before committing the execution results

Algorithm 8 Updating the Blockchain State

1: Initialize set $R_e(x)$ to be empty
2: Initialize set C_e to be empty
3: Upon Receiving a valid $\langle \textsf{COMMIT}, S, n \rangle_{\sigma_n}$ message valid $(x, r) \in S$
4: Add (r, n) to $R_e(x)$ Matching records in $R_e(x) \geq \tau(A)$
5: Update the blockchain state
6: Add x to C_e

of transaction T_6.

To prevent a deadlock situation, one possibility is that agents send a commit message as soon as the execution of each transaction is completed. While this approach solves the blocking problem, the number of exchanged commit messages will be large. Indeed, if a block includes n transactions and each application has on average m agents, there will be total $n * m$ exchanged commit messages for the block.

A more efficient way is to send commit messages when the execution results are needed by some other agents. Basically, an agent keeps executing the transactions and collecting the results until the results of an executed transaction is needed by some other transactions which belong to other applications. At that time, the agent generates a commit message including the results of all the executed transactions and multicasts it to all executor nodes. Upon receiving a commit message from an executor, the node validates the signature and logs the message. Once the node receives the specified number of matching results for a transaction, the results are reflected in the datastore and the transaction is marked as committed.

Algorithm 7 presents the multicasting procedure on a node e. An empty set X_e is initiated to store the results of the executed transactions. When the execution of a transaction x is completed, the execution result (x, r) is added to X_e and transaction x is removed from the waiting transactions W_e.

Then, the procedure checks all the successor nodes of x in the dependency graph. If any of the successor nodes of x belongs to an application different from the application

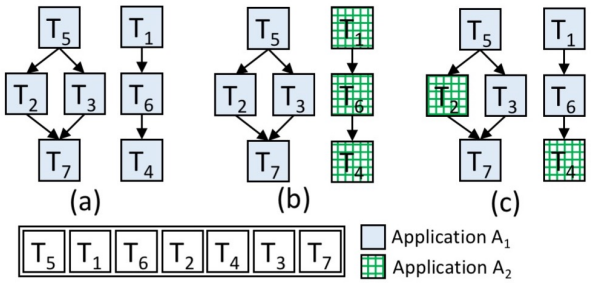


Figure 5.4: Three dependency graphs

of x, the execution result of transaction x might be needed by other agents, thus a multicasting has to occur. To do so, node e removes all the stored results from X_e and puts them in a commit message and multicast the commit message to all other executor nodes.

For example, in Figure 5.4(c), upon executing the transaction T_5, since T_5 has a successor node T_2 that belongs to another application, the executor node multicasts a commit message including the execution results of T_5 to all other executor nodes. Note that if T_1 is already executed, the executor node puts the execution results of T_1 in the commit message as well. Similarly, when the execution of T_6 is completed, the executor node multicasts a commit message including the execution results of T_6 and any other executed but not yet multicast transactions.

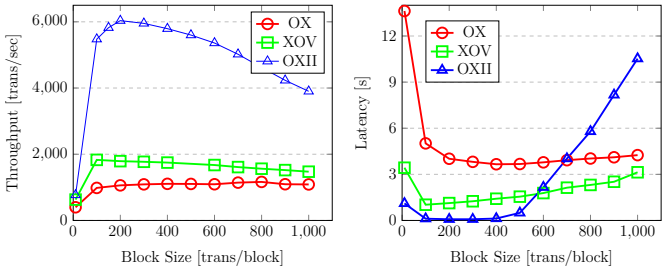
Finally, the updating procedure receives commit messages from other executor nodes and updates the blockchain state. The updating procedure on a node e is presented in Algorithm 8. the procedure first initializes an empty set $R_e(x)$ for each transaction x in the block. It also initializes an empty set C_e to collect committed transactions. When node e receives a commit message $\left\langle \text{COMMIT}, S, n \right\rangle_{\sigma_n}$ from some executor n, it checks the signature to be valid and then checks the set S. Recall that S consists of pairs of transactions and their execution results. For each pair (x, r), it first checks whether node n is an agent for the application of transaction x and then a pair of (r, n), i.e., execution

results and the executor to $R_e(x)$. Assuming A is the x's application, If the number of the matching tuples in $R_e(x)$ is equal to $\tau(A)$, i.e., the specified number of messages for the transaction's application, the execution results are valid and can be committed. As a result, the procedure updates the blockchain state (datastore) and adds the transaction x to the committed transactions C_e.

5.5 Experimental Evaluations

In this section, we conduct several experiments to evaluate different paradigms for permissioned blockchains. We discussed the two existing paradigms for permissioned blockchains in Section 5.2: sequential order-execute (OX) where requests are ordered and then executed sequentially on every node, and execute-order-validate (XOV) introduced by Hyperledger Fabric [27] where requests are executed by the agents of each application, ordered by the ordering service, and validated by every peer. We implemented two permissioned blockchain systems specifically designed in the OX and XOV paradigms as well as ParBlockchain that is designed in the OXII paradigm. It should be noted that our implementation of XOV is different from the Hyperledger fabric system. Hyperledger is a distributed operating system and includes many components which are not the focus of our evaluations. In fact, the purpose of our experiments is to compare the architectural aspect of the blockchain systems, thus, all three systems are implemented using the same programming language (*Java*). To have a fair comparison, we also used similar libraries and optimization techniques for all three systems as far as possible.

We implemented a simple accounting application where each client has several accounts. Each account can be seen as a pair of $(amount, PK)$ where PK is the public key of the owner of the account. Clients can send requests to transfer assets from one or more of their accounts to other accounts. For example, a simple transaction T initiated



(a) Throughput (b) Latency

Figure 5.5: Increasing the block size

by client c might "transfer x units from account 1001 to account 1002". The transaction is valid if c is the owner of account 1001 and the account balance is at least x. Here the read-set of transaction T is $\rho(T) = \{1001\}$ and its write-set is $\omega(T) = \{1001, 1002\}$. A transaction might read and write several records.

The experiments were conducted on the Amazon EC2 platform. Each VM is Compute Optimized c4.2xlarge instance with 8 vCPUs and 15GM RAM, Intel Xeon E5-2666 v3 processor clocked at 3.50 GHz. For orderers, similar to Hyperledger [27], we use a typical Kafka orderer setup with 3 ZooKeeper nodes, 4 Kafka brokers and 3 orderers, all on distinct VMs. Unless explicitly mentioned differently, there are three applications in total each with a separate executor (endorser) node.

When reporting throughput measurements, we use an increasing number of clients running on a single VM, until the end-to-end throughput is saturated, and state the throughput just below saturation. Throughput numbers are reported as the average measured during the steady state of an experiment.

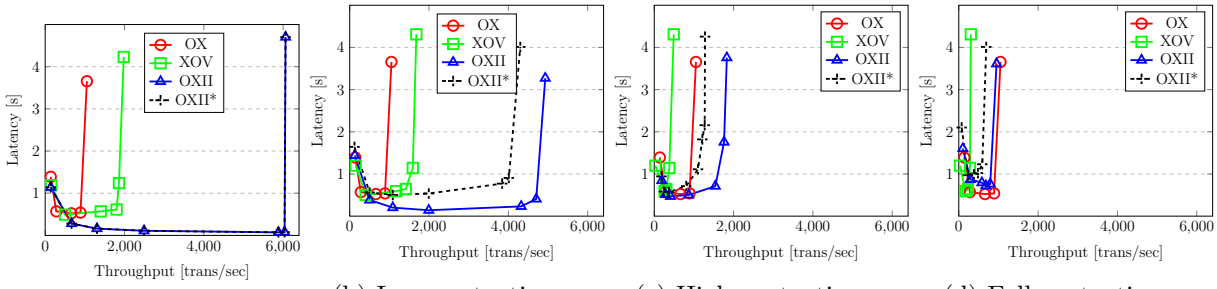
5.5.1 Choosing the Block Size

An important parameter that impacts both throughput and latency is the block size. To evaluate the impact of the block size on performance, in this set of experiments, assuming that the transactions have the same size, we increase the number of transactions

in each block from 10 to 1000 in a no-contention workload. For each block size, the peak throughput and the corresponding average end-to-end latency is measured. As can be seen in Figure 5.5, by increasing the number of transactions per block till ∼200, the throughput of OXII increases, however, any further increasing reduces the throughput due to the large number of required computations for the dependency graph generation. Similarly, by increasing the number of transactions per block till ∼200, the delay decreases. Afterward, adding more transactions to the dependency graph becomes more time consuming than multicasting the block. As a result, OXII is able to process more than 6000 transactions in $78ms$ with 200 transactions per block. In the OX paradigm, since nodes execute transactions sequentially, the block creation time is negligible in comparison to the execution time, thus other than in the first experiment, increasing the number of transactions per block does not significantly affect the throughput and latency. In the XOV paradigm, since executors (endorsers) of the three applications can execute the transactions in parallel, the performance is better than OX (twice as much as OX in its peak throughput). However, its performance is still much less than OXII, i.e., the peak throughput of XOV is 30% of the peak throughput of OXII as OXII can execute many (and not only three) non-conflicting transactions in parallel. As can be seen, the peak throughput of XOV is obtained in ∼100 transactions per block.

5.5.2 Performance in Workloads with Contention

In the next set of experiments, we measure the performance of all three paradigms for workloads with different degrees of contention. we consider no-contention, low-contention (20% conflict), high-contention (80% conflict), and full-contention workloads where the results are shown in Figure 5.6(a)-(d) respectively. Note that the dependency graph of each block in the first workload has no edge whereas the dependency graph of each block



(a) No-contention (0%) (b) Low-contention (20%) (c) High-contention (80%) (d) Full-contention (100%)

Figure 5.6: Increasing the degree of contention

in the last workload is a chain. In OX and OXII there are 200 transactions per block and for XOV, we keep changing the block size to find its peak throughput. Contentions could happen between the transactions of the same application or the transactions of different applications (if they access shared data). In OX, since nodes execute transactions sequentially, there is no difference between these two types of contention. In XOV also, since the execution is the first phase, there is no much difference between contention within an application or across applications and they both result in transaction abort. In OXII, however, as discussed in Section 5.4.3, for contention across applications, the agents of different applications communicate to each other during the execution of a block of transactions, thus the performance is affected. As a result, in this set of experiments, for each workload, we report the performance of OX, XOV, OXII with conflicting transactions within an application, and OXII with conflicting transactions across applications (the dashed line).

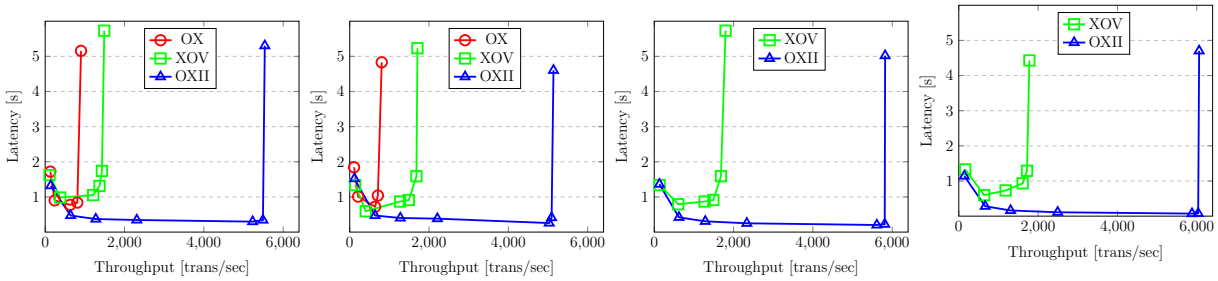
As mentioned earlier, in the OX paradigm, transactions are executed sequentially. As a result, the performance of OX remains unchanged in different workloads. XOV can execute 3 (number of applications) transactions in parallel and since the workload has no-contention, the execution results are valid. OXII, on the other hand, significantly benefits from no-contention workloads by executing the transactions in parallel. As shown in Figure 5.6(a), OXII executes more than 6000 transactions with latency less than 80 ms

whereas the peak throughput of OX is 900 transactions with more than 500 ms latency. XOV can also execute around 1800 transactions in 600 ms (70% less throughput and 7.5 times latency in comparison to OXII). Since the workload has no conflicting transaction, there is no contention across applications.

By increasing the degree of contention (Figure 5.6(b) and Figure 5.6(c)), the throughput of XOV decreases dramatically, e.g., the peak throughput of XOV in a high-contention workload is around 25% of its peak throughput in a no-contention workload. This decrease is expected because XOV validates and aborts the conflicting transactions at the very end (last phase). The throughput of OXII is also affected by increasing the degree of contention, however, it still shows better performance than both OX and XOV, i.e., OXII is still able to process 1600 transactions in sub-second latency whereas OX and XOV process 900 and 350 transactions respectively. Processing the workloads with contention across the applications decreases the performance of OXII due to the increasing rounds of communication between executors of different applications.

In a full-contention workload, as can be seen in Figure 5.6(d), OXII similar to OX, executes the transactions sequentially, but, because of the dependency graph generation overhead, its performance is a bit worse than OX. The performance of the XOV paradigm, on the other hand, is highly reduced. Since all the transactions within a block conflict, it can only commit one transaction per block (we reduced the block size of XOV to record its peak throughput).

In a full-contention workload with contention across applications (dashed line in Figure 5.6(d)), OXII has high latency and low throughput. Such a workload can be seen as a chain of translations where consecutive transactions belong to different applications. As a result, to execute each transaction, a message exchange between a pair of executors is needed.



(a) Apart client (b) Apart orderers (C) Apart executors (d) Apart non-executors

Figure 5.7: Scalability over multiple data centers

5.5.3 Scalability over Multiple Data Centers

In the last set of experiments, we measure the scalability of the blockchain systems over multiple data centers. To this end, each time we move one group of nodes, i.e., clients, orderers, executors, or non-executors, to AWS Asia Pacific (Tokyo) Region data center, leaving the other nodes in the AWS US West Region data center (the RTT between these two data centers is 113 ms). We consider a no-contention workload. The results can be seen in Figure 5.7.

Moving the clients has the most impact on the XOV paradigm because in XOV clients participate in the first two phases. Indeed, they send the requests to the executors (endorsers), receive endorsements, and then send the endorsements to the orderer nodes. Whereas in OX and OXII, clients send the requests and do not participate in other phases of the protocol. As a result, as can be seen in Figure 5.7(a), the delay of XOV becomes much larger.

Orderers are the core part of all three blockchains; they receive transactions from clients, agree on the order of the transactions, put the transactions into blocks, and send the blocks to every node. As a result, moving them to a far data center, as shown in Figure 5.7(b), results in a considerable delay. Note that in OX, a subset of nodes are considered as orderers.

In the last two experiments (Figure 5.7(c)-(d)), we move executor (endorser) and non-exe nodes to the far data center. Since there is no such a separation between nodes in the OX paradigm, we do not perform these two experiments. Moving executor nodes adds latency to the two phases of communication in XOV (clients to executors and executors to clients) and one phase of communication in OXII (orderers to executors). Note that when the executors execute the messages and receive enough number of matching results from other executors, the transaction is counted as committed. In addition, no communication between executors is needed since we consider a no-contention workload. Finally, moving non-executor nodes has no impact on the performance of OXII, because those nodes are only informed about the blockchain state. But in XOV, non-executors validate the blocks.

5.6 Summary

In this chapter, we proposed OXII, an order-execute paradigm for permissioned blockchain to support distributed applications that execute concurrently. OXII is able to handle the workload with conflicting transactions without rolling back the processed transactions or executing transactions sequentially. Conflicts between the transactions of a single application as well as the transactions of different applications are addressed in OXII. We also presented ParBlockchain, a high performance permissioned blockchain system designed specifically in the OXII paradigm. Our experimental evaluations show that in workloads with conflicting transactions, ParBlockchain shows a better performance in comparison to both order-execute and execute-order permissioned blockchain systems.

Chapter 6

SharPer: On Scalability of Permissioned Blockchains

6.1 Introduction

Scalability is one of the main obstacles to business adoption of blockchain systems. Scalability is the ability of a blockchain system to process an increasing number of transactions by adding resources to the system. The scalability of blockchain systems has been addressed in several studies using different on-chain, e.g., increasing the block size, and off-chain, e.g., Lightning Networks [78] [34], techniques. Increasing the block size, however, increases both the propagation time and the verification time of the block which makes operating full nodes more expensive, and this in turn could cause less decentralization in the network [79]. Off-chain solutions also suffer from security issues [80] especially denial-of-service attacks [1].

Partitioning the data into multiple shards that are maintained by different subsets of non-malicious nodes is a proven approach to improve the scalability of distributed

[1] https://www.trustnodes.com/2018/03/21/lightning-network-ddos-sends-20-nodes

databases [35]. In such an approach, the performance of the database scales linearly with the number of nodes. Recently, sharding has been utilized by several approaches in the presence of Byzantine nodes in both permissionless and permissioned blockchain systems. Sharded permissionless blockchains, e.g., Elastico [81], OmniLedger [82], and Rapidchain [83], ensure *probabilistic* correctness by randomly assigning nodes to *committees* (partitions) resulting in a uniform distribution of faulty nodes in committees. OmniLedger and Rapidchain also support cross-shard transactions using Byzantine consensus protocols.

Sharding techniques have also been used by different permissioned blockchains, e.g., Fabric [27], Cosmos [84], RSCoin [85], and AHL [86]. In Fabric, channels are introduced to shard the system. A channel is a partitioned state of the full system that is autonomously managed by a (logically) separate set of nodes, but is still aware of the bigger system it belongs to [28]. By using channels, Fabric is able to process intra-shard transactions efficiently. However, processing any cross-shard transaction needs either the existence of a trusted channel among the participants or an atomic commit protocol [28]. Cosmos [84] introduces Inter-Blockchain Communication (IBC) to initiate cross-blockchain operations. Interacting chains in IBC, however, must be aware of the state of each other which requires establishing a bidirectional trusted channel between two blockchains. In AHL [86], Dang et al. employ a trusted hardware (the technique that is presented in [87] [88] [89]) to decrease the number of required nodes within each committee. AHL randomly assigns nodes to the committees and ensures safety with a high probability if each committee consists of 80 nodes (instead of ∼600 nodes in OmniLedger). Nevertheless, running Byzantine fault-tolerant protocols among 80 nodes results in high latency. In addition, in AHL [86], consensus on the order of cross-shard transactions not only requires an extra set of nodes (called a *reference committee*), but also results in a large number of communication phases. Furthermore, since a single reference committee pro-

cesses cross-shard transactions, AHL is not able to process cross-shard transactions with non-overlapping committees in parallel.

In many systems, especially permissioned blockchains, the number of available Byzantine nodes is much larger than $3f + 1$. In such systems, using all the nodes to establish consensus degrades performance since more messages are being exchanged without providing improved resiliency, e.g., in PBFT [21], the number of message exchanges is quadratic in terms of the number of nodes. Different techniques have been presented to address this issue. In the active/passive replication technique, the protocol relies only on $3f+1$ active nodes to establish consensus whereas FaB [38] uses $5f+1$ replicas to establish consensus in two phases instead of three as in PBFT. Similar techniques have been presented for crash failures to use $3f+1$ replicas instead of $2f+1$ [90] [91]. However, such techniques do not utilize the extra nodes efficiently when a very high percentage of nodes are non-faulty.

In this chapter, we present a model including a blockchain ledger for sharded permissioned blockchains, introduce consensus protocols to order both intra- and cross-shard transactions on either crash-only or Byzantine nodes and design a sharded permissioned blockchain system, *SharPer*, to improve scalability. SharPer partitions the nodes into *clusters* of either $2f + 1$ crash-only or $3f + 1$ Byzantine nodes to guarantee safety and can be used specifically in networks with very high percentage of non-faulty nodes.

SharPer assigns data shards to the clusters where each cluster processes the transactions that access its corresponding shard. If a transaction accesses only a single shard, i.e., an *intra-shard transaction*, the corresponding cluster orders and executes the transaction locally. As a result, intra-shard transactions of different clusters are independent of each other, and can be processed in parallel. However, for a *cross-shard transaction*, agreement among *all* and *only* involved clusters is required. Nevertheless, if two cross-shard transactions have no overlapping clusters, they still can be processed in parallel.

Since the ordering of different transactions might be performed in parallel and the system includes cross-shard transactions, the blockchain ledger of SharPer is represented as a *directed acyclic graph* including all intra- and cross-shard transactions. Nonetheless, for the sake of performance, the blockchain ledger is *not maintained* by any node and nodes of each cluster maintain their own *view* of the ledger including its intra-shard transactions and the cross-shard transactions that the cluster is involved in. The main contributions of this chapter are:

- SharPer, a permissioned blockchain system that supports the concurrent processing of transactions by clustering the nodes into clusters and sharding the data and the blockchain ledger. SharPer supports both intra-shard and cross-shard transactions.
- Two *flattened* consensus protocols for ordering cross-shard transactions among all and only the involved clusters in networks consisting of either crash-only or Byzantine nodes. The protocols order cross-shard transactions with non-overlapping clusters in parallel.

The rest of this chapter is organized as follows. The SharPer model is introduced in Section 6.2. Sections 6.3 and 6.4 show how consensus works in SharPer. Section 6.5 presents a performance evaluation of SharPer, and Section 6.6 concludes the chapter.

6.2 The SharPer Model

SharPer is a permissioned blockchain system designed specifically to achieve high scalability in networks with a very large percentage of non-faulty nodes. SharPer partitions the nodes into clusters and assigns a data shard to each cluster. Each node, in addition to a data shard, stores a view of the blockchain ledger. In this section, we show how clusters and shards are formed and then, introduce the blockchain ledger.

Figure 6.1: The infrastructure of SharPer with 16 Byzantine nodes where $f = 1$

6.2.1 Cluster and Shard Formation

In existing sharded permissionless blockchain systems, e.g., OmniLedger [82], nodes are assigned to clusters (committees) randomly. In such systems, to ensure safety, i.e., each cluster includes at most one-third faulty nodes, with a high probability $(1-2^{-20})$, each cluster consists of hundreds of nodes. In the permissioned blockchain system AHL [86], safety is ensured with the same probability with clusters of only 80 nodes using trusted hardware, however, as discussed earlier, running fault-tolerant protocols among 80 nodes results in high latency. In SharPer, on the other hand, the number of nodes, N, is assumed to be much larger than $3f+1$ (or $2f+1$ if nodes are crash-only), thus, nodes are partitioned into clusters each large enough to tolerate f failures. As a result and in contrast to AHL, SharPer provides a *deterministic* safety guarantee (not a probabilistic one), hence there is no need to reconfigure clusters or assign nodes to clusters in a random manner. Note that both the probabilistic approach and trusted hardware technique can also be utilized in SharPer resulting in enhanced performance.

In SharPer and in the presence of crash-only nodes, each cluster includes $2f+1$ nodes (the last cluster might include more nodes) and similarly, in the Byzantine failure model, each cluster includes $3f+1$ nodes. Nodes are assigned to the clusters mainly based on their geographical distance, i.e., nodes that are close to each other are assigned to the same cluster to reduce the latency of communication within a cluster. We denote the set of clusters by $P = \{p_1, p_2, ...\}$. If nodes are crash-only, the number of clusters, $|P|$, is

equal to $\frac{N}{2f+1}$. Similarly, in the presence of Byzantine nodes, the number of clusters, $|P|$, is $\frac{N}{3f+1}$. The number of clusters in SharPer, indeed, depends on the number of nodes, number of failures, and the failure model of nodes. As a result, the lower the percentage of faulty nodes, the more the number of clusters. Since there are $|P|$ clusters, the data is also sharded into $|P|$ shards, thus each cluster maintains a single data shard that is replicated on the nodes of the cluster. We denote shards by d_1, ..., $d_{|P|}$ where each shard d_i is replicated over the nodes of cluster p_i.

Figure 6.1 illustrates the SharPer infrastructure for a blockchain system consisting of 16 nodes following Byzantine failure model where $f = 1$. As can be seen, the system consists of four clusters ($|P|=\frac{16}{4}$) of size four ($3f+1$). The data is sharded into four shards where each shard d_i is replicated on the nodes of cluster p_i. Nodes within each cluster, in addition to a data shard, store a view of the blockchain ledger.

An appropriate sharding needs to be *workload-aware*, i.e., have prior knowledge of the data and how the data is accessed by different transactions. Workload-aware data sharding increases the probability of maintaining the records which are accessed by a single transaction in the same shard [92]. Different approaches have been proposed to minimize the number of distributed transactions in a sharded system [93], nevertheless, there might still be a portion of transactions that accesses records from different shards. As a result, SharPer supports two types of transactions: *intra-shard* transactions that access the records within a shard and *cross-shard* transactions that accesses records from different shards.

6.2.2 Blockchain Ledger

In SharPer, each data shard is replicated on all nodes of a cluster. As a result, to ensure data consistency, a total order among transactions (both intra- and cross-

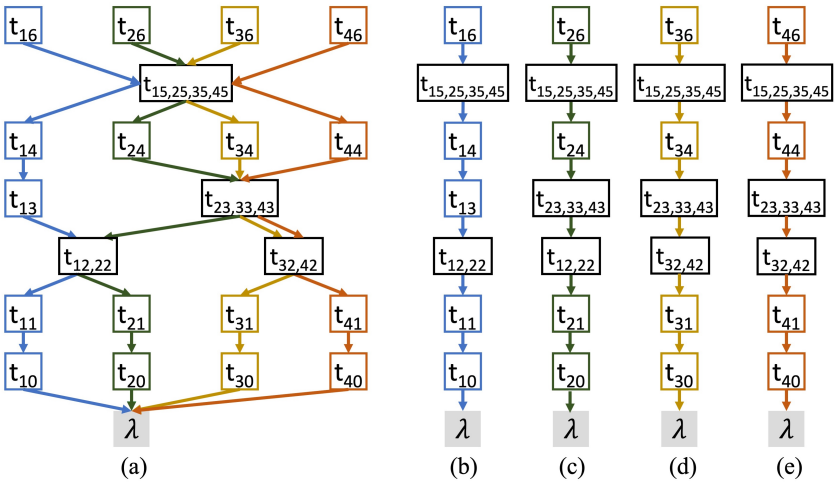


Figure 6.2: (a): A blockchain ledger, (b)-(e): Different views of the blockchain

shard) that access the same data shard is needed. The total order of transactions in the blockchain ledger is captured by *chaining* transaction blocks (we assume each block consists of a single transaction) together, i.e., each block includes a sequence number or the cryptographic hash of the previous transaction block. Since more than one cluster is involved in each cross-shard transaction, similar to CAPER, the ledger is formed as a *directed acyclic graph*. The ledger also includes a unique initialization block, called the *genesis* block.

Fig. 6.2(a) shows a blockchain ledger created in the SharPer model consisting of four clusters p_1, p_2, p_3, and p_4 (data shards d_1, d_2, d_3, and d_4). In this figure, λ is the genesis block of the blockchain. Intra- and cross-shard transactions are also specified. For example, t_{10}, t_{11}, t_{13}, t_{14}, and t_{16} are the intra-shard transactions of cluster p_1. Each cross-shard transaction is labeled with $t_{o_1,..,o_k}$ where k is the number of involved clusters and o_i indicates the order of the transaction among the transactions of the i^{th} involved cluster. This is needed to ensure that cross-shard transactions are ordered correctly with regard to the intra-cluster transactions of all involved clusters. For example, $t_{12,22}$ and $t_{15,25,35,45}$ are two cross-shard transactions where $t_{12,22}$ accesses data shards d_1 and d_2,

whereas $t_{15,25,35,45}$ accesses all four data shards.

As can be seen, there is a total order among the transactions (both intra- and cross-shard) that access a data shard, e.g., t_{10}, t_{11}, $t_{12,22}$, t_{13}, t_{14}, $t_{15,25,35,45}$, and t_{16} are chained together. In addition, intra-shard transactions of different clusters can be added to the blockchain ledger in parallel, e.g., t_{11}, t_{21}, t_{31}, and t_{41} can be processed by different clusters in parallel. Similarly, if two cross-shard transactions access disjoint subsets of shards, they can be added to the ledger in parallel as well, e.g., $t_{12,22}$ and $t_{32,42}$.

In SharPer, similar to CAPER, the entire blockchain ledger is *not maintained* by any cluster and each cluster maintains only its own *view* of the blockchain ledger including the transactions that access the data shard of the cluster. The blockchain ledger is indeed the union of all these physical views.

Fig. 6.2(b)-(e) show the views of the ledger for clusters p_1, p_2, p_3, and p_4 respectively. As can be seen, each cluster p_i maintains only a view of the ledger consisting of the intra-shard transactions of p_i and the cross-shard transactions that access d_i. Those transactions are chained together.

6.3 Consensus with Crash-Only Nodes

In this section, we first show how consensus is established in SharPer for intra-shard and cross-shard transactions in the presence of crash-only nodes. Then, the primary failure handling routine of SharPer is presented and finally the correctness of SharPer is proven.

6.3.1 Intra-shard consensus

SharPer uses multi-Paxos, a variation of Paxos [20], where *the primary* (a pre-elected node that initiates consensus) is relatively stable, to establish consensus on the order

of intra-shard transactions. In SharPer, clients send signed requests to the primary. The primary then assigns a sequence number to the request (to provide a total order among transactions) and multicasts a propose message (called accept in Paxos) including the intra-shard transaction to every node within the cluster. Instead of a sequence number, the primary can also include the cryptographic hash of the previous transaction block, $H(t)$, in the message where $H(.)$ denotes the cryptographic hash function and t is the previous block that is ordered by the cluster. Upon receiving a valid propose message from the primary, each node sends an accept (i.e., accepted) message to the primary. The primary waits for f accept messages from different nodes (plus itself becomes $f+1$), multicasts a *signed* commit message to every node within the cluster, appends the transaction block including the transaction and the commit message (as evidence of the transaction's validity) to the blockchain ledger, executes the transaction, and sends a reply to the client. Upon receiving a commit message from the primary, each node appends the transaction block including the transaction and the received commit message to its blockchain ledger. The client also waits for a valid reply from the primary to accept the result. If the client does not receive replies soon enough, it multicasts the request to all nodes within the cluster. If the request has already been processed, the nodes simply send the execution result back to the client. Otherwise, if the node is not the primary, it relays the request to the primary. If the primary does not multicast the request to the nodes of the cluster, it will eventually be suspected to be faulty by nodes by the nodes. Note that since commit messages include the digest (cryptographic hash) of the corresponding transactions, appending valid signed commit messages to the blockchain ledger in addition to the transactions, provides the same level of *immutability* guarantee as including the cryptographic hash of the previous transaction in the transaction block, i.e., any attempt to alter the block data can easily be detected.

Algorithm 9 Cross-shard Consensus with Crash-Only Nodes

1: **init():**
2: r := *node_id*
3: p_i := the cluster that initiates the consensus
4: $\pi(p_j)$:= the primary node of cluster p_j
5: P := set of involved clusters

6: **upon receiving** valid request m and $(r == \pi(p_i))$
7: **multicast** $\langle \mathsf{PROPOSE}, h_i, d, m \rangle$ to P

8: **upon receiving** valid $\langle \mathsf{PROPOSE}, h_i, d, m \rangle$ from *primary* $\pi(p_i)$
9: if r is not waiting for commit message of request m' where m and m' intersect in some other cluster p_k
10: **send** $\langle \mathsf{ACCEPT}, h_i, h_j, d, r \rangle$ to *primary* $\pi(p_i)$

11: **upon receiving** f+1 valid matching $\langle \mathsf{ACCEPT}, h_i, h_j, d, r \rangle$ from every cluster p_j in P and $(r == \pi(p_i))$
12: **multicast** $\langle \mathsf{COMMIT}, h_i, h_j, ..., h_k, d \rangle_{\sigma_{\pi(p_i)}}$ to P
13: **append** the transaction and commit message to the ledger

14: **upon receiving** $\langle \mathsf{COMMIT}, h_i, h_j, ..., h_k, d \rangle_{\sigma_{\pi(p_i)}}$ from $\pi(p_i)$
15: **append** the transaction and commit message to the ledger

6.3.2 Cross-Shard Consensus

Cross-shard transactions access records from different data shards which are maintained by different clusters. To ensure data consistency, cross-shard transactions have to be appended to the blockchain ledgers of all involved clusters in the same order. As a result, consensus among all involved clusters on the order of cross-shard transactions is needed. In this section, we show how SharPer processes cross-shard transactions in a network consisting of crash-only nodes.

A client sends its request (i.e., a cross-shard transaction) to the (pre-elected) primary node of a cluster (i.e., one of the clusters that store data records accessed by the cross-shard transaction). Note that once a primary node of a cluster is elected, it initiates all intra-shard transactions of the cluster as well as cross-shard transactions that are sent to the cluster by clients. Upon receiving a valid request from a client, primary node π initiates the protocol among the involved clusters by multicasting a propose message including the transaction to *all* nodes of all involved clusters, i.e., all clusters that store data records accessed by the cross-shard transaction. Once a node receives a valid propose message, it sends an accept message to the primary. The primary waits for matching accept

messages from $f+1$ nodes of *each* involved cluster to ensure that the majority of every cluster agree with the order of the transaction (recall that each cluster includes $2f+1$ nodes). The primary then multicasts a commit message to all nodes of every involved cluster, appends the transaction block to its ledger, executes the transaction, and sends a reply to the client. Upon receiving a commit message from the primary, each node also appends the transaction block to its ledger.

Algorithm 9 presents the normal case operation for SharPer to process a cross-shard transaction in the presence of crash-only nodes. Although not explicitly mentioned, every sent and received message is logged by nodes. As indicated in lines 1-5 of the algorithm, p_i is the cluster that initiates the transaction, $\pi(p_j)$ represents the primary node of cluster p_j, and P is the set of involved clusters in the transaction.

As shown in lines 6-7, upon receiving a valid signed cross-shard request $m = \langle \text{REQUEST}, op, \tau_c, c \rangle_{\sigma_c}$ from an authorized client c (with timestamp τ_c) to execute operation op, the primary node $\pi(p_i)$ of the initiator cluster p_i assigns sequence number h_i to the request and multicasts a propose message $\langle \text{PROPOSE}, h_i, d, m \rangle$ to the nodes of every involved cluster where m is the client's request message and $d = D(m)$ is the digest of m. The sequence number h_i represents the correct order of the transaction block in the initiator cluster p_i. Since all nodes are crash-only, there is no need to sign messages.

Upon receiving a propose message, as indicated in lines 8-10, each node r of an involved cluster p_j validates the message and its sequence number. If node r is currently waiting for a commit message of some cross-shard request m' where the involved clusters of two requests m and m' intersect in some other cluster p_k, the node does not process the new request m before the earlier request m' gets committed. This ensures that requests are committed in the same order on different clusters. Otherwise, node r sends an accept message $\langle \text{ACCEPT}, h_i, h_j, d, r \rangle$ to primary node $\pi(p_i)$ where h_j is the sequence number, assigned by r, that represents the correct order of request m in cluster p_j and d is the

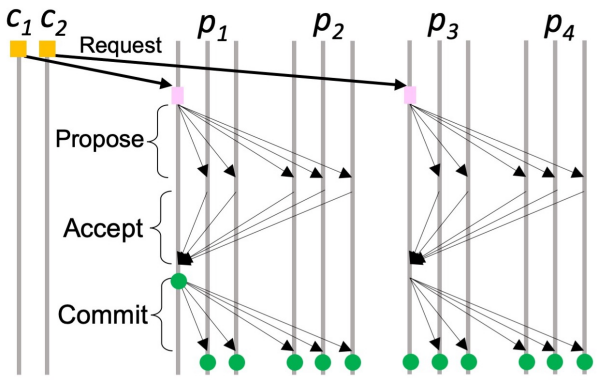


Figure 6.3: Two *concurrent* cross-shard transaction flows for crash-only nodes

digest of m.

Once primary $\pi(p_i)$ receives valid matching accept messages from $f+1$ nodes of *every* involved cluster p_j with matching h_j and also h_i and d that match to the propose message sent by $\pi(p_i)$, as presented in lines 11-13, it collects all valid sequence numbers (e.g., h_i, h_j, ..., h_k) from the accept messages of all involved clusters (e.g., p_i, p_j, ..., p_k) and multicasts a commit message $\langle \text{COMMIT}, [h_i, h_j, ..., h_k], d \rangle_{\sigma_{\pi(p_i)}}$ to the nodes of all involved clusters. The order of sequence numbers h_i, h_j, ..., h_k in the message is an ascending order determined by their cluster ids. The sequence number, indeed, consists of multiple sub-sequence numbers where each sub-sequence number presents the local order of the transaction in one of the involved clusters. The primary signs its commit message because it might be used later by nodes to prove the correctness of the transaction block.

Finally, as shown in lines 14-15, once a node of some cluster p_j receives a valid commit message from primary $\pi(p_i)$, the node considers the transaction as committed (even if the node has not sent an accept message for that request). If all transactions with lower sequence numbers than h_j has been committed, the node appends the transaction as well as the corresponding commit message to the ledger and executes it. This ensures that all replicas execute requests in the same order as required to provide the safety property.

Figure 6.3 shows the normal case operation for SharPer to execute two concurrent

cross-shard transactions in the presence of crash-only nodes where each transaction accesses two disjoint shards. The network consists of four clusters where each cluster includes three nodes ($f = 1$).

6.3.3 Dealing with Conflicting Messages

In the presented consensus protocol and after multicasting propose messages, the primary might not receive a quorum of *matching* accept messages from $f+1$ nodes of every involved cluster (i.e., received accept messages might have different sequence numbers) for two reasons. First, the primary nodes of different clusters might multicast their propose messages in parallel, hence, different nodes in an overlapping cluster might receive the messages in different order. Second, nodes might assign inconsistent sequence numbers since they have not necessarily received the latest propose message from the primary of their own cluster. We now propose an optimization to reduce the likelihood of such conflicts. This optimization is demonstrated in Algorithm 10. In case of non-matching accept messages, as indicated in lines 1-2 of Algorithm 10, primary $\pi(p_i)$ needs to re-initiate the request in *only* the *conflicting clusters*, i.e., clusters that have not sent $f + 1$ matching accept messages to the primary node. However, to prevent any further conflicts, primary $\pi(p_i)$ multicasts a super-propose message with the same structure as propose messages *only* to the primary nodes of the conflicting clusters. Once primary $\pi(p_i)$ sends a super-propose message for transaction m to the primary node of a cluster, $\pi(p_i)$ does not accept any further accept messages for transaction m from that cluster. As shown in lines 3-4, the primary node of each conflicting cluster then assigns a sequence number and multicasts a super-accept message (with the same structure as accept messages) to the nodes of its cluster and also the initiator primary $\pi(p_i)$. Upon receiving a super-accept message from the primary of its cluster, as presented in lines 5-6, each node logs the message and sends

a super-accept message with the same sequence number to primary $\pi(p_i)$. Once primary $\pi(p_i)$ has received valid matching super-accept messages from $f+1$ nodes of *every* conflicting cluster, it returns to its normal operation, as presented in lines 16-18 of Algorithm 9, and multicasts commit messages.

In heavy workloads with a high percentage of cross-shard transaction, the probability of receiving conflicting accept messages might be high. Therefore, instead of multicasting propose messages, waiting for probably conflicting accept messages and then re-initiating the transaction by multicasting super-propose messages, the primary node of the initiator cluster can initially multicast super-propose messages to the primary nodes of other involved clusters as well as the nodes of its own cluster. In this way, since the primary of each cluster assigns all sequence numbers for both intra-shard and cross-shard transactions, no conflicts will occurs. It should be noted that, this solution comes with an extra (intra-cluster) message passing from the primary to the nodes of each cluster. Note that depending on the type of workload and percentage of cross-shard transactions, SharPer can dynamically switch between these two techniques to deal with conflicting messages efficiently.

To deal with conflicting cross-shard transactions, i.e., cross-shard transactions that are initiated in parallel and overlap in some clusters, the system designer can also specify *mega-primary* nodes. A mega-primary node is the primary node of one of the clusters in any set P which initiates all cross-shard transactions that access *all* clusters in P. In particular, any transaction that accesses a set of clusters $\{p_i, p_j, ..., p_k\}$ is initiated by the primary node of cluster i where $i = min(i, j, ..., k)$. For example, if SharPer includes three clusters p_1, p_2, and p_3, using a mega primary, cross-shard transactions that access two clusters p_1 and p_2, two clusters p_1 and p_3, or all three clusters p_1, p_2, and p_3 are initiated by the primary node of p_1 (since $1 = min(1, 2, 3)$) and cross-shard transactions that access two clusters p_2 and p_3 are initiated by the primary node of p_2

Algorithm 10 Dealing with Conflicting ACCEPT Messages

******The configuration is the same as Algorithm 9******
1: if accept messages of cluster p_j not matching and $r == \pi(p_i)$
2: **multicast** $\langle$SUPER-PROPOSE,h_i,d,$m\rangle$ to $\pi(p_j)$

3: **upon receiving** $\langle$SUPER-PROPOSE,h_i,d,$m\rangle$ from $\pi(p_i)$ and $r==\pi(p_j)$
4: **multicast** $\langle$SUPER-ACCEPT, $h_i, h_j, d, r\rangle$ to $\pi(p_i)$ and all nodes of p_j

5: **upon receiving** $\langle$SUPER-ACCEPT,h_i,h_j,d,$\pi(p_j)\rangle$ from $\pi(p_j)$ and $r \in p_j$
6: **send** $\langle$SUPER-ACCEPT, $h_i, h_j, d, r\rangle$ to $\pi(p_i)$

(since $2 = min(2, 3)$). Note that different systems can specify mega primary nodes in different ways depending on the workload and geographical distance between clusters.

6.3.4 Primary Failure Handling

The goal of the primary failure handling routine is to provide liveness by allowing the system to make progress when a primary fails. It prevents replicas from waiting indefinitely for requests to execute. The primary failure handling routine must guarantee that it will not introduce any changes in a history that has been already completed at a correct client. The routine is triggered by timeout. When node r of some cluster p_j receives a valid propose message from a primary for either an intra-shard or a cross-shard transaction, it starts a timer that expires after some predefined time τ. Time τ for cross-shard transactions is larger because processing cross-shard transactions requires agreement from all involved clusters and takes longer. If the timer has expired and the node has not received any message from the primary node, the node suspects that the primary is faulty. The primary failure handling routine is performed by the nodes of the same cluster as the faulty primary. However, if a node r is involved in a cross-shard transaction that was initiated by some other cluster p_i and the timer of r has expired, node r (of cluster p_j) multicasts an accept-query message $\langle$ACCEPT-QUERY, $h_i, h_j, d, r\rangle$ message to every node of the initiator cluster p_i (the cluster of the faulty primary) where h_i and h_j are the sequence numbers assigned to the transaction by clusters p_i and p_j (in

the corresponding propose and accept messages). Note that nodes of a cluster do not participate in the primary failure handling routine of other clusters except for sending accept-query messages. Otherwise (when the node r and the faulty primary are in the same cluster), the protocol uses the leader election phase of Paxos [20] to elect the new primary, and the new primary will handle all the uncommitted intra- and cross-shard transactions, and take care of the new client requests. Indeed, similar to Paxos, the node r tries to become the primary node of the cluster by multicasting a prepare message $\langle \text{PREPARE}, H, r \rangle$ to every node of its cluster where H is a proposal number higher than every sequence number received from the previous primary nodes. If a node q receives a prepare message with a proposal number H higher than every previous sequence or proposal number received, the node returns a promise message $\langle \text{PROMISE}, H, q \rangle$ to the sender. The node that receives f promise messages (including itself becomes $f + 1$) becomes the new primary.

Once the new primary is elected, it multicasts accept-query message $\langle \text{ACCEPT-QUERY}, h \rangle$ to the nodes of its cluster for any sequence number h ($h < H$) that is still uncommitted (either unknown or accepted). If h is a cross-shard transaction, the primary multicasts the accept-query message to every node of all involved clusters. The primary becomes aware of transaction type (intra- or cross-shard) and hence the involved clusters either from the received propose message or from the received responses. Once a node receives an accept-query message for some sequence number h, if the node has already received a commit message (from the previous primary) for sequence number h, it sends a committed message $\langle \langle \text{COMMITTED}, h, d \rangle, m \rangle$ to the new primary where m is the commit message received from the previous primary. Note that for cross-shard transactions, as explained earlier, h is a combination of several sequence numbers (one per each involved cluster). Otherwise, if the node has received a propose message for sequence number h, sent an accept message to the previous primary but has not received a commit message, it resends its accept message

to the new primary. Finally, if the node has not received either propose or commit message for sequence number h_i, it sends an unknown message $\langle \text{UNKNOWN}, h, \varnothing \rangle$ to the primary.

The primary collects all responses for each sequence number h. If the primary has received a commit message or $f+1$ matching propose messages (from each involved cluster in case of a cross-shard transaction) for a sequence number h, it multicasts a commit message to every node (of all involved clusters). Else, if the primary has received at least one (and at most f) matching propose messages (from any involved cluster in case of cross-shard transactions) for a sequence number h, it multicasts a propose message to every node (of all involved clusters). Otherwise, the primary multicasts a propose message $\langle \text{PROPOSE}, h_i, d, \text{no-op} \rangle$ to the backups where the "no-op" command leaves the state unchanged. The last situation happens when the previous primary has assigned sequence number and multicast propose messages to every node, however, no one has received its message. Once all unknown transactions to the primary are processed, the primary starts processing new transactions.

6.3.5 Correctness Arguments

Consensus protocols have to satisfy *safety* and *liveness* properties. Safety means all correct nodes receive the same requests in the same order whereas liveness means all correct requests are eventually ordered. In this section, the safety (agreement, validity, and consistency) and liveness (termination) properties of SharPer in the presence of crash-only nodes are demonstrated. Since intra-shard transactions follow Paxos, we mainly focus on cross-shard transactions.

Lemma 6.3.1 (*Agreement*) If node r commits request m with sequence number h, no other correct node commits request m' ($m \neq m'$) with the same sequence number h.

Proof: Let m and m' ($m \neq m'$) be two committed requests with sequence numbers

$h = [h_i, h_j, h_k, ...]$ and $h' = [h'_k, h'_l, h'_m, ..]$ respectively. Committing a request requires matching accept messages from $f+1$ different nodes of *every* involved cluster. Therefore, if the involved clusters of m and m' intersect in cluster p_k, at least $f+1$ nodes of cluster p_k have sent matching accept messages for m, and similarly, at least $f+1$ nodes of cluster p_k have sent matching accept messages for m'. Since each cluster includes $2f+1$ nodes and nodes are non-malicious, $h_k \neq h'_k$. Note that the same proof logic applies in special cases where m or m' is an intra-shard transaction (i.e., $h = h_k$ or $h' = h'_k$).

If the primary fails, since each committed request has been replicated on a quorum Q_1 of $f+1$ nodes and to become elected primary agreement from a quorum Q_2 of $f+1$ nodes is needed, Q_1 and Q_2 must intersect in at least one node that is aware of the latest committed request. Hence, SharPer guarantees the agreement property for both intra-shard as well as cross-shard transactions. ■

Lemma 6.3.2 (*Validity*) If a correct node r commits m, then m must have been proposed by some correct node π.

Proof: In cross-shard consensus with crash-only nodes, validity is ensured since crash-only nodes do not send fictitious messages. ■

Lemma 6.3.3 (*Consistency*) Let P_μ denote the set of involved clusters for a request μ. For any two committed requests m and m' and any two nodes r_1 and r_2 such that $r_1 \in p_i$, $r_2 \in p_j$, and $\{p_i, p_j\} \in P_m \cap P_{m'}$, if m is committed before m' in r_1, then m is committed before m' in r_2.

Proof: As mentioned in Section 6.3.2, once a node r_1 of some cluster p_i receives a propose message for some cross-shard transaction m, if the node is involved in some other uncommitted cross-shard transaction m' where $|P_m \cap P_{m'}| > 1$, i.e., some other cluster p_j is also involved in both transactions, node r_1 does not send an accept message

for transaction m before m' gets committed. In this way, since committing request m requires $f+1$ accept messages from *every* involved cluster, m cannot be committed until m' is committed. As a result the order of committing messages is the same in all involved nodes. In the special case where $i = j$ (both nodes r_1 and r_2 belong to the same cluster), if the primary of the cluster assigns the sequence number, there will be no inconsistency among nodes. Otherwise, when the nodes assign sequence numbers and even if r_1 and r_2 initially assign inconsistent sequence numbers, since at least $f+1$ matching accept messages from different nodes of the cluster are needed to commit a request and the cluster includes $2f+1$ nodes, the order of committing transactions on nodes r_1 and r_2 must be consistent. ■

It should be noted that in such a case we might face a deadlock where different clusters might need to re-initiate their transactions after some predefined time. To prevent any further deadlock, SharPer defines different waiting times for different clusters.

Lemma 6.3.4 (*Termination*) A request m issued by a correct client eventually completes.

Proof: SharPer, as mentioned earlier and due to the FLP impossibility result [53], guarantees liveness only during periods of synchrony. To show that a request issued by a correct client eventually completes, we need to address three scenarios. First, if the primary is non-faulty and accept messages are non-conflicting, as shown in Algorithm 9, the protocol ensures that the correct client receives reply from the primary. Second, if a non-faulty primary has multicast propose messages but not received matching accept messages from $f+1$ nodes of every involved clusters, as explained in Sections 6.3.3, the primary re-initiates the transaction by multicasting super-propose messages to only the primary nodes of the involved clusters. In this way, since the primary node of each cluster assigns the sequence number (in its super-accept message), super-accept messages that are

received from each cluster are matching, thus increasing the chances of termination. If the primary node of any involved cluster has failed before multicasting super-accept messages, the primary failure handling routine will trigger and the new elected primary will handle the transaction. Third, if the primary fails, as explained in Sections 6.3.4, the nodes that are involved in an uncommitted transaction (initiated by the faulty primary) detect its failure (using timeouts) and trigger the primary failure handling. The new primary then will handle all uncommitted transactions. ■

6.4 Consensus with Byzantine Nodes

In this section, intra- and cross-shard consensus in the presence of Byzantine nodes are presented first followed by the view change routine. Then, the correctness of SharPer with malicious failures is proven, and finally, an optimization for clustered networks is discussed,

6.4.1 Intra-Shard Consensus

Most Byzantine fault-tolerant protocols, e.g., PBFT [21], require $3f+1$ nodes to guarantee safety in the presence of at most f *malicious* nodes. PBFT consists of *agreement* and *view change* routines where the agreement routine orders requests for execution by the nodes, and the view change routine coordinates the election of a new primary when the current primary is faulty. The nodes move through a succession of configurations called *views* [94] [95] where in each view, one node, called *the primary*, initiates the protocol and the others are *backups*.

To establish consensus on the order of intra-shard transactions during a normal case execution of PBFT, a client c requests an intra-shard transaction by sending a signed request message to the primary. When the primary receives a valid request from an

authorized client, it initiates the consensus protocol by assigning a sequence number and multicasting a propose (called pre-prepare in original PBFT) message including the requested transaction to all nodes within the cluster. Once a node receives a valid propose message from the primary, it multicasts an accept (prepare) message to every node within the cluster. Each node then waits for $2f$ valid accept messages from different nodes (including itself) that match the propose message and then multicasts a commit message to all the nodes within the cluster. Once a node receives $2f$ valid commit messages from different nodes that match its own commit message, it appends the transaction block including all $2f + 1$ commit message to the ledger (to ensure immutability), executes the transaction, and sends a reply to the client. Finally, the client waits for $f+1$ valid matching responses from different replicas to make sure at least one correct replica executed its request. If the client does not receive reply messages soon enough, it multicasts the request to all nodes within the cluster. If the request has already been processed, the nodes simply re-send the reply message to the client (replicas remember the last reply message they sent to each client). Otherwise, if the node is not the primary, it relays the request to the primary. If the primary does not multicast the request to the nodes of the cluster, it will eventually be suspected to be faulty by nodes to cause a view change.

6.4.2 Cross-Shard Consensus with Byzantine Nodes

In the presence of malicious nodes, a Byzantine fault-tolerant protocol is needed to order cross-shard transactions where for each cross-shard transaction, similar to the crash-only case, agreement from all involved clusters is needed. Unlike in the case of crash failure where the quorum size is $f + 1$, in consensus with Byzantine nodes, the quorum size is $2f + 1$. In addition and due to the potential malicious behaviour of the primary node, all nodes of every involved cluster multicast both accept and commit

Algorithm 11 Cross-shard Consensus with Byzantine Nodes

1: **init():**
2: r := *node_id*
3: p_i := the cluster that initiates the consensus
4: $\pi(p_j)$:= the primary node of p_j
5: P := set of involved clusters

6: **upon receiving** valid transaction m and $(r == \pi(p_i))$
7: **multicast** $\langle\langle \mathsf{PROPOSE}, h_i, v_i d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ to P

8: **upon receiving** valid $\langle\langle \mathsf{PROPOSE}, h_i, v_i d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ from $\pi(p_i)$
9: if r is not involved in any uncommitted request m' where m and m' intersect in some other cluster p_k
10: **multicast** $\langle \mathsf{ACCEPT}, h_i, h_j, v_i, v_j, d, r\rangle_{\sigma_r}$ to P

11: **upon receiving** valid matching $\langle \mathsf{ACCEPT}, h_i, h_j, v_i, v_j, d, r\rangle_{\sigma_r}$ from $2f{+}1$ different nodes of every cluster p_j in P
12: **multicast** $\langle \mathsf{COMMIT}, h_i, h_j, ..., h_k, v_i, v_j, ..., v_k, d, r\rangle_{\sigma_r}$ to P

13: **upon receiving** valid $\langle \mathsf{COMMIT}, h_i, h_j, ..., h_k, v_i, v_j, ..., v_k, d, r\rangle_{\sigma_r}$ from $2f + 1$ nodes of every cluster in P
14: **append** the transaction and commit messages to the ledger

messages to each other. In cross-shard consensus with Byzantine node, similar to PBFT, nodes of each cluster move through views where views are numbered consecutively. Node π $(1 \leq \pi \leq 3f{+}1)$ is *the primary* of view v if $\pi{=}(v \mod (3f{+}1))$.

In the presence of malicious nodes, and upon receiving a valid request (cross-shard transaction) from a client, similar to the crash-only case, primary node π initiates the protocol among the involved clusters by multicasting a propose message including the transaction to *all* nodes of all involved clusters. Once a node receives a valid propose message, it multicasts an accept message to all nodes of every involved clusters. Each node then waits for $2f + 1$ matching valid accept messages from different nodes of *each* involved cluster before multicasting a commit message to all nodes of the involved clusters. Upon receiving $2f{+}1$ matching valid commit message from different nodes of *each* involved cluster, each node appends the transaction block to the ledger.

The normal case operation for SharPer to process a cross-shard transaction in the presence of Byzantine nodes is presented in Algorithm 11. Similar to Algorithm 9 and as shown in lines 1-5, p_i is the initiator cluster, P is the set of involved clusters, and $\pi(p_j)$ indicates the primary node of cluster p_j.

Once the initiator primary $\pi(p_i)$ receives a valid signed cross-shard request from an

authorized client, as presented in lines 6-7, the primary assigns sequence number h_i to the request and multicasts a *signed* propose message including sequence number h_i, view number v_i (that indicates the view of cluster p_i in which the message is being sent) and digest d of the request. As before, sequence number h_i is used to ensure that the new transaction block is ordered correctly with respect to the blocks that the cluster has been involved in. Requests are piggybacked in propose messages to keep propose messages small. Since the network might include malicious nodes, the primary signs its message.

Once a node r of an involved cluster p_j receives a propose message for a request m, as indicated in lines 8-10, it validates the signature and message digest. If the node belongs to the initiator cluster ($i = j$), it also checks h_i to be valid (within a certain range) since a malicious primary might multicast a request with an invalid sequence number. Furthermore, if the node is currently involved in an uncommitted cross-shard request m' where the involved clusters of two requests m and m' overlap in some other cluster, as explained in the crash-only case, the node does not process the new request m before the earlier request m' is processed. This is needed to ensure requests are committed in the same order on different clusters. The node then multicasts a signed accept message including the corresponding sequence number h_j (that represents the order of request m in cluster p_j), the view number v_j of cluster p_j as well as the digest d of request m to *every* node of *all* involved clusters.

Each node waits for valid accept messages with matching sequence and view numbers from $2f+1$ nodes of *every* involved cluster with h_i, and d that match the propose message which is sent by primary $\pi(p_i)$. We define the predicate $\mathsf{accepted\text{-}local}_{p_j}(m, h_i, h_j, v_i, v_j, r)$ to be true if and only if node r has received the request m, a propose for m with sequence number h_i in view v_i of the initiator cluster p_i and $2f + 1$ singed accept messages form different nodes of an involved cluster p_j that match the propose message. The predicate $\mathsf{accepted}(m, h, v, r)$ where $h = [h_i, h_j, ..., h_k]$ and $v = [v_i, v_j, ..., v_k]$ is then defined to be

true on node r if and only if accepted-local$_{p_j}$ is true for *every* involved cluster p_j in cross-shard request m. The order of sequence numbers and view numbers in the predicate is an ascending order determined by their cluster ids. Here, since nodes might behave maliciously, each cluster includes $3f + 1$ nodes and $2f + 1$ matching messages from *all* involved clusters for each step of the protocol are needed. The propose and accept phases of the algorithm basically guarantee that non-faulty nodes agree on a total order for the transactions. When accepted(m, h, v, r) becomes true, as presented in lines 11-12, the node r multicasts a signed commit message $\langle \text{COMMIT}, h, v, d, r \rangle_{\sigma_r}$ to every node of all involved clusters.

Finally, as shown in lines 13-14, each node waits for valid matching signed commit messages from $2f + 1$ nodes of *every* involved clusters that match its commit message. The predicate committed-local$_{p_j}(m, h, v, r)$ is defined to be true on node r if and only if accepted(m, h, v, r) is true and node r has accepted $2f+1$ valid matching commit messages from different nodes of cluster p_j that match the propose message for cross-shard request m. The predicate committed(m, h, v, r) is then defined to be true on node r if and only if committed-local$_{p_j}$ is true for *every* involved cluster p_j in cross-shard request m. The committed predicate indeed shows that at least $f + 1$ nodes of each involved cluster have multicast valid commit messages. When the committed predicate becomes true, the node considers the transaction as committed. If the node has executed all transactions with lower sequence numbers than h_j, it appends the transaction as well as the corresponding commit message to the ledger and executes it.

Figure 6.4 shows the processing of two concurrent cross-shard transactions in the presence of Byzantine nodes where each transaction accesses two disjoint data shards. The network consists of four clusters where each cluster includes four nodes ($f = 1$).

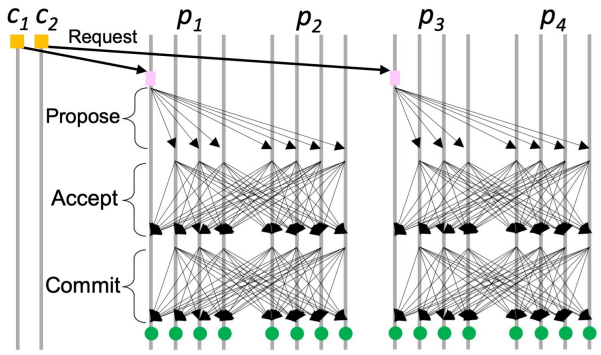


Figure 6.4: Two *concurrent* cross-shard transaction flows for Byzantine nodes

6.4.3 Dealing with Conflicting Messages

Algorithm 12 Dealing with Conflicting ACCEPT Messages

******The configuration is the same as Algorithm 11******

1: if accept messages of cluster p_j not matching and $(r == \pi(p_j))$

2: **multicast** $\langle$SUPER-ACCEPT, $h_i, h_j, v_i, v_j, d, r\rangle_{\sigma_r}$ to nodes of p_j

3: **upon receiving** $\langle$SUPER-ACCEPT, $h_i, h_j, v_i, v_j, d, \pi(p_j)\rangle_{\sigma_{\pi(p_j)}}$ and $r \in p_j$

4: if less than $2f+1$ valid accept messages from p_j for m is logged

5: **multicast** $\langle$SUPER-ACCEPT, $h_i, h_j, v_i, v_j, d, r\rangle_{\sigma_r}$ to P

In the consensus protocol with Byzantine nodes, similar to the crash-only case, nodes might not receive a quorum of $2f+1$ *matching* accept messages from every cluster due to conflicting accept messages. In such a situation, as presented in lines 1-2 of Algorithm 12, the primary node of each conflicting cluster p_j (i.e., a cluster with less than $2f+1$ matching accept messages) multicasts a super-accept message (with the same structure as accept messages) to the nodes of its own cluster. Once a node receives a super-accept message for some cross-shard transaction m from the primary node of its cluster, as shown in lines 3-5, it first validates the message. If the node has already received $2f+1$ matching accept messages for transaction m from the nodes of its cluster (which might happen in case of a malicious primary), the node simply ignores the received super-accept message. Otherwise, the node multicasts a super-accept message to all nodes of every

involved cluster.

In heavy workloads with a high percentage of cross-shard transactions, since the probability of receiving conflicting accept messages is high, Similar to the crash-only case, the primary node of the initiator cluster can initially multicast super-propose messages to the primary nodes of other involved clusters as well as the nodes of its own cluster.

In addition, SharPer can employ the mega-primary technique where for any set P of clusters, the primary node of one of the clusters, called *mega-primary*, initiates all cross-transactions that access *all* clusters in P.

6.4.4 View Change

The view change routine provides liveness by allowing the system to make progress when a primary fails. Similar to the crash-only case, view changes are triggered by timeout. If the timer of some node r expires, node r suspects that the primary is faulty and starts a view change. There are two cases. First, if the initiator primary is not in the cluster of node r, similar to the crash-only case, node r multicasts a signed accept-query message to every node of the initiator cluster (the cluster of the faulty primary). If a node receives accept-query messages from $2f + 1$ different nodes of another cluster, the node suspects that the primary of its cluster is faulty and initiates a view change. Second, when node r and the faulty primary are in the same cluster, similar to PBFT, node r initiates a view change. To begin the view change routine, node r stops accepting propose, accept, super-accept, and commit messages and multicasts a view-change message $\left\langle \text{VIEW-CHANGE}, v+1, h, \xi, \mathcal{A}, \mathcal{C}, r \right\rangle_{\sigma_r}$ to every node within its cluster where h is the sequence number of the last stable checkpoint (is explained later) known to r, ξ is the proof of checkpoint, $\mathcal{A}$ is the set of received valid intra- and cross-shard accept and super-accept messages, and $\mathcal{C}$ is the set of received valid commit messages for requests with a sequence

number higher than h. An accept-query message is valid if it is received from at least $2f+1$ different nodes of the same cluster. Note that, SharPer, similar to PBFT, use the state transfer technique to checkpoint the state of different nodes. Each node generates checkpoints periodically when a request sequence number is divisible by some constant (checkpoint period) and multicasts them to other nodes in its cluster. Once a node has received $2f+1$ checkpoint messages (called the proof of checkpoint) for a sequence number, the checkpoint becomes stable.

When primary $\pi'(p_j)$ of new view $v+1$ receives $2f$ valid view-change messages from different nodes of its cluster, it multicasts a $\Big\langle \text{NEW-VIEW}, v{+}1, \Sigma, \mathcal{P}', \mathcal{C}', \Big\rangle_{\sigma_{\pi'(p_j)}}$ message to all nodes where Σ is the set of $2f+1$ valid view-change messages ($2f$ messages from other nodes plus its own message), and $\mathcal{P}'$ and $\mathcal{C}'$ are two sets of propose and commit messages respectively which are constructed as follows.

Let l be the sequence number of the latest checkpoint, and h be the highest sequence number of a propose message in all the received $\mathcal{A}$ sets. For each sequence number n where $l < n \leq h$.

• It first checks all commit messages in set $\mathcal{C}$ of the received view-change messages. If the primary finds $2f+1$ valid matching commit messages from either cluster p_j for intra-shard request m or from every cluster for cross-shard request m, the primary adds the commit messages to $\mathcal{C}'$.

• If the primary node $\pi'(p_j)$ finds a set of $2f+1$ matching valid accept messages for an intra-shard transaction or a set of $2f+1$ matching valid accept or super-accept messages coming from the same cluster for a cross-shard transaction, the primary $\pi'(p_j)$ adds a $\Big\langle \text{PROPOSE}, v{+}1, n, d \Big\rangle_{\sigma_{\pi'(p_j)}}$ to $\mathcal{P}'$ where d is the digest of the request.

• If accept messages of the nodes of its cluster are not matching and the request is a cross-shard transaction initiated by other cluster, the primary assigns a sequence number

and adds $\left\langle \text{SUPER-ACCEPT}, v{+}1, n, d \right\rangle_{\sigma_{\pi'(p_j)}}$ to $\mathcal{P}'$ where d is the digest of the request.

• Otherwise, the primary adds a $\left\langle \text{PROPOSE}, v{+}1, n, d^{\emptyset} \right\rangle_{\sigma_{\pi'(p_j)}}$ to $\mathcal{P}'$ where $d^{\emptyset}$ is the digest of a special no-op command that is transmitted by the protocol like other requests but leaves the state unchanged.

The primary then inserts all the messages in $\mathcal{P}'$ and $\mathcal{C}'$ to its log. It also checks the log to make sure its log contains the latest stable checkpoint. If not, the primary inserts checkpoint messages for the checkpoint l and obtain missing blocks in its blockchain form another node. For each cross-shard transaction, the primary also multicasts the corresponding message, e.g., propose, accept, or super-accept to the nodes of all involved clusters.

Once a node in view v receives a valid new-view message from the primary of view $v+1$, the node logs all messages, updates its checkpoint in the same way as the primary, and for each propose or super-accept message, multicasts an accept or super-accept message (respectively) to the nodes of the involved clusters. Note that non-faulty nodes in view v will not accept a propose message for a new view $v' > v$ without having received a new-view message for v'.

Note that nodes redo the protocol for requests with sequence number between l and h, however, they do not re-execute requests. In addition, if a node does not have a request message or or a stable checkpoint, it obtains missing information from another node.

6.4.5 Correctness Arguments

In this section we demonstrate how SharPer satisfies safety (agreement, validity, and consistency) and liveness (termination) properties in the presence of Byzantine nodes.

Lemma 6.4.1 (*Agreement*) If node r commits request m with sequence number h, no other correct node commits request m' ($m \neq m'$) with the same sequence number h.

Proof: The propose and accept phases of the Byzantine cross-shard consensus protocol guarantee that correct nodes agree on a total order of requests within a view. Indeed, if the accepted(m, h, v, r) predicate where $h = [h_i, h_j, ..., h_k]$ and $v = [v_i, v_j, ..., v_k]$ is true, then accepted(m', h, v, q) is false for any non-faulty node q (including $r = q$) and any m' such that $m \neq m'$. This is true because (m, h, v, r) implies that accepted-local$_{p_j}(m, h_i, h_j, v_i, v_j, r)$ is true for each involved cluster p_j and since each cluster include $3f+1$ nodes, at least $2f+1$ nodes within the cluster (from which at least $f+1$ nodes are non-faulty) have sent accept (or propose) messages for request m with sequence number h_j in view v_j. As a result, for accepted(m', h, v, q) to be true, at least one of those non-faulty nodes needs to have sent two conflicting accept messages with the same sequence number, same view number, but different message digest. This condition guarantees that first, a malicious primary cannot violate the safety and second, at most one of the concurrent *conflicting* transactions, i.e., transactions that overlap in at least one cluster, can collect the required number of messages $(2f + 1)$ from each overlapping cluster.

Across different views, the view-change routine of SharPer guarantees that non-faulty nodes of some cluster p_j agree on the sequence number of requests that are committed-local in different views at different node. The committed-local$_{p_j}$ predicate becomes correct on node r if r has received a quorum Q_1 of $2f+1$ matching commit messages from different nodes of cluster p_j. To change the view of cluster p_j, a quorum Q_2 of $2f + 1$ valid view-change messages is needed. Since there are $3f + 1$ nodes in each cluster, Q_1 and Q_2 intersect in at least one correct node, thus if a request is accepted in a previous view, it is propagated to subsequent views. ■

Lemma 6.4.2 (*Validity*) If a correct node r commits m, then m must have been proposed by some correct node π.

Proof: In the presence of Byzantine nodes, validity is guaranteed mainly based

on standard cryptographic assumptions about collision-resistant hashes, encryption, and signatures which the adversary cannot subvert them (as explained in Section 6.2). Since the request as well as all messages are signed and either the request or its digest is included in each message (to prevent changes and alterations to any part of the message), and in each step $2f + 1$ matching messages (from each cluster) are required, if a request is committed, the same request must have been proposed earlier. ■

Lemma 6.4.3 (*Consistency*) Let P_μ denote the set of involved clusters for a request μ. For any two committed requests m and m' and any two nodes r_1 and r_2 such that $r_1 \in p_i$, $r_2 \in p_j$, and $\{p_i, p_j\} \in P_m \cap P_{m'}$, if m is committed before m' in r_1, then m is committed before m' in r_2.

Proof: Consistency is guaranteed in the same way as crash-only nodes (Lemma 6.3.3). ■

Lemma 6.4.4 (*Termination*) A request m issued by a correct client eventually completes.

Proof: To provide termination during periods of synchrony, similar to the crash-only case, three scenarios need to be addressed. If the primary in non-faulty and accept messages are non-conflicting, following Algorithm 11, request m completes. If the primary in non-faulty, however accept messages are conflicting, as mentioned in Section 6.4.3, the request will be re-initiated in the conflicting clusters using super-propose messages. Finally, view change routines (Section 6.4.4) handle primary failures. ■

6.4.6 An Optimization for Clustered Networks

We now illustrate how prior knowledge of where the faulty nodes are placed could help in increasing the number of clusters, and hence parallelism and overall performance.

In SharPer and in the presence of crash-only nodes, we assume that the number of nodes is much more than $2f + 1$ and therefore, partition the network into clusters of $2f + 1$ nodes. This is needed because we are not aware of *where* the f faulty nodes are placed. As a result, since they all might be in the same cluster, to guarantee safety each cluster includes $2f + 1$ nodes. Similarly, and in the presence of Byzantine nodes, each cluster consists of $3f + 1$ nodes. However, if we have some prior knowledge of where the faulty nodes are placed, we might be able to increase the number of clusters. In particular, nodes might be (geographically) partitioned into several groups (e.g., clouds) where f is known for each individual group of nodes. Hence, clustering can be performed within each group instead of the entire network. Indeed, different cloud environments might have different failure properties, e.g., while renting nodes from a particular cloud might be expensive, the maximum number of possible concurrent failures, f, in that cloud could be smaller than a cloud with cheaper nodes. As an example, consider a network of Byzantine nodes with $n = 23$ and $f = 3$ where nodes are partitioned into two groups of A and B (placed in two different cloud environments) such that $n_A = 7$, $n_B = 16$, $f_A = 2$, and $f_B = 1$. Without being aware of A and B, since there are totally 23 nodes and $f = 3$, the number of clusters is $|P| = \frac{n}{3f+1} = \frac{23}{10} = 2$. However, knowing f_A and f_B, we can cluster A and B separately and as a result, $|P_A| = \frac{n_A}{3f_A+1} = \frac{7}{7} = 1$ and $|P_B| = \frac{n_B}{3f_B+1} = \frac{16}{4} = 4$. Thus, the network is partitioned into five clusters (three more clusters in comparison to the previous case). This is useful especially in cloud environments where nodes are placed in different clouds with different properties (e.g., different f). Note that the same technique can be applied when the nodes are crash-only.

6.5 Experimental Evaluations

In this section, we conduct several experiments to evaluate SharPer. We have implemented a blockchain-based accounting application where the data records are client accounts (every client might have several accounts). Clients of the application can initiate transactions to transfer assets from one or more of their accounts to other accounts (accounts might be in the same shard or different shards). A transaction might read and write several records. The experiments were conducted on the Amazon EC2 platform. Each VM is c4.2xlarge instance with 8 vCPUs and 15GB RAM, Intel Xeon E5-2666 v3 processor clocked at 3.50 GHz. When reporting throughput measurements, we use an increasing number of clients running on a single VM, until the end-to-end throughput is saturated, and state the throughput (x axis) and latency (y axis) just below saturation. Throughput and latency numbers are reported as the average measured during the steady state of an experiment.

6.5.1 Impact of Cross-Shard Transactions with Crash-Only Nodes

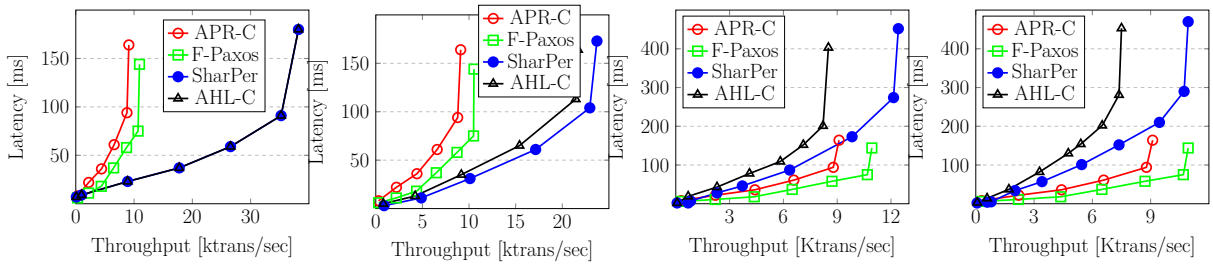
In the first set of experiments, we measure the performance of SharPer for workloads with different percentages of cross-shard transactions where nodes are crash-only. We compare SharPer with the two main approaches for exploiting the availability of extra resources: the active/passive replication technique and Fast Paxos [90]. We implemented two permissioned blockchain systems referred to as *APR-C* and *FPaxos* where their consensus protocols follow the active/passive replication and Fast Paxos designs respectively. In addition to SharPer and these two systems, we also implemented a modified version of the state of the art sharded permissioned blockchain system AHL [86]. AHL has two novel aspects: first, its intra-shard consensus protocol that uses trusted hardware to restrict the malicious behavior of nodes, and second, its cross-shard consensus protocol

where a reference committee uses 2PC to order the transactions. Since the emphasis of the experiments is on cross-shard transactions, we implemented a modified version of AHL, called AHL-C where the intra-shard transactions are processed similar to SharPer, however, the cross-shard transactions are performed similar to AHL [86]. In this set of experiments, since the nodes are crash-only, the reference committee uses Paxos [20] to establish consensus. Note that, since intra-shard consensus is pluggable, the trusted hardware technique can be employed in SharPer as well.

We consider a network with 12 nodes. In SharPer and AHL-C, the nodes are divided into four clusters where each cluster consists of 3 nodes and uses Paxos with f=1 to establish consensus. In AHL-C, a reference committee of three crash-only nodes is also considered to establish consensus on the order of cross-shard transactions. The data is also equally sharded into four shards. In the APR-C blockchain system, 3 nodes are used as the active replicas and the execution results are sent to the remaining 9 nodes whereas FPaxos uses 4 nodes ($3f + 1$) to establish consensus and the results are sent to the remaining 8 nodes.

We consider four different workloads with (1) no cross-shard transactions, (2) 20% cross-shard transactions, (3) 80% cross-shard transactions, and (4) 100% cross-shard transactions. We also assume that two (randomly chosen) shards are involved in each cross-shard transaction. Note that since APR-C and FPaxos do not use sharding, the percentage of cross-shard transactions does not affect their performance. The load is also equally distributed among all the nodes.

As can be seen in Figure 6.5(a), when there are no cross-shard transactions, SharPer is able to process 35230 transactions with 91 ms latency before the end-to-end throughput is saturated (the penultimate point). Note that in this setting, since there are no cross-shard transactions, each cluster orders and executes its transactions independently, thus the throughput of the entire system will increase linearly with the increasing number of



(a) 0% Cross-shard (b) 20% Cross-shard (c) 80% Cross-shard (d) 100% cross-shard

Figure 6.5: Increasing the percentage of cross-shard transactions (crash-only nodes)

clusters. Since for intra-shard transactions, AHL-C uses the same technique as SharPer, its results are identical to SharPer. APR-C and FPaxos are also able to process 8800 and 10700 transactions with 95 ms and 75 ms latency respectively before the end-to-end throughput is saturated (the penultimate points). Note that since FPaxos establishes consensus in less number of phases, it has better performance than APR-C. However, they both have much lower throughput in comparison to SharPer (25% and 33% of SharPer at 60 ms latency). The results mainly demonstrate the effectiveness of employing the sharding technique in blockchain.

By increasing the percentage of cross-shard transactions to 20% (Figure 6.5(b)), the throughput is reduced due to the overhead of cross-shard transactions. In this setting, SharPer is still able to process 23000 transaction with 100 ms latency (the penultimate point) whereas AHL-C processes 21000 transactions at the same latency. This is expected because first, SharPer, in contrast to AHL-C, is able to process non-overlapping cross-shard transactions in parallel, and second, the cross-shard protocol of SharPer involves less number of communication phases. As mentioned before, since the sharding technique is not utilized by APR-C and FPaxos, the percentage of cross-shard transactions does not affect their performance.

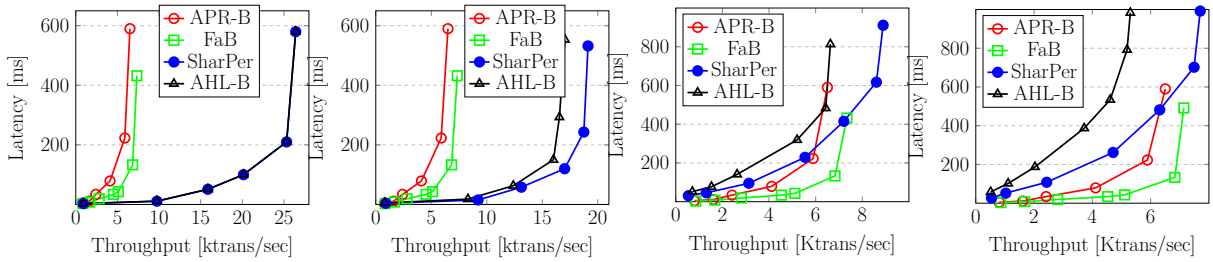
Similarly, increasing the percentage of cross-shard transactions to 80% (Figure 6.5(c))

and finally, 100% (Figure 6.5(d)) reduces the peak throughput of SharPer to 12300 and 10500, respectively. Note that by increasing the percentage of cross-shard transactions, SharPer still shows much better performance compare to AHL-C (44% better in their peak throughput with 100% cross-shard transactions) because SharPer is still able to process non-overlapping cross-transactions in parallel and also needs less number of communication phases. In these two scenarios, since APR-C and FPaxos order the transactions using only three ($2f+1$) and four ($3f+1$) nodes, their latency is lower than SharPer. Specially FPaxos processes transactions with significantly lower latency due to its fast consensus routine. However, since a large percentage of transactions is cross-shard, SharPer needs the participation of all involved clusters to order transactions and using sharding has no significant advantage.

6.5.2 Impact of Cross-Shard Transactions with Byzantine Nodes

In the second set of experiments, we repeat the previous scenarios on networks with Byzantine nodes. Similar to the previous section, we implement four permissioned blockchain systems: (1) SharPer, (2) APR-B where its consensus protocol follows the active/passive replication technique on Byzantine nodes), (3) FaB where its consensus protocol follows Fast Byzantine Consensus protocol [38] and uses $5f + 1$ nodes (instead of $3f + 1$) to establish consensus in two phases (instead of three as in PBFT), and (4) AHL-B where its intra-shard transactions are processed using PBFT (similar to SharPer) and its cross-shard transactions follow AHL [86].

We consider a network with 16 nodes. In SharPer and AHL-B, the nodes are partitioned into 4 clusters where each cluster consists of four nodes and uses PBFT protocol with $f = 1$ to establish consensus on its transactions. In addition to these 16 nodes, in AHL-B, a reference committee of four Byzantine nodes is also considered to establish



(a) 0% Cross-shard (b) 20% Cross-shard (c) 80% Cross-shard (d) 100% cross-shard

Figure 6.6: Increasing the percentage of cross-shard transactions (Byzantine nodes)

consensus on the order of cross-shard transactions. In the APR-B blockchain system, 4 nodes are used as the active replicas and finally, FaB uses 6 nodes ($5f + 1$) to establish consensus. Similar to the previous case, since APR-B and FaB do not use sharding, the percentage of cross-shard transactions does not affect their performance.

As shown in Figure 6.6(a), with no cross-shard transactions, SharPer is able to process more than 25000 transactions with 200 ms latency. As before, since for intra-shard transactions, AHL-B uses the same technique as SharPer, the results of SharPer and AHL-B are the same. APR-B and FaB also process 5900 and 6800 transactions (23% and 27% of SharPer) with 220 ms and 130 ms latency respectively. Note that since transactions are processed in two phases (instead of three), FaB has lower latency in comparison to APR-B.

Increasing the percentage of cross-shard transactions to 20%, reduces the peak throughput of SharPer to 18700 (with 240 ms latency). In this scenario and in comparison to AHL-B, SharPer is able to process 15% more transactions (at their respective peak throughput) because of the parallel ordering of cross-shard transactions and establishing cross-shard consensus in less number of phases. As mentioned before, since the sharding technique is not utilized by APR-B and FaB, the percentage of cross-shard transactions does not affect their performance. Note that with 20% cross-shard transactions, the peak throughput of SharPer is 320% and 270% of the peak throughput of APR-B and FaB

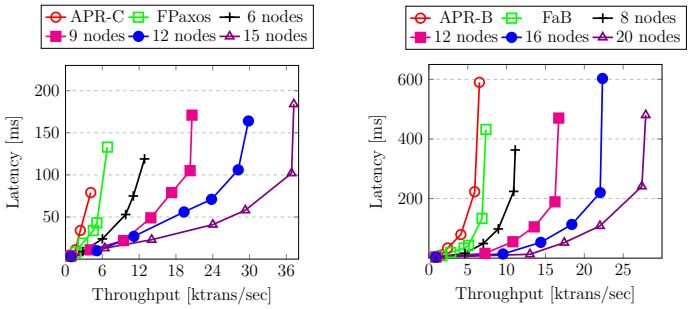
respectively.

With 80% cross-shard transactions, the peak throughput of SharPer reduces to 8600 which is still 34% higher than the peak throughput of AHL-B (6400) due to parallel processing of non-overlapping cross-shard transactions. Finally, when all transactions are cross-shard, SharPer is able to process 7500 transactions with 700 ms latency whereas AHL-B processes 5000 transactions (67% of SharPer) with the same latency. In the last two scenarios (80% and 100% cross-shard transactions), because of the high percentage of cross-shard transactions, using sharding techniques has no significant advantage and since APR-B and FaB rely on only four ($3f+1$) and six ($5f+1$) nodes to order transactions respectively, their latency is lower than SharPer. However, in SharPer, simultaneous processing of non-overlapping transactions results in improved throughput.

6.5.3 Performance with Different Number of Nodes

In the last set of experiments, we measure the performance of SharPer in networks with different number of nodes. We measure the performance of SharPer in a network including 6, 9, 12, and 15 crash-only nodes as well as 8, 12, 16 and 20 Byzantine nodes (2, 3, 4 and 5 clusters). The workloads also include 90% intra- and 10% cross-shard transactions (the typical settings in partitioned databases [60] [61]).

As can be seen in Figure 6.7(a), when nodes follow the crash failure model, by increasing the number of nodes (clusters) the throughput of the system increases almost linearly. This is expected because 90% of transactions are intra-shard transactions and, as shown earlier, for intra-shard transactions, the throughput of the entire system will increase linearly with the increasing number of clusters. In addition, since cross-shard transactions access two clusters, by increasing the number of clusters, the chance of parallel processing of such transactions increases. As shown in Figure 6.7(a), in the settings



(a) Crash-only nodes (b) Byzantine nodes

Figure 6.7: Increasing the number of nodes

with five clusters, SharPer is able to process 37000 transactions with 100 ms latency. Since increasing the number of nodes does not significantly affect the performance of APR-C and FPaxos systems, their performance will be similar to what is reported in Figure 6.5. However, as can be seen, in a network consisting of 6 nodes (50% more nodes than FPaxos) SharPer processes upto 11060 transactions (88% more than FPaxos) with the same (75 ms) latency.

Similarly, when nodes follow the Byzantine failure model, increasing the number of clusters enhances the overall throughput of SharPer, as shown in Figure 6.7(b). In this scenario, SharPer can process more than 27000 transactions with 240 ms latency on a network with five clusters. Furthermore, in a network with 8 nodes and with 200 ms latency, SharPer, using only 33% more nodes, is able to process 58% more transactions than FaB. This set of experiments demonstrates the scalability of SharPer as the number of clusters increases.

6.5.4 Discussion

Overall, the evaluation results can be summarized as follow.

First, in typical settings where workloads include low percentage (less than 20%) of cross-shard transactions, SharPer demonstrates better performance with both crash-only

and Byzantine nodes in comparison to other approaches. The performance of SharPer is better than AHL [86] because of the cross-shard consensus routine of SharPer that, in contrast to AHL, can order cross-shard transactions with non-overlapping clusters in parallel. The performance of SharPer is much (three to four times) better than both FPaxos (FaB) and active/passive replication (APR-C and APR-B) since SharPer uses the sharding technique and is able to process intra-shard transactions of different clusters in parallel whereas in both FPaxos (FaB) and active/passive replication, transactions are processed sequentially. Furthermore, and as shown in Figure 6.7, the performance of SharPer enhances semi-linearly with the increasing number of clusters, which clearly demonstrates the scalability of SharPer.

Second, in settings with high percentage of cross-shard transactions, using sharding techniques has no significant advantage. As a result, in the presence of extra nodes, using FPaxos (FaB) and active-passive replication (APR-C and APR-B) results in better performance (specially less latency). Note that as mentioned before, the typical settings in partitioned database systems includes only 10% cross-shard transactions [60] [61].

6.6 Summary

In this chapter, we proposed SharPer, a permissioned blockchain system which is designed specifically for networks with a very high percentage of non-faulty nodes ($N \gg 3f + 1$ for Byzantine or $N \gg 2f + 1$ for crash-only nodes). SharPer utilizes the extra resources by partitioning the nodes into clusters of $3f+1$ Byzantine (or $2f+1$ crash-only) nodes and processing the transactions on different clusters in parallel. The blockchain ledger in SharPer is formed as a directed acyclic graph which in not maintained by any node. Nodes of each cluster indeed maintain a view of the blockchain ledger including the intra-shard transactions of the cluster as well as the cross-shard transactions that

the cluster is involved in. A flattened consensus protocol is also introduced to order cross-shard transactions without relying on an extra set of nodes or trusted participants. Furthermore, SharPer is able to process cross-shard transactions with non-overlapping clusters in parallel. Our experiments show that in workloads with low percentage of cross-shard transactions (typical settings), SharPer demonstrates better performance with both crash-only and Byzantine nodes in comparison to other approaches and the throughput of SharPer will increase semi-linearly by increasing the number of clusters.

Chapter 7

SeeMoRe: On Fault Tolerance of Permissioned Blockchains

7.1 Introduction

Permissioned blockchain rely on consensus protocols to provide fault tolerance. Fault-tolerant protocols are the main building block of permissioned blockchain systems and have also been extensively used in the distributed database infrastructure of large enterprises such as such as Google's Spanner [35], Amazon's Dynamo [36], and Facebook's Tao [37], thus highlighting the critical role of SMR in data management. While today's enterprises mostly rely on cloud storage to run their business applications and benefits from its scalability, and easy access of cloud storage [96], storing data on a single cloud may reduce robustness and performance [97] [98] [99]. Robustness is the ability to ensure availability (liveness) and one-copy semantics (safety) despite failures, while performance deals with response time (latency) and the number of processed requests per time unit (throughput) [42]. Fault-tolerant protocols are designed to satisfy both robustness and performance concerns using State Machine Replication (SMR) [50] techniques.

Large scale data management systems utilize SMR to provide fault tolerance and to increase the performance of the system. While large enterprises might have their own Geo-replicated fault-tolerant cloud storage around the world, smaller enterprises may only have a local private cloud that is lacking in resources to guarantee fault tolerance. One solution is to store all the data on third-party public cloud providers [100] [101] [102]. Public clouds provide several advantages like elasticity and durability, but they often suffer from security concerns, e.g., malicious attacks [103]. Private clouds, on the other hand, may not provide sufficient elasticity and durability, however, they are more secure. The trustworthiness of a private cloud allows an enterprise to build services that can utilize crash fault-tolerant protocols, i.e., protocols that make progress when a bounded number of replicas only fail in a benign manner, for example by either crashing or being unresponsive. But due to lack of private resources, if a third-party public cloud is used, the nodes of the public cloud *may* behave maliciously, in which case a more robust fault-tolerant protocol is needed that allows the system to continue operating correctly, even when some replicas exhibit arbitrary, possibly malicious behavior. Current Byzantine fault-tolerant protocols (e.g., PBFT [21]) introduce significant communication and latency overheads in order to tolerate failures since they consider all failures as malicious.

An alternative solution to storing all the data in a public cloud is to use a hybrid cloud storage system consisting of both private and public clouds [98]. In a hybrid cloud, the nodes in the private cloud are trusted and may crash but do not behave maliciously whereas the nodes in the public cloud(s) might be malicious. Hybrid clouds address the *security* concerns of using *only* public clouds by giving enterprises the ability to still use their private clouds with their trusted, non-malicious nodes. In addition, storing data on multiple clouds is more *reliable*, e.g., if a cloud outage happens, the system might still process requests. Moreover, while a small private cloud may represent a *scalability*

bottleneck, the system can rent as many servers as required from public clouds. The benefits of hybrid clouds necessitate designing new protocols that can leverage the trust of private clouds and the scalability of public clouds.

Despite years of intensive research, existing fault-tolerant protocols do not adequately address all the characteristics of hybrid environments. On one hand, the existing Byzantine fault-tolerant protocols [21] [38] [39] [40] [41] [42] [43] do not distinguish between crash and malicious failures, and consider all failures as malicious, thus incurring a higher cost in terms of performance. On the other hand, the hybrid protocols [44] [45] that have been designed to tolerate both crash and malicious failures, make no assumption on *where* the crash or malicious failures may occur. As a result, using these protocols in a hybrid environment (either cloud or blockchain), where all machines in the private environment are known to be trusted while machines in the public environment could be compromised and hence malicious, results in an unnecessary performance overhead.

In this chapter, we present *SeeMoRe*[1]: a State Machine Replication protocol that leverages the localization of crash and malicious failures in a *hybrid environment*. SeeMoRe considers a private environment consisting of trusted replicas, a subset of which may fail-stop, and a public environment where a subset of the replicas may behave maliciously. SeeMoRe takes explicit advantage of this knowledge to improve performance by reducing the number of communication phases and messages exchanged and/or the number of required replicas. SeeMoRe has three different modes of operation which can be used depending on the load on the private environment, and the latency between the public and the private environment. We also introduce a dynamic technique to transition from one mode to another.

A key contribution of this chapter is to show how being aware of *where* different types

[1]SeeMoRe is derived from Seemorq, a benevolent, mythical bird in Persian mythology which appears as a peacock with the head of a dog and the claws of a lion. Seemorq in Persian literature also refers to a group of birds who flew together to achieve a common goal.

of failures (crash and malicious) may occur in hybrid environments, results in designing more efficient protocols. In particular, this chapter makes the following contributions:

- A model for hybrid environments is presented which can be used by enterprises that do not have enough servers in their trusted private environment, e.g., cloud to run fault-tolerant protocols and gives them the option of renting from untrusted public environment.
- SeeMoRe, a hybrid protocol that tolerates both crash and malicious failures, is developed in three different modes. Being aware of where the crash faults may occur and where the malicious faults can occur results in reducing the number of communication phases, messages exchanged and/or required replicas. In addition, a technique to dynamically switch between different modes of SeeMoRe is presented.

The rest of this chapter is organized as follows. The system model is introduced in Section 7.2. Section 7.3 presents a method to compute the required number of replicas from a public cloud. The design of SeeMoRe is proposed in Section 7.4. Section 7.5 shows the performance evaluation, and Section 7.6 concludes the chapter.

7.2 System Model

In this section, we introduce the system model wherein an application layer, such as a distributed database management system, relies on a replication service to store copies of data across an environment consisting of private and public clouds. The replication service aims to replicate the data across some trusted and some untrusted servers. The replicas can be geo-distributed to provide low data access latency to clients across the globe or they can be geographically confined to tolerate both crash or malicious failures. Such a replication service can use SeeMoRe and we specify the assumptions on which

SeeMoRe is built in this section.

7.2.1 Basic Assumptions

We consider a hybrid failure model that admits both crash and malicious failures in a hybrid environment, e.g., cloud, where crash failures may occur in the private cloud and malicious failures may *only* occur in the public cloud. Note that a malicious failure can encompass a crash failure but since the trust assumptions are low, we do not distinguish between a crash or a malicious failure in the public cloud. This is indeed a realistic assumption as the private cloud is hosted locally where the client resides, and hence under their control, while the public cloud is managed externally by public cloud providers. We call the nodes in the private cloud *trusted* and the nodes in the public cloud *untrusted*. The model puts no restrictions on clients, except that their numbers must be finite, however, safety and liveness require some constraints on the number of faulty servers.

7.2.2 Quorum and Network Size

We consider a cloud environment consisting of private and public clouds. The system is an asynchronous distributed system containing a set of N (N=S+P) replicas where S of them are in a private and P of them are in a public cloud. The private cloud consists of non-malicious (either non-faulty or crashed) nodes, whereas nodes in the public cloud can be either non-faulty nodes or Byzantine nodes. The bound on the maximum number of crashed nodes in the private cloud and malicious nodes in the public cloud is assumed to be c and m respectively. We call the nodes in the private cloud *trusted* and the nodes in the public cloud *untrusted*. All the clients and the replicas know which replicas are trusted and which are untrusted.

Failures are divided into two disjoint classes: malicious and crash failures. In crash

fault-tolerant models, e.g., Paxos [20], given that c nodes can crash, a request is replicated to a quorum consisting of at least $c+1$ nodes to provide fault tolerance and to guarantee that a value once decided will remain decided in spite of failures (safety). Furthermore, any two quorums intersect on at least *one* node and as a result, $2c+1$ is the minimum number of nodes that allows an asynchronous system to provide the safety property.

In the Byzantine failure models, e.g., PBFT [21], given that m nodes can be malicious, the quorum size should be at least $2m+1$ to ensure that non-faulty replicas outnumber the malicious ones, i.e., a request is replicated in enough non-faulty nodes to guarantee safety in the presence of m failures. This implies that any two quorums intersect with at least $m+1$ nodes to ensure one correct node in the intersection, thus the minimum network size is $3m+1$ [55].

Likewise, in the hybrid model, to tolerate c crash and m malicious failures, the quorum size must include at least $2m+c+1$ nodes [45]. This also guarantees that the intersection of any two quorums includes at least $m+1$ nodes. Since the quorum size is $2m+c+1$ and the intersection of any two quorum Q and Q' is $m+1$ nodes, $|Q|+|Q'| = N+m+1 = 4m+2c+2$, thus, as shown in [45], the (minimum) network size, N, is

$$N = 3m + 2c + 1. \tag{7.1}$$

Intuitively, if there are f failures (of any type) in a network, the network size has to be at least f larger than the quorum size, as any network with smaller size could lead to a deadlock situation where none of the f faulty servers are participating. Since, $f = m+c$ and the quorum size Q is $2m+c+1$, the network size should be at least $Q+f$ i.e., $3m+2c+1$.

7.3 Public Cloud

The hybrid failure model presented in Section 7.2 can be used by enterprises that own private clouds with a limited number of trusted servers which is *insufficient* to run a fault-tolerant protocol. This model gives them the option of renting from untrusted public clouds. In this section, we present two methods to identify the number of servers an enterprise needs to rent from a public cloud.

A business that owns an insufficient number of trusted (crash-only) servers needs to rent more servers from some untrusted public clouds to satisfy the minimum network size constraints ($3m + 2c + 1$). Public clouds might provide some statistics that show the percentage of faulty nodes in the cloud. If there is no information on the type of failures, i.e. crash or malicious, within the public cloud, we consider all the faulty nodes as malicious and We assume that the ratio of malicious nodes in public cloud (m) to the size of public cloud ($\mathcal{P}$) is known and is equal to $\alpha = \frac{m}{\mathcal{P}}$. Note that, we assume a uniform distribution of malicious nodes in public cloud, i.e., in any set $\pi \subseteq \mathcal{P}$, at most $\alpha \times \pi$ nodes are malicious.

Given the size of the private cloud S, the bound on the maximum number of crashed nodes c in the private cloud, and the ratio α of malicious nodes (m) in the public cloud to the size of the public cloud ($\mathcal{P}$), the task is to identify the required number of nodes P to be rented from the public cloud that allows satisfying the protocol constraints.

The total number of nodes in the network is $N = S + P$. Given our assumption of α, we get $m = \alpha P$. Replacing m in Equation 7.1, we get $N = 3\alpha P + 2c + 1$, which means, $(3\alpha - 1)P = S - (2c + 1)$, thus:

$$P = \left\lceil \frac{S - (2c + 1)}{3\alpha - 1} \right\rceil \tag{7.2}$$

As an example consider the situation that a private cloud has 2 servers where one of them might be faulty, i.e., $S = 2$, and $c = 1$, and we want to rent servers from a public cloud with $\alpha = 0.3$. Here, $P = \frac{2-2-1}{3*0.3-1} = \frac{-1}{-0.1} = 10$, which means we need to rent 10 servers from the public cloud to provide the safety constraints of the replication protocol.

In Equation 7.2, if the size of the private cloud (S) is equal or greater than $2c + 1$, then the private cloud does not need to rent any nodes and can run a crash fault-tolerant protocol like Paxos [20] by itself. If there is no private cloud ($S = 0$) or all the nodes in the private cloud are faulty ($S = c$), using the private cloud has no advantage and it is more reasonable to rent all the required nodes from the public cloud and run a Byzantine fault-tolerant protocol in the public cloud. However, if $c < S < 2c + 1$, renting nodes from a public cloud and running SeeMoRe will result in better performance.

Similarly, if $\alpha \geq 1/3$, (i.e., more than one-third of the nodes in the public cloud are malicious), then the public cloud cannot satisfy the network size constraint for Byzantine fault-tolerance. Hence, an enterprise will need to rent servers if its private cloud size, S, is between $c + 1$ and $2c$, and it can rent servers from public cloud providers that satisfy $\alpha < 1/3$. It should be noted that even if the size of the private cloud is equal or greater than $2c + 1$, and the public cloud does not satisfy the $\alpha<1/3$ constraint, an enterprise might still rent some replicas from the public cloud for load balancing purposes.

Note that Equation 7.2 can easily be extended to address the situation where the public cloud provides information on the ratio of both malicious and crash nodes, i.e., the ratio of malicious nodes to the size of public cloud ($\alpha = \frac{m}{\mathcal{P}}$) as well as the ratio of crash nodes to the size of public cloud ($\beta = \frac{c}{\mathcal{P}}$) are known. In such a situation, Equation 7.2 can be rewritten as:

$$P = \lceil \frac{S - (2c + 1)}{3\alpha + 2\beta - 1} \rceil \tag{7.3}$$

This method, which identifies the required number of nodes from a public cloud,

assumes a uniform distribution of faulty nodes in the public cloud. However, public clouds might not guarantee a uniform distribution of α and alternatively specify the maximum number of concurrent failures in a cluster of rental nodes explicitly. In such a setting, even if an enterprise rents a portion of that cluster, there is no guarantee that the percentage of the faulty nodes in that portion is equal to the percentage of the faulty nodes in the entire cluster. For example, a public cloud might guarantee that in a cluster of 10 nodes, at most two concurrent failures can occur. Nonetheless, if an enterprise rents only two nodes from that cluster, both of them might fail at the same time. Assuming that the number of concurrent malicious failures in a (cluster of nodes in a) public cloud is given and equal to M, we would want to identify the required number of nodes P to rent from such a public cloud. The total number of nodes in the network is $N = 3m + 2c + 1 = S + P$ and there is no guarantee on a uniform distribution of malicious nodes in the public cloud, thus $m = M$. Hence, the required number of nodes is $P = (3M + 2c + 1) - S$.

Similar to the first method, if the public cloud distinguishes between different types of failures and provides information on the number of both crash and malicious failures, given as C and M, the required number of nodes from the public cloud is $P = (3M + 2C + 2c + 1) - S$ where c, similar as before, is the number of crash failures in the private cloud.

Finally, it should be noted that both methods of identifying the public cloud size can be generalized to multiple public clouds as well. In Such a settings, since different public clouds might have different ratio (number) of faulty nodes, the equation might have multiple solutions.

7.4 SeeMoRe

In this section, we present SeeMoRe, a hybrid fault-tolerant protocol for a public/private cloud environment that tolerates m malicious failures in the public and c crash failures in the private cloud.

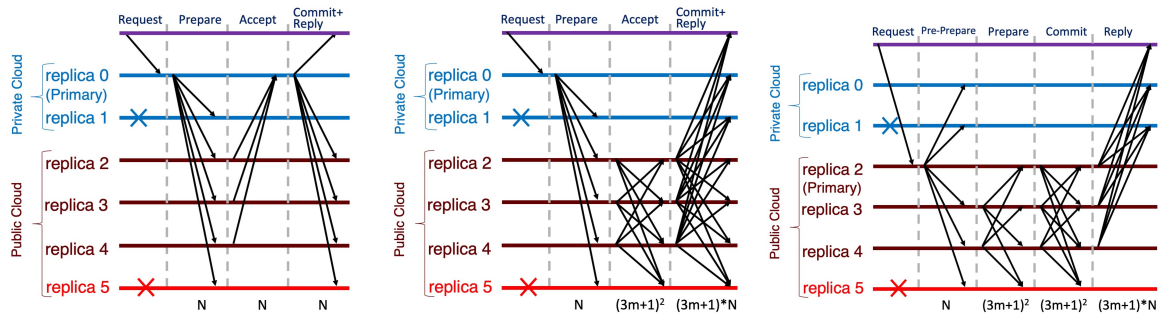
SeeMoRe is inspired by the known Byzantine fault-tolerant protocol PBFT [41] and consists of *agreement* and *view change* routines where the agreement routine orders requests for execution by the replicas, and the view change routine coordinates the election of a new primary when the current primary is faulty.

Based on the impossibility (FLP) result [53], SeeMoRe, similar to most fault-tolerant protocols, ensures the safety property without any synchrony assumption, however, a weak synchrony assumption is needed to satisfy liveness.

We identify each replica using an integer in $[0, ..., N-1]$ where replicas in the private cloud, i.e., trusted replicas, have identifiers in $[0, ..., S-1]$ and replicas in the public cloud, i.e., untrusted replicas, are identified using integers in $[S, ..., N-1]$.

In SeeMoRe, the replicas move through a succession of configurations called *views* [94] [95]. In a view, one replica is *the primary* and the others are *backups*. Depending on the mode, some backups are *passive* and do not participate in the agreement. Views are numbered consecutively. All replicas are initially in view 0 and are aware of their current view number.

We explain SeeMoRe in three different modes: *Trusted Primary, Centralized Coordination (TPCC)*, *Trusted Primary, Decentralized Coordination (TPDC)*, and *Untrusted Primary, Decentralized Coordination (UPDC)*. In the first mode, TPCC, the primary is *always* in the private cloud, thus the primary is non-malicious. The second mode, TPDC, is used to reduce the load on the private cloud by assuming that the primary is still in the private cloud, but instead of processing the client requests itself, depends on



(a) The TPCC Mode (b) The TPDC Mode (c) The UPDC Mode

Figure 7.1: The normal case operation of the three modes of SeeMoRe

$3m + 1$ nodes in the public cloud to process the requests. This mode reduces the load on the private cloud, because except for the primary, which does a single broadcast of the client's request, other replicas in the private cloud are passive and do not participate in any phases. Finally, in the third mode, UPDC, an untrusted node is chosen as the primary and the protocol relies completely on the public cloud to process requests. This mode is useful when we intentionally rely completely on the public cloud for two purposes: (1) load balancing when all the nodes in the private cloud are heavily loaded, or (2) reducing the delay when there is a large network distance between the private and the public cloud and the latency of having one more phase of communication within the public cloud is less than the latency of exchanging messages between the two clouds. The agreement routine of the UPDC mode is the same as PBFT [41], however, the view change routine can be more efficient.

In this section, we describe each of these three modes in detail, followed by a technique to dynamically switch between the modes. We use π to show the current mode of the protocol where $\pi \in \{1, 2, 3\}$ and 1, 2, and 3 are the TPCC, TPDC, and UPDC modes respectively. We also present a short discussion on the different modes of SeeMoRe and compare them with some known relevant protocols.

7.4.1 TPCC Mode: Trusted Primary, Centralized Coordination

Owning a private cloud gives SeeMoRe the chance to choose a trusted node as the primary. When the primary is trusted, all the non-faulty backups receive correct messages from the primary, which eliminates the need to multicast messages by replicas to realize whether all the non-faulty ones receive the same message or not. Thus, we can reduce one phase of communication and a large number of messages.

Figure 7.1(a) shows the normal case operation of the TPCC mode. Here, replicas 0 and 1 are trusted ($S = 2$) and the four other replicas, 2 to 5, are untrusted ($P = 4$). In addition, one of the trusted replicas (1) is crashed ($c = 1$) and one of the untrusted replicas (5) is malicious ($m = 1$). With a trusted primary, the total number of exchanged messages is $3N$.

The pseudo-code for the TPCC mode is presented in Algorithm 13. Although not explicitly mentioned, every sent and received message is logged by the replicas. Each replica is initialized with a set of variables as indicated in lines 1-4 of the algorithm. The primary of view v is a replica p such that $p = (v \mod S)$. A client ς requests a state machine operation op by sending a message $\langle \textsf{REQUEST}, op, ts_\varsigma, \varsigma \rangle_{\sigma_\varsigma}$ to replica p it believes to be the primary. The client's timestamp ts_ς is used to totally order the requests and to ensure exactly-once semantics. The client also signs the message with signature σ_ς for authentication.

As indicated in lines 5-8, upon receiving a client request, the primary p first checks if the signature and timestamp in the request are valid and simply discards the message otherwise. The primary assigns a sequence number n to the request and multicasts a signed $\langle \langle \textsf{PREPARE}, v, n, d \rangle_{\sigma_p}, \mu \rangle$ message to all the replicas where v is the current view, μ is the client's request message, and d is the digest of μ. At the same time, the primary

Algorithm 13 The Normal-Case Operation in the *TPCC* mode

```
1: init():
2:    r := replicaId
3:    v := viewNumber
4:    if r = (v mod S) then isPrimary := true

5: upon receiving μ=⟨REQUEST, op, ts_ς, ς⟩_σς and isPrimary:
6:    if μ is valid then
7:       assign sequence number n
8:       send ⟨⟨PREPARE, v, n, d⟩_σp, μ⟩ to all replicas

9: upon receiving ⟨⟨PREPARE, v, n, d⟩_σp, μ⟩ from primary p:
10:   if v is valid then
11:      send ⟨ACCEPT, v, n, d, r⟩ to primary p

12: upon receiving ⟨ACCEPT, v, n, d, r⟩ from 2m+c replicas and isPrimary:
13:   send ⟨⟨COMMIT, v, n, d⟩_σp, μ⟩ to all replicas
14:   execute operation op
15:   send ⟨REPLY, π, v, ts_ς, u⟩_σp to client ς with result u
```

appends the message to its log. The primary signs its message, because it might be used by other replicas later in view changes as a proof of receiving the message.

As shown in lines 9-11 of the algorithm, upon receipt of $\langle\langle \textsf{PREPARE}, v, n, d\rangle_{\sigma_p}, \mu\rangle$ from primary p, replica r checks if view v is equal to the replica's view. It then logs the prepare message, and responds to the primary with $\langle \textsf{ACCEPT}, v, n, d, r\rangle$ message. Since accept messages are sent only to the trusted primary and are not used later for any other purposes, there is no need to sign these messages.

Upon collecting $2m+c$ valid accept messages from different replicas (plus itself becomes $2m+c+1$) for the request μ in view v with sequence number n, as seen in lines 12-15, the primary multicasts a commit message $\langle\langle \textsf{COMMIT}, v, n, d\rangle_{\sigma_p}, \mu\rangle$ to all replicas. The primary attaches the request μ to its commit message, so that if a replica has not received a prepare message for that request, it can still execute the request. The primary also executes the operation *op* and sends a reply message $\langle \textsf{REPLY}, \pi, v, ts_\varsigma, u\rangle_{\sigma_p}$ to client ς. Mode number π and view number v are sent to clients to enable them to track the current mode and view and hence the current primary. It is important especially when a mode change or view change occurs, replacing the primary.

Once a replica receives a valid commit message with correct view number from the

primary, it executes the operation *op*, if all requests with lower sequence numbers than n has been executed. This ensures that all non-malicious replicas execute requests in the same order as required to provide the safety property. Note that even if the replica has not received a prepare message for that request, as long as the view number is valid and the message comes from the primary, the replica considers the request as committed.

When the client receives a reply message $\langle \text{REPLY}, \pi, v, ts_\varsigma, u \rangle_{\sigma_p}$ with a valid signature from primary p and with the same timestamp as the client's request, it accepts u as the result of the requested operation. If the client does not receive a reply from the primary after a preset time, the client may suspect a crashed primary. The client then broadcasts the same request to all replicas. A replica, upon receiving the client's request, checks if it has already executed the request; if so, it simply sends the reply message to the client. The client waits for a reply from the private cloud or $m+1$ matching reply messages from the public cloud before accepting the result. The primary will eventually be suspected to be faulty by enough replicas to trigger a view change.

State Transfer. A fault-tolerant protocol must provide a way to checkpoint the state of different replicas. It is especially required in an asynchronous system where even non-faulty replicas can fall arbitrarily behind. Checkpointing also brings slow replicas up to date so that they may execute more recent requests. Similar to [21], in our protocol, checkpoints are generated periodically when a request sequence number is divisible by some constant (checkpoint period).

Trusted primary p produces a checkpoint and multicasts a $\langle \text{CHECKPOINT}, n, d \rangle_{\sigma_p}$ message to the other replicas, where n is the sequence number of the last executed request and d is the digest of the state. A server considers a checkpoint to be *stable* when it receives a checkpoint message for sequence number n signed by trusted primary p. We call this message a *checkpoint certificate*, which proves that the replica's state was correct until

that request execution.

View Changes. The goal of the view change routine is to provide liveness by allowing the system to make progress when a primary fails. It prevents replicas from waiting indefinitely for requests to execute. A view change must guarantee that it will not introduce any changes in a history that has been already completed at a correct client. Most view change routines [95] [94] [21] [39] [104] [105] [106] are triggered by timeouts and require enough non-faulty replicas to exchange view change messages. SeeMoRe uses a similar technique in the TPCC mode. In such a situation, replicas detect the failure and reach agreement to change the view from v to v'. The primary of new view v' then handles the uncommitted requests, and takes care of the new client requests.

View changes are triggered by timeout. When a replica receives a valid prepare message from the primary, it starts a timer that expires after some defined time τ. When the backup receives a valid commit message, the timer is stopped, but if at that point the backup is waiting for a commit message for some other request, it restarts the timer. If the timer of a replica r for some prepare message expires, the backup suspects that the primary is faulty, it stops accepting prepare and commit messages and multicasts a $\langle \text{VIEW-CHANGE}, v{+}1, n, \xi, \mathcal{P}, \mathcal{C} \rangle$ message to all replicas where n is the sequence number of the last stable checkpoint known to r, ξ is the checkpoint certificate, and $\mathcal{P}$ and $\mathcal{C}$ are two sets of received valid prepare (without the request message μ) and commit messages for requests with a sequence number higher than n. When primary p' of new view $v+1$ receives $2m+c$ valid view-change messages from different replicas, it multicasts a $\langle \text{NEW-VIEW}, v+1, \mathcal{P}', \mathcal{C}' \rangle_{\sigma_{p'}}$ message to all replicas where $\mathcal{P}'$ and $\mathcal{C}'$ are two sets of prepare and commit messages respectively which are constructed as follows.

Let l be the sequence number of the latest checkpoint, and h be the highest sequence number of a prepare message in all the received $\mathcal{P}$ sets. For each sequence number n where

$l < n \leq h$, the primary does the following steps:

1) It checks all commit messages in set $\mathcal{C}$ of the view-change messages. If the primary finds a commit message with a valid signature σ_p (p was the primary of view v) for request μ, the primary adds a $\langle\langle \text{COMMIT}, v+1, n, d\rangle_{\sigma_{p'}}, \mu\rangle$ to $\mathcal{C}'$

2) If no such commit message is found, the primary checks the prepare messages in $\mathcal{P}$ sets:

• If the primary finds $2m + c + 1$ valid prepare messages for n, it adds a $\langle\langle \text{COMMIT}, v+1, n, d\rangle_{\sigma_{p'}}, \mu\rangle$ to $\mathcal{C}'$.

• Else, if it receives at least one valid prepare message for n, the primary adds a $\langle\langle \text{PREPARE}, v+1, n, d\rangle_{\sigma_{p'}}, \mu\rangle$ to $\mathcal{P}'$.

3) If none of the above situations occur, there is no valid request for n, so the primary adds a $\langle \text{PREPARE}, v+1, n, d\rangle_{\sigma_{p'}}, \mu^{\emptyset}\rangle$ to $\mathcal{P}'$ where $\mu^{\emptyset}$ is a special *no-op* command that is transmitted by the protocol like other requests but leaves the state unchanged. The third situation happens when no replica has received a prepare message from the previous primary.

In contrast to PBFT, since the primary is trusted, it does not need to append all the view-change messages in the new-view message which makes the new-view messages much smaller. The primary inserts all the messages in $\mathcal{P}'$ and $\mathcal{C}'$ to its log. It also checks the log to make sure its log contains the latest stable checkpoint. If not, the primary inserts checkpoint messages for the checkpoint l and discards the earlier information from the log.

Once a replica in view v receives a new-view message from the primary of view $v+1$, the replica logs all prepare and commit messages, updates its checkpoint in the same way as the primary, and for each prepare message, sends an accept message to the primary. Non-faulty replicas in view v will not accept a prepare message for a new view $v' > v$ without having received a new-view message for v'.

Correctness. Within a view, since the primary is trusted and it assigns sequence numbers to the requests, safety is ensured as long as the primary does not fail. Indeed, for any two committed requests r_1 and r_2 with sequence numbers n_1 and n_2 respectively, if $D(r_1) = D(r_2)$, then $n = n'$.

If the primary fails a view change is executed. To ensure safety across views, the primary waits for $2m + c$ accept messages (considering itself, a quorum of $2m + c + 1$) from different replicas to ensure that committed requests are totally ordered across views. In fact, for any two committed requests r_1 and r_2 with sequence numbers n_1 and n_2, since a quorum of $2m+c+1$ replicas commits r_1 and a quorum of $2m+c+1$ replicas commits r_2, and these two quorums have at least $m + 1$ overlapping nodes, there should be at least one non-faulty node that commits both r_1 and r_2 but this is not possible because the node is not faulty. As a result, if $D(r_1) = D(r_2)$, then $n = n'$. This guarantees that in the event of primary failure, any new quorum of $2m + c + 1$ replicas will have at least $m + 1$ overlapping nodes that received a prepare message (and sent accept) for request μ from the previous primary. Thus, there is at least one non-faulty node in that quorum that helps the protocol to process request μ in the new view.

7.4.2 TPDC Mode: Trusted Primary, Decentralized Coordination

The TPDC mode is proposed to reduce the load on the private cloud. In this mode, a *trusted primary* receives a request message, assigns a sequence number, and relies on $3m + 1$ *untrusted nodes* (in the public cloud) to process the request. These $3m + 1$ nodes are called *proxies*. Since a trusted primary assigns the sequence number to the request before broadcasting, this reduces the scope of any malicious behaviour. Whereas in PBFT, when replicas receive a message from the primary, they perform one round

of communication to make sure all non-faulty replicas agree on a total order for the requests within a view. However, here, since a trusted primary assigns the sequence numbers, similar to the TPCC mode, there is no need for that phase.

Figure 7.1(b) shows the normal case operation of SeeMoRe with a trusted primary (node 0). As before, two replicas are trusted ($S = 2$), four replicas are untrusted ($P = 4$), $c = 1$, and $m = 1$. Since a trusted primary assigns sequence numbers, the protocol, similar to Paxos, needs two phases to process requests. However, since the protocol tolerates malicious failures, the number of messages in terms of the number of replicas, similar to PBFT, is quadratic. Here, there are totally $N + (3m + 1)^2 + (3m + 1) * N$ messages exchanged where $3m + 1$ is the total number of proxies. In this example, since $m = 1$, all replicas in the public cloud are proxies.

Algorithm 14 provides the pseudo-code for the TPDC mode. Lines 1-5 indicate the initialization of state variables for the primary and proxies. A replica r in the public cloud is a *proxy* in view v if $r-(v \mod P) \in [S, ..., S+3m]$. Here since replicas are in the public cloud, r is an integer in $[S, ..., N-1]$. The public cloud might have more than $3m+1$ replicas, however, $3m+1$ is enough to establish consensus and any additional replicas may degrade the performance. The trusted primary of view v is chosen in the same way as the first mode, i.e., p is the primary if $p=(v \mod S)$.

As shown in lines 6-9 of the algorithm, the primary, upon receiving request μ, validates the timestamp and signature of μ, assigns a sequence number n, and multicasts signed prepare message $\langle \langle \textsf{PREPARE}, v, n, d \rangle_{\sigma_p}, \mu \rangle$ to all replicas.

When a proxy receives a prepare message from the primary, as indicated in lines 10-12, it validates the view number, logs the message and sends a signed accept message $\langle \textsf{ACCEPT}, v, n, d, r \rangle_{\sigma_r}$ to all the other proxies. Here, in contrast to the first mode, the proxy signs its message as a proof of message reception in case of a view change.

As described in lines 13-17 of the algorithm, upon receiving $2m + 1$ matching accept

Algorithm 14 The Normal-Case Operation in the *TPDC* mode

1: **init():**
2: $r :=$ replicaId
3: $v :=$ viewNumber
4: if $r = (v \mod S)$ then isPrimary := true
5: else if $r - (v \mod P) \in [S, .., S + 3m]$ then isProxy := true

6: **upon receiving** $\mu = \langle \text{REQUEST}, op, ts_\varsigma, \varsigma \rangle_{\sigma_\varsigma}$ and isPrimary:
7: if μ is *valid* then
8: **assign** *sequence number* n
9: **send** $\langle \langle \text{PREPARE}, v, n, d \rangle_{\sigma_p}, \mu \rangle$ to all replicas

10: **upon receiving** $\langle \langle \text{PREPARE}, v, n, d \rangle_{\sigma_p}, \mu \rangle$ from the primary p and isProxy:
11: if v is *valid* then
12: **send** $\langle \text{ACCEPT}, v, n, d, r \rangle_{\sigma_r}$ to all proxies

13: **upon receiving** $\langle \text{ACCEPT}, v, n, d, r \rangle$ from *2m+1* proxies:
14: **send** $\langle \text{COMMIT}, v, n, d, r \rangle_{\sigma_r}$ to all other proxies
15: **send** $\langle \text{INFORM}, v, n, d, r \rangle_{\sigma_r}$ to all private cloud nodes and non-proxy nodes in public cloud
16: **execute** operation *op*
17: **send** $\langle \text{REPLY}, \pi, v, ts_\varsigma, u \rangle_{\sigma_r}$ to client ς with result u

messages (including its own message) with correct signatures, a proxy r multicasts a commit message $\langle \text{COMMIT}, v, n, d, r \rangle_{\sigma_r}$ to the other proxies. Each proxy r also sends a signed inform message $\langle \text{INFORM}, v, n, r, d \rangle_{\sigma_r}$ to all the nodes in the private cloud and all non-proxy nodes in the public cloud. Non-proxy nodes wait for $2m + 1$ valid matching inform messages from different proxies which are matched by the prepare message that they received from the primary before executing the request. If a proxy has executed all requests with sequence numbers lower than n, it executes the request n and sends a reply message $\langle \text{REPLY}, \pi, v, ts_\varsigma, u \rangle_{\sigma_r}$ to the client.

Any other replica that receives $m + 1$ matching commit messages from the proxies with valid signatures, correct message digest, and view numbers equal to its view number considers the request as committed, and executes the request. Since all the replicas receive prepare messages from the primary, they have access to the request and can execute it.

The client also waits for $m + 1$ matching reply messages from different proxies before accepting the result. If the client has not received a valid reply after a preset time, the client multicasts the request to the proxies. The proxies re-send the result if the request has already been processed and the client waits for $m + 1$ matching reply messages from

the proxies before accepting the result. Otherwise, similar to the first mode, eventually the primary will be suspected to be faulty by enough replicas and a view change will be triggered.

State Transfer. Checkpointing in the TPDC mode works in the same way as the TPCC mode. Trusted primary p multicasts a signed checkpoint message to all other replicas with the sequence number of the last executed request and the digest of the state. Upon receiving a checkpoint message from the primary, a server considers that a checkpoint is *stable* and logs the message which is used later as a *checkpoint certificate*.

View Changes. In the TPDC mode, similar to the TPCC mode, the primary of new view handles the view change, however, only nodes in the public cloud send view-change messages. The view-change messages $\langle \text{VIEW-CHANGE}, v+1, n, \xi, \mathcal{P} \rangle$ are sent to all the nodes in the public cloud and the primary of the next view where ξ is the checkpoint certificate for sequence number n, and $\mathcal{P}$ is the set of received valid prepare messages with a sequence number higher than n.

Primary p' of the new view waits for $2m+1$ valid view-change messages from the proxies of the last active view, i.e., the view with a non-faulty primary, and multicasts a new-view message $\langle \text{NEW-VIEW}, v+1, \mathcal{P}' \rangle_{\sigma_{p'}}$ to all the replicas where for each sequence number n (between the latest checkpoint and the highest sequence number of a prepare message), if there is any valid prepare message in set $\mathcal{P}$ of the received view-change messages, the primary adds a $\langle \text{PREPARE}, v+1, n, d \rangle_{\sigma_{p'}}$ to $\mathcal{P}'$. Else, there is no valid request for n, so similar to the TPCC mode, the primary adds a no-op prepare message $\langle \text{PREPARE}, v+1, n, d \rangle_{\sigma_{p'}}, \mu^{\emptyset} \rangle$ to $\mathcal{P}'$.

Here, again, since the primary is trusted it does not need to include view-change messages in the new-view message. The primary then inserts all the messages in $\mathcal{P}'$ to its log and updates its checkpoint, if needed.

Once a proxy of view $v + 1$ receives a new-view message from the primary of view $v + 1$, the proxy logs all prepare messages, updates its checkpoint, and multicasts an accept message to all the proxies for each prepare message in $\mathcal{P}'$. Other replicas also receive the new-view message to be informed that the view is changed.

Correctness. Within a view, since the primary is trusted and it assigns sequence number to the requests, similar to the TPCC mode, safety is ensured as long as the primary does not fail. To ensure safety across views, since $3m + 1$ nodes participate in the protocol, to commit a message, $2m + 1$ matching accept messages are needed. In fact, for any two committed requests r_1 and r_2 with sequence numbers n_1 and n_2, since a quorum of $3m+1$ replicas commits r_1 and a quorum of $3m + 1$ replicas commits r_2, and these two quorums have at least $m+1$ overlapping nodes, there is at least one non-faulty node that commits both r_1 and r_2. But this is not possible because the replica is non-faulty. As a result, if $D(r_1)=D(r_2)$, then $n=n'$.

7.4.3 UPDC Mode: Untrusted Primary, Decentralized Coordination

One characteristic of online services is the ever changing patterns in client requests. While there might be periods of high traffic thus overloading some servers, at other periods, the resources may be underutilized. Also, depending on server placements and communication delays, enterprises may benefit from protocols that allow a subset of the servers, e.g. only the public cloud, to handle certain client requests.

The third mode of the protocol, the UPDC mode, is presented to handle two different situations. First, when the private cloud is heavily loaded and the public cloud can handle the requests by itself for load balancing. Second, when there is a large network distance between the private and the public cloud and the latency due to one more phase is less

than the latency of exchanging messages between the two clouds. In both situations, the nodes in the private cloud become passive replicas in the agreement routine and are only informed about the committed messages. However, they still may participate in the view change routine.

In the UPDC mode, SeeMoRe completely relies on $3m + 1$ nodes in the public cloud to process the requests using PBFT [21]. The untrusted primary of view v in the UPDC mode is replica p where $p = (v \mod P) + S$. Similar to the TPDC mode, since there might be more than $3m + 1$ replicas in the public cloud, in each view, $3m + 1$ are chosen as proxies. Node i is a proxy in view v if $i - (v \mod P) \in [S, ..., S + 3m]$. This ensures that the primary is always a proxy. As indicated in Figure 7.1(c), similar to PBFT, the UPDC mode processes the requests in three phases: pre-prepare, prepare, and commit. As can be seen, the replicas in the private cloud have no participation in any phases and are only informed about the committed requests. The total number of exchanged messages in the UPDC mode is $N + 2 * (3m + 1)^2 + (1 + S) * (3m + 1)$.

The normal case operation of this mode is similar to PBFT [21] in terms of the required number of phases (three) and the message passing pattern. Figure 7.1(c) show the normal case operation of SeeMoRe in the Peacock mode. As can be seen, the replicas in the private cloud have no participation in any phases and are only informed about the committed request.

In particular, the algorithm works as follow. A client sends a request μ to an untrusted primary. The primary then, validates the message, assigns a sequence number to the request and multicasts a signed pre-prepare message along with the request to all the replicas. The primary also, adds the digest of μ to the message which is used later by the other replicas to validate the message.

Upon receiving a valid pre-prepare message from the primary, a proxy checks the message and multicasts a signed prepare message to all other proxies. Nodes other than

proxies only log the received pre-prepare message. When a proxy receives $2m + 1$ valid matching prepare messages from other proxies for a request, it multicasts a commit message to all other proxies. As soon as a proxy has accepted $2m+1$ commits (possibly including its own) from different proxies that match the pre-prepare message for the request, it executes the request and sends a reply including the result of the execution to the client. The client waits for $m+1$ valid replies from different proxies of that view before accepting the result. When the request is committed, each proxy r also sends a signed inform message $\left\langle \text{INFORM}, v, n, r, d \right\rangle_{\sigma_r}$ to all the nodes in the private cloud and all non-proxy nodes in the public cloud. Other nodes also wait for $m+1$ valid matching inform messages from different proxies before executing the request.

State Transfer. In the Peacock mode of the protocol, since the primary might be a malicious node, we can not rely on it to produce the checkpoint, so all the replicas participate in checkpointing. Once a replica r executes a request with a sequence number n equal to the checkpoint period, it produces a checkpoint message $\left\langle \text{CHECKPOINT}, n, d, r \right\rangle_{\sigma_r}$ and multicasts the message to the other replicas. The replica has to sign the checkpoint message because it will be used later in view changes. A server considers that a checkpoint is *stable* when it receives $2m + c + 1$ checkpoint messages for sequence number n with the same digest d from different replicas (possibly including its own message). This set of messages is called a *checkpoint certificate*, which proves that the replica's state was correct until that request execution.

View Changes. In the UPDC mode, we rely on a trusted node in the private cloud, called *transferer*, to change the view. Indeed, instead of the primary of the new view, a transferer changes the view. Replica t in the private cloud is the transferer of view v' (changes the view from v to v') if $t = (v' \mod S)$. Choosing a transferer to change views helps in minimizing the size of new-view messages and more importantly, reduces the

delay between the request and its reply. Because even if there are consecutive malicious primary nodes, since the transferer takes care of the uncommitted requests of view v, the protocol does not carry the messages from one view to another. In contrast, in PBFT, it is possible that a valid request in view v be committed in view $v + m$ (when there are m consecutive primaries). Other than the transferer, view change in the UPDC mode is similar to PBFT. proxies multicast view-change messages consisting of the sequence number n of the last stable checkpoint known to the proxy, the checkpoint certificate ξ (a set of $2m + c + 1$ checkpoint messages), and a set $\mathcal{A}$ consisting of $2m + 1$ valid accept messages for each request with a sequence number higher than n. When the transferer t of new view $v+1$ receives $2m+1$ valid view-change messages from different proxies of view v, it multicasts a new-view message to all replicas in both public and private clouds. The new-view message contains a set of prepare messages $\mathcal{P}'$ where for each sequence number n, if the transferer finds a set of $2m + 1$ valid accept message in set $\mathcal{A}$ of a received view-change messages, it adds a prepare message $\left\langle \text{PREPARE}, v+1, n, d \right\rangle_{\sigma_{p'}}$ to $\mathcal{P}'$. Else, the transferer adds a no-op prepare message $\left\langle \text{PREPARE}, v+1, n, d^{\emptyset} \right\rangle_{\sigma_{p'}}$ to $\mathcal{P}'$. Here, since the transferer is trusted, it does not need to put all the view-change messages in the new-view message.

The transferer then logs all the prepare messages that are sent and updates its checkpoint if needed. proxies, on the other hand, log all the prepare messages, update their checkpoints, and send an accept message to all other proxies for each prepare message. Other nodes in the private and public cloud receive the new-view to be informed that the view is changed. Once the transferer has changed the view and the new primary receives the new-view message from the transferer, it starts to process new requests in view $v + 1$.

In this mode, we might have consecutive views with malicious primaries. As a result, the uncommitted requests in view v will be carried from one view to another. However, since there are at most m malicious replicas, in the worst case scenario, after m view

changes, a non-faulty replica becomes the primary.

Correctness. In the UPDC mode, the protocol ensures safety and liveness similar to PBFT [21].

7.4.4 Dynamic Mode Switching

We now show how to dynamically switch between different modes of SeeMoRe. An enterprise might prefer to use the TPCC mode of SeeMoRe, because it needs fewer phases (in comparison to the UPDC mode) and less number of message exchanges (in comparison to the TPDC or UPDC mode). However, if the private cloud becomes heavily loaded, or at some point, a high percentage of requests are sent by clients that are far from the private cloud and much closer to the public cloud, it might be beneficial to switch to the TPDC or UPDC mode. SeeMoRe might also plan to switch back to the TPCC mode, e.g., when the load on the private cloud is reduced. To change the mode, the protocol also has to change the view, because the primary and the set of participant replicas might be different in different modes. Therefore, to handle a mode change, the protocol first performs a view change, and then the primary of the new view in the new mode starts to process new requests.

For the switch to happen a *trusted* replica s multicasts a $\langle \textsf{MODE-CHANGE}, v+1, \pi' \rangle_{\sigma_s}$ to all the replicas where π' is the new mode of the protocol, i.e., TPCC, TPDC, or UPDC. When the protocol wants to switch to the TPCC or TPDC mode, replica s is the primary of view $v+1$, and when it switches to the UPDC mode, replica s is the transferer of view $v+1$.

Table 7.1: Comparison of fault-tolerant protocols

Protocol	phases	messages	Receiving Network	Quorum size
TPCC	2	$\mathcal{O}(n)$	$3m+2c+1$	$2m+c+1$
TPDC	2	$\mathcal{O}(n^2)$	$3m+1$	$2m+1$
UPDC	3	$\mathcal{O}(n^2)$	$3m+1$	$2m+1$
Paxos	2	$\mathcal{O}(n)$	$2f+1$	$f+1$
PBFT	3	$\mathcal{O}(n^2)$	$3f+1$	$2f+1$
UpRight	2	$\mathcal{O}(n^2)$	$3m+2c+1$	$2m+c+1$

7.4.5 Discussion

In this section, we compare the different modes of SeeMoRe with three well-known protocols: the crash fault-tolerant protocol Paxos [20], the Byzantine fault-tolerant protocol PBFT [21], and the hybrid fault-tolerant protocol UpRight [45]. We consider (1) the number of communication phases, (2) the number of message exchanges, (3) the receiving network size, and (4) the quorum size in this comparison. The results are reported in Table 7.1.

The knowledge of where a crash or a malicious failure may occur and thus choosing a trusted primary simply reduces one phase of communication. In fact, in PBFT, the prepare phase is needed only to make sure that non-faulty replicas receive matching pre-prepare messages from the primary. In contrast, in the TPCC and TPDC modes of SeeMoRe, since the primary is a trusted node, replicas receive the same message from the primary, thus there is no need for that phase of communication and the requests, similar to Paxos, are processed in two phases (while in contrast to Paxos malicious failures can occur in the public cloud). In comparison to Upright, although Upright processes the requests in two phases, it utilizes the speculative execution technique introduced by Zyzzyva [39] which becomes costly in the presence of failures.

The number of message exchanges in the TPCC mode is similar to Paxos and is linear in terms of the total number of replicas. In the TPDC mode, the number of messages

is quadratic, however it is still much less than PBFT (since it has one phase of n-to-n communication instead of two). UPDC and Upright also have a quadratic number of messages. The higher number of message exchanges results in higher latency especially in networks with a large number of nodes.

The TPCC mode, similar to Upright, needs $3m + 2c + 1$ nodes to receive a client request. In the TPDC mode, however, only the trusted primary and $3m + 1$ nodes from the public cloud participate in each phase. Since the UPDC mode utilizes PBFT, the number of phases and message exchanges are the same as PBFT. However, since the primary is in the public cloud, communicating with the private cloud has no advantage, thus it proceeds with $3m+1$ nodes instead of $3m+2c+1$ as in the TPCC mode and UpRight.

7.5 Experimental Evaluation

This section evaluates the performance of the SeeMoRe protocol. SeeMoRe is implemented by adapting the BFT-SMaRt library [107]. We mainly reuse the communication layer of BFT-SMaRt but implement our agreement and view change routines for the different modes of the protocol. Note that the SeeMoRe implementation follows the optimized implementation of Paxos and PBFT from the original BFT-SMaRt codebase, resulting in a similar implementation complexity.

In each experiment, we compare different modes of SeeMoRe with an asynchronous crash fault-tolerant (CFT) protocol, an asynchronous Byzantine fault-tolerant (BFT) protocol, and a simplified version of the asynchronous hybrid fault-tolerant protocol UpRight [45] (we call it *S-UpRight*). For both CFT and BFT we use the original BFT-SMaRt codebase (the optimized implementations of Paxos [20] and PBFT [21]). UpRight consists of first, a hybrid model that tolerates both crash and malicious failures (in a

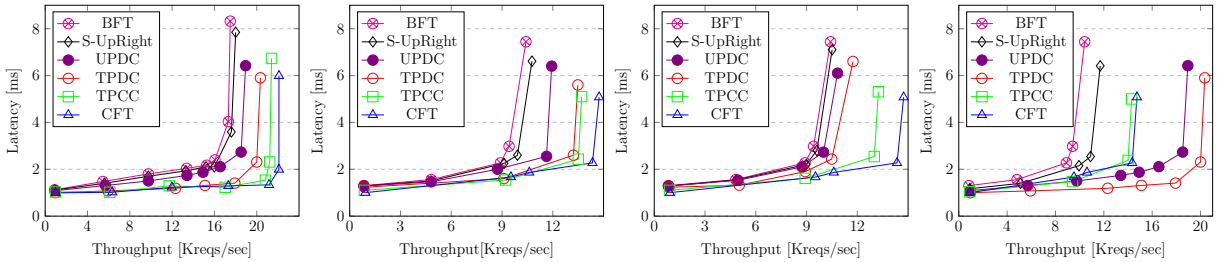
network of size $3m + 2c + 1$), and second, an optimistic protocol that combines a set of techniques such as speculative execution [39] and separation of ordering and execution [104]. S-UpRight includes the UpRight hybrid model since this part of the UpRight is relevant to SeeMoRe, however, to ensure a fair comparison with other protocols and since all other protocols use the pessimistic approach, we use a PBFT-like protocol (i.e., PBFT protocol with $3m + 2c + 1$ nodes instead of $3f + 1$ nodes) instead of the UpRight protocol. Note that, both the speculative execution and separation of ordering from execution techniques can be integrated into SeeMoRe as well.

The experiments were conducted on the Amazon EC2 platform. Each VM is Compute Optimized c4.2xlarge instances with 8 vCPUs and 15GB RAM, Intel Xeon E5-2666 v3 processor clocked at 3.50 GHz. In the experiments (except for part C), both the public and private clouds are located in the same data center i.e., AWS US West Region.

In each experiment, we vary the number of requests sent by all the clients per second from 10^3 to 10^6 (by increasing the number of clients running on a single VM) and measure the end-to-end throughput (x axis) and latency (y axis) of the system. Each client waits for the reply before sending a subsequent request.

7.5.1 Fault-Tolerance Scalability

In the first set of experiments, we evaluate the performance of SeeMoRe with different number of maximum possible failures (f). We consider the 0/0 micro-benchmark (both request and reply payload sizes are close to 0 KB) and evaluate SeeMoRe, S-UpRight, CFT, and BFT protocols. Since, $f = c + m$, we evaluate CFT and BFT to tolerate $c + m$ failures in each experiment. In all these scenarios and for SeeMoRe, we put $2c$ nodes in the private and $3m + 1$ nodes in the public cloud. The results are shown in Fig. 7.2(a)-(d).



(a) $f = 2$ ($c = 1$, $m = 1$), N: SeeMoRe, S-UpRight=6, CFT=5, BFT=7

(b) $f = 4$ ($c = 2$, $m = 2$), N: SeeMoRe, S-UpRight=11, CFT=9, BFT=13

(c) $f = 4$ ($c = 1$, $m = 3$), N: SeeMoRe, S-UpRight=12, CFT=9, BFT=13

(d) $f = 4$ ($c = 3$, $m = 1$), N: SeeMoRe, S-UpRight=10, CFT=9, BFT=13

Figure 7.2: Performance with different number of failures

In the first scenario, when $f = 2$ ($c = m = 1$), the network size of the different protocols is close to each other (BFT requires 7, SeeMoRe and S-UpRight require 6, and CFT requires 5 nodes). As a result, as can be seen in Fig. 7.2(a), the performance of the TPCC mode becomes very close to CFT (8% difference in their peak throughput). Similarly, the performances of S-UpRight and BFT are close to each other (4% difference in their peak throughput). Note that the UPDC mode shows better performance than S-UpRight (still worst than the TPDC and TPCC modes) because in the UPDC mode, SeeMoRe relies only on the public cloud which consists of only 4 nodes. In addition, while in comparison to the TPCC mode, both the UPDC and TPDC modes need less number of nodes, the TPCC mode has better performance because it needs less number of phases and message exchanges.

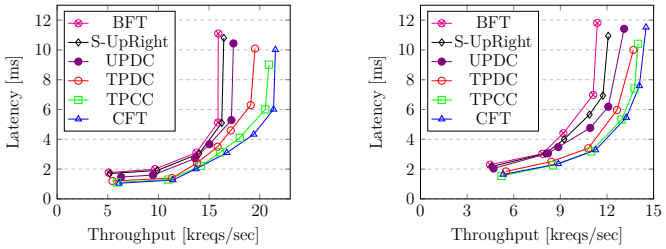
In the next three scenarios, the network tolerates the same number of failures ($f = 4$), as a result, the performance of BFT and CFT does not change from one scenario to another. However, since the number of crash and malicious failures are varied, the network size of SeeMoRe and S-UpRight changes. Hence, they show different performance in different scenarios.

When both m and c increase to 2 (Fig. 7.2(b)), The TPDC mode shows similar

performance to the TPCC mode. This is the result of the trade-off between the quorum size and the message complexity; Only 5 nodes ($2m + 1$) participate in the TPDC mode which requires $\mathcal{O}(n^2)$ number of messages whereas the quorum size of the TPCC mode is 7 ($2m+c+1$) but it requires $\mathcal{O}(n)$ messages (see Table 7.1). In addition, since SeeMoRe in the UPDC mode communicates with only 7 nodes, it shows much better performance than BFT (24% more throughput) and even S-UpRight (18% more throughput).

By increasing the number of tolerated malicious failures to 3 while reducing the number of tolerated crash failures back to 1 (Fig. 7.2(c)), the network size of SeeMoRe becomes closer to the BFT network size. As a result, CFT shows better performance (12% difference in its peak throughput) than the TPCC mode and also the performance of the UPDC and TPDC modes, which communicate with 10 nodes in the public cloud, becomes closer to S-UpRight and BFT (with 12 and 13 nodes).

On the other hand, increasing the number of tolerated crash failures to 3 while maintaining the number of malicious failures to 1 (Fig. 7.2(d)) results in a network size close to CFT. In this setting, the performance of the TPDC and UPDC modes become better than both the TPCC mode and CFT. This is expected because the TPDC mode processes a request in the public cloud which needs only 4 replicas (since $m = 1$) but with the same number of phases as the TPCC mode. Similarly, although the UPDC mode processes requests in three phases, since it needs fewer servers to proceed, its performance is better than the TPCC mode and CFT. In fact, since the number of malicious failures in this scenario is the same as the first scenario, both the TPDC and UPDC modes show the same performance as the first scenario (Fig. 7.2(a)).



(a) Benchmark 0/4 (b) Benchmark 4/0

Figure 7.3: Performance with different payload size ($c = m = 1$)

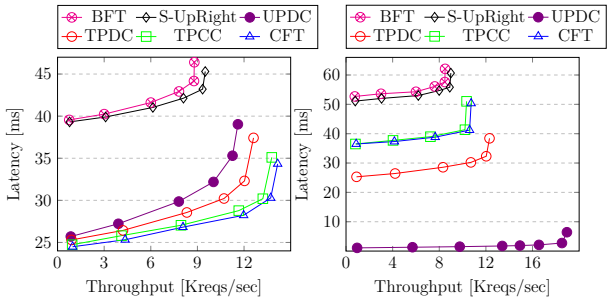
7.5.2 Changing Payload Size

We now repeat the base case scenario (c=m=1) of the previous experiments (Fig. 7.2(a)) using two micro-benchmarks 0/4, 4/0 to show how request and reply sizes affect the performance of different protocol. Figs. 7.3(a) and 7.3(b) show the throughput and latency for 0/4 and 4/0 micro-benchmarks respectively. Since the TPCC and TPDC modes need less communication phases and message exchanges, their performance is close to CFT, e.g., for latency equal to 4 ms, the throughput of the TPCC and TPDC modes is 10% and 17% less than CFT respectively. Similarly, the UPDC mode and S-UpRight are close to BFT, e.g., with 4 ms latency, the throughput of the UPDC mode is the same as BFT. Note that due to the overhead of request transmission, the request size affects the performance of all protocols more than the reply size.

7.5.3 Scalability Across Multiple Data Centers

We next repeat the base case scenario (c=m=1) of the first experiment (Fig. 7.2(a)), however, place the private and public clouds on different data centers, i.e., California and Oregon, with RTT = $22ms$ and place clients first close to the private cloud (Fig. 7.4(a)) and then close to the public cloud (Fig. 7.4(b)). In this set of experiments, we assume that the primary node of CFT, BFT, and S-UpRight protocols is in the private cloud.

Fig. 7.4(a) clearly shows the advantages of SeeMoRe as the clients are close to the



(a) Clients close to Private Cloud

(b) Clients close to Public Cloud

Figure 7.4: Performance with multiple data centers ($c = m = 1$)

private cloud. In this case all requests in all three modes of SeeMoRe as well as CFT only require two phases of cross-cloud communication (one round trip). BFT and S-UpRight, on the other hand, require three phases of communication between the clouds which results in significantly higher latency.

Fig. 7.4(b) clearly demonstrates the significant advantages of the UPDC mode, where the clients are close to the public cloud and hence all requests are entirely processed in the public cloud without any cross-cloud communication. TPDC requires two cross-cloud phases of communications (clients to the primary and the primary to the public cloud) whereas TPCC as well as CFT process the requests with three phases of cross-cloud communication. Finally, BFT and S-UpRight process requests with higher latency because of the four required phases of cross-cloud communication (including request messages coming from clients to the primary).

Comparing the results of multi data centers experiments and the experiments with more number of nodes shows that latency within a quorum of recipients (across data centers) is much more important than the quorum size.

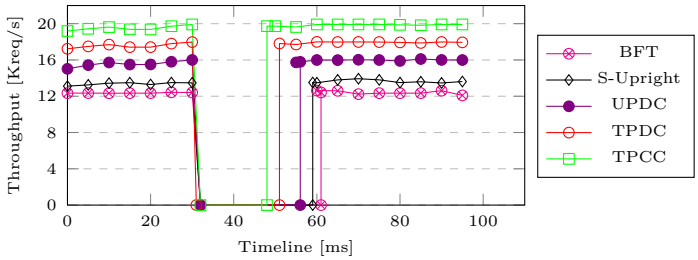


Figure 7.5: Performance during view change

7.5.4 Performance During View Change

Finally, we evaluate the impact of view change on the performance of SeeMoRe. We trigger a primary failure during the processing of the last request before the end of a checkpoint period to evaluate the worst-case overhead that can be caused by a failure. To simulate failures, the process of the faulty nodes has been terminated. We consider the base case scenario ($c = m = 1$) with a total network of $N = 6$ nodes (for SeeMoRe), where 2 nodes are in the private cloud and 4 in the public cloud (both clouds are placed in the same data center). The experiment was run with micro-benchmark 0/0 and with a checkpoint period of 10000 request i.e., a checkpoint is taken every 10000 requests. Fig. 7.5 shows the behavior of SeeMoRe, S-UpRight and BFT where the y-axis is throughput and the x-axis is a timeline with a failure injected around time 30. As can be seen, the protocols behave as expected until the failure is triggered. This failure and the view change routine cause the protocols to be temporarily out of service (in particular, 15, 20, and 24 millisecond in the TPCC, TPDC, and UPDC modes respectively). However, when the view change is complete, the throughput increases to the original level for each protocol. As can be seen, BFT takes twice as much time as the TPCC mode to revive and continue to process the requests. The UPDC mode also recovers faster than S-UpRight and BFT due to its use of transferers. Note that since mode switching is performed in the same way as view change, the results of this experiment are applicable to mode

switching as well.

Overall, the evaluation results for a network that tolerates $f = m + c$ failures where m and c are the number of malicious and crash failures respectively, can be summarized as follow. First, when c is equal or less than m (for small c and m), the performance of SeeMoRe in the TPCC mode is very close to Paxos due to the required number of phases and message exchanges in the TPCC mode. In addition, when c is larger than m, SeeMoRe in both TPDC and UPDC modes demonstrates better performance than the TPCC mode and Paxos since in both modes, SeeMoRe relies completely on the public cloud to process the requests. Furthermore, if the clients are close to the public cloud, UPDC processes the requests with significantly lower latency. Moreover, all three modes of SeeMoRe show better performance than the hybrid protocol S-UpRight since SeeMoRe is aware of where different types of faults may occur. Finally, all three modes also have better performance than BFT since they reduce the number of communication phases, messages exchanged and required nodes.

7.6 Summary

In this chapter, we proposed SeeMoRe, a hybrid state machine replication protocol to tolerate both crash and malicious failures in a public/private cloud environment. SeeMoRe is targeted to be used by smaller enterprises that own a small set of servers and intend to rent servers from public cloud providers. Such an enterprise can highly benefit from SeeMoRe, as the protocol distinguishes between crash failures that could occur within the trusted private cloud and malicious failures that could only occur in the public cloud. SeeMoRe can execute in any one of three modes, TPCC, TPDC, and UPDC, and can dynamically switch among these modes. The TPCC and TPDC modes of SeeMoRe require less communication phases and message exchanges while the UPDC

mode is useful for a heavily loaded private cloud or when there is a large network distance between the two clouds. Our evaluations show that the performance of TPCC and TPDC modes is close to Paxos while in contrast to Paxos, which only tolerates crash failures, malicious failures can occur in both TPCC and TPDC. In the UPDC mode, since the primary is in the public cloud, its performance is similar to PBFT with m failures. However, in comparison to UpRight, which requires quorums of size $2m + c + 1$, UPDC needs quorums of size $2m + 1$, and hence is more efficient.

Chapter 8

Related Work

8.1 Performance and Confidentiality

A permissioned blockchain consists of a set of known, identified nodes that might not fully trust each other. In permissioned blockchains, traditional consensus protocols can be used to order the requests [72]. The Order-execute paradigm is widely used in different permissioned blockchains. Existing permissioned blockchains that employ the order-execute paradigm, differ mainly in their ordering routines. The ordering protocol of Tendermint [33] differs from the original PBFT in two ways, first, only a subset of nodes participate in the consensus protocol and second, the leader is changed after the construction of every block (leader rotation). Quorum [31] as an Ethereum-based [46] permissioned blockchain introduces a consensus protocol based on Raft [108]: a well-known crash fault-tolerant protocol. Quorum, similar to CAPER, supports public and private transactions, however, in Quorum, both public and private transactions are ordered using the same consensus protocol resulting in low throughput. Quorum also uses the Zero-knowledge proof technique to ensure confidentiality of private transactions. Chain Core [109], Multichain [32], Hyperledger Iroha [110], Corda [111], and

ResilientDB [112], are some other prominent permissioned blockchains that follow the order-execute paradigm. Hyperledger Fabric [27] is a permissioned blockchain that employs the execute-order-validate (XOV) paradigm introduced by Eve [73]. Fabric leverages parallelism by executing the transactions of different enterprises simultaneously and presents modular design, pluggable fault-tolerant protocol, policy-based endorsement, and non-deterministic execution for the first time in the context of permissioned blockchains. In a recent release, Fabric also utilizes the Raft protocol [108] for its ordering service where a leader node is elected (per channel) and replicates messages to the followers. Raft mainly helps organizations to have their own ordering nodes, participating in the ordering service, which leads to a more decentralized system. Several recent studies attempt to improve the performance of Fabric [113–119]. ParBlockchain utilizes some of the Fabric properties such as modular design and pluggable fault-tolerant protocol. However, ParBlockchain is an *order-execute* paradigm. In addition, while Fabric checks the read-write conflict in the last phase (validation) which might result in transaction abort, ParBlockchain ensures correct results by generating dependency graphs in the first phase (ordering). As a result, workloads with contention benefit most from ParBlockchain. Fabric also needs four phases of communications other than the ordering protocol (clients to endorsers, endorsers to clients, clients to orderers, and orderers to peers) while ParBlockchain requires three phases (clients to orderers, orderers to executors, executors to peers) resulting in less latency. It should be noted that since execution is the first phase of Fabric, in comparison to ParBlockchain, its performance is less affected by the inconsistencies between the execution results (arise from malicious executors or non-deterministic execution). CAPER also utilizes some of the Fabric properties such as modular design and pluggable protocol. In addition, and in contrast to single-channel Fabric, CAPER constructs blocks simultaneously and ensures the confidentiality of both transaction data and ledger, whereas Fabric ensures only transaction data confidentiality

using Private Data Collections [29]. Private Data Collections manage confidential data that two or more entities want to keep private from other entities. Fabric also addresses atomic cross-chain swap between permissioned blockchains that are deployed on different channels by either assuming the existence of a trusted channel among the participants or using an atomic commit protocol [28] [27]. CAPER is different from Fabric in two ways: First, cross-chain communications follow the service level agreements and are visible to everyone, and second, CAPER does not need a trusted channel among the participants.

Enterprises deployed on the same or different blockchains need to communicate with each other in order to exchange assets or information. Atomic cross-chain swaps [24] are used for trading assets on two unrelated blockchains. Atomic swaps use hash-lock and time-lock mechanisms to either perform all or none of a cryptographically linked set of transactions. AC^3WN [25] is another atomic cross-chain commitment protocol where an open permissionless network of witnesses is used to guarantee that conflicting events could never simultaneously occur and either all smart contracts in an atomic cross-chain transaction are redeemed or all of them are refunded. Interledger protocols (ILPV [26]) which are presented by the World Wide Web Consortium (W3C) use a generalization of atomic swaps and enable secure transfers between two blockchain ledgers using escrow transactions. Since the redemption of an escrow transaction needs fulfillment of all the terms of an agreement, the transfer is atomic. Lightning network [120] [34] also generalizes atomic swap to transfer assets between two different clients via a network of micro-payment channels. Blocknet [121], BTC [122], Xclaim [123], POA Bridge [124] (designed specifically for Ethereum), Wanchain [125], and Fusion [126] are some other blockchain systems that allow users to transfer assets between two chains.

Using sidechains is proposed in [127] to transfer assets from a main blockchain to the sidechain(s) and execute some transactions in the sidechain(s). Sidechains can reduce confirmation time, support more functionality than the main blockchain, and reduce

the transaction cost. In sidechains, a set of known nodes, called functionaries, are responsible for moving the assets back from the sidechain to the main chain. Liquid [128], Plasma [129], Sidechains [130], and RSK [131] are some other blockchain systems that use sidechains. Polkadot [132] and Cosmos [133] also construct a main chain which is used by a set of (side) blockchains, i.e., parachains in Polkadot and zones in Cosmos, to exchange value or information. Both Polkadot and Cosmos rely on Byzantine consensus protocols in both sender and receiver sides.

The DAG structure is mainly used to increase the throughput of the system by exploiting the parallel construction of blocks resulting in the parallel execution of transactions in different blocks. In such a structure, the blocks (transactions) that are independent of each other can be appended to the ledger simultaneously. Since in a DAG structure, the blocks are constructed in parallel, existing permissionless blockchain systems present different techniques to prevent (resolve) the double spending problem. Byteball [134] and Iota [135] are two DAG structured permissionless blockchains. In Byteball, a set of privileged users, called witnesses, determines a total order on the DAG to prevent double spending, whereas, in Iota [135], the number of descendant transactions is used to commit a transaction and abort the other one. In Iota, the blockchain, called Tangle, grows in more than one direction. Indeed, once a user issues a transaction, the user must pick two existing transactions and approve them (results in adding edges from the new transaction to the existing ones). The user will also solve a small PoW puzzle similar to Bitcoin. Hashgraph [136] is another DAG structured permissioned blockchain that combines a voting algorithm with a gossip protocol to achieve consensus among nodes. In Hashgraph nodes submit transactions (events) and gossip about transactions by randomly choosing other nodes (neighbors). Each transaction in Hashgraph includes the hash of the previous transactions of both sender and receiver. This process continues until convergence when all nodes become aware of all transactions. Hashgraph, in contrast to CAPER,

does not distinguish between internal and cross-enterprise transactions which results in lower performance as well as confidentiality issues. Vegvisir [137], which is designed for IoT environments, Ghost [138], Inclusive protocol [139], DagCoin [140], Phantom [141], Spectre [142], and MeshCash [143], are some other DAG structured blockchain systems. In CAPER, in contrast to all these blockchains, internal transactions of different enterprises are added independent of each others and only cross-enterprise transactions need a global consensus. As a result, first, the double spending problem never occurs, and second, internal transactions can be processed simultaneously, which results in lower latency and higher throughput. In SharPer and in contrast to all these blockchains, since intra-shard transactions of different clusters access disjoint data shards, they can be processed simultaneously which results in lower latency and higher throughput.

8.2 Verifiability

Enhancing privacy in the context of crowdworking has been addressed by several recent studies with various kinds of guarantees, from differential privacy [144, 145] to cryptography [146–148], mostly focusing on spatial crowdsourcing and the use of geo-location to perform assignment. In ZebraLancer [30] and ZKCrowd [149], blockchains and consensus protocols are also used to add transparency guarantees on top of privacy. However, all these works consider a single-platform context, with no external constraints, preventing many real-life legislation to apply. To the best of our knowledge, SEPAR is the first to support a multi-platform crowdworking context, with external constraints, transparency, and privacy expectations at the same time.

Providing anonymity as well as untraceability has been addressed by ZCash [150] which is restricted to the management of crypto-currency issues. Hawk and Raziel [151, 152] manage wider issues, and include general smart contracts. However, these solutions

do not incorporate infrastructures with multiple platforms, nor implement constraints (let alone anonymized ones). Finally, Solidus [153] proposes to privately manage a multi-platform banking system, with individual banks managing their own clients, while allowing cross-platform transactions. While Solidus may be sufficient for banking systems, it does not consider users that subscribe to multiple platforms, nor envisions global profiles or constraints.

8.3 Scalability

Scalability is the ability of a system to process an increasing number of transactions by adding resources to the system. While the Visa payment service is able to handle on average 2000 transactions per second, Bitcoin and Ethereum can handle at most 7 and 15 transactions per second respectively. To address the scalability issue different techniques have been proposed. Off-chain (layer two) solutions, which are built on top of the main-chain, do not increase the throughput of the protocol, rather move a portion of the transactions off the chain. For example, in Lightning Networks [78] [34], assets are transferred between two different clients via a network of micro-payment channels instead of the main blockchain. While off-chain solutions increase the throughput of the system, they suffer from security issues, e.g., denial-of-service attacks.

In general security, decentralization, and performance are known as the *scalability trilemma* in blockchain systems. Security requires resistance to threats such as the denial-of-service attacks, 51% attack, or Sybil attacks; decentralization means no single entity can hijack the chain, censor it, or introduce changes in governance; and performance is the ability to handle thousands of transactions per second.

On-chain (layer one) solutions, on the other hand, increase the throughput of the main chain. Layer one solutions are categorized into vertical and horizontal techniques.

In vertical scalability, more power is added to each node to perform more tasks. One trivial solution is to increase the block size which results in processing more transactions at once, thus enhancing performance. Increasing the block size, however, increases both the propagation time and the verification time of the block which makes operating full nodes more expensive, and this in turn could cause less decentralization in the network.

Horizontal techniques, on the other hand, increase the number of nodes in the network to process more transactions. However, most blockchain systems require every transaction to be processed by every single node in the network. As a result, increasing the number of nodes, does not necessarily enhance the performance of the system.

Another horizontal solution to enhance the scalability of blockchain systems is sharding. Partitioning the data into multiple shards that are maintained by different subsets of nodes is a proven approach to enhance the scalability of databases [35]. Data sharding techniques are commonly used in globally distributed databases such as H-store [154], Calvin [60], Spanner [35], Scatter [155], Google's Megastore [156], Amazon's Dynamo [36], Facebook's Tao [37], and E-store [61]. In such systems servers (nodes) are assumed to be crash-only and a coordinator node is used to process crash-shard transactions. Agrawal et al. [157] categorize sharded, replicated database systems into replicated object systems, e.g., Spanner [35] and replicated transaction systems, e.g., replicated commit protocol [158]. SharPer is inspired by distributed database systems and has applied the sharding technique to the blockchain domain. Furthermore, SharPer proposes consensus protocol for network consisting of Byzantine nodes and introduces a flattened cross-shard consensus protocol instead of a coordinator-based one.

Sharding techniques have been used in both permissionless, e.g., Elastico [81], OmniLedger [82], and Rapidchain [83], and permissioned blockchain systems, e.g., multi-channel Fabric [28], AHL [86], Cosmos [84], and RSCoin [85] to improve scalability. In Elastico [81], nodes randomly join different committees by solving some PoW puzzle.

Committees, then, run PBFT [21] individually to reach consensus on the order of intra-shard transactions. Finally, a leader committee verifies the transactions that are ordered by committees and creates a global block. In Elastico, the blockchain ledger is maintained by all nodes and cross-shard transactions are not supported. In addition, while running PBFT among hundreds of nodes decreases the performance of the protocol, reducing the number of nodes within each shard increases the failure probability [82]. The considerable overhead and latency in re-configuring committees, which is needed in every epoch, and the possibility to bias the randomness, which might result in compromising the committee selection process by malicious nodes, are some of the other drawbacks of Elastico [83].

OmniLedger [82] attempts to fix some of the drawbacks of Elastico by introducing a more secure method to assign nodes to committees and proposing an atomic protocol for cross-shard transactions. The intra-shard consensus protocol of OmniLedger uses a variant of ByzCoin [159] and assumes partially-synchronous channels to achieve fast consensus. However, it relies on a client to participate actively and coordinate a lock/unlock protocol to process cross-shard transactions which, as shown in [86], might result in blocking issues. Furthermore, as mentioned in [83], OmniLedger is vulnerable to denial-of-service (DoS) attacks. In multi-channel Fabric [27] [28], as explained in Section 4.1, processing cross-shard transactions, in contrast to SharPer, requires either the existence of a trusted channel among the participants or an atomic commit protocol (inspired by two-phase commit) [28]. Similarly, in Cosmos [84], interacting chains in any Inter-Blockchain Communication must be aware of the state of each other which requires establishing a bidirectional trusted channel between two blockchains.

AHL [86] employs a trusted hardware (the technique that is presented in [87–89]) to restrict the malicious behavior of nodes which results in committees of $2f + 1$ nodes (instead of $3f+1$). The system also relies on an extra set of nodes, called a reference com-

mittee, to process cross-shard transactions using the classic two-phase commit (2PC) and two-phase locking (2PL) protocols where the reference committee plays the coordinator role. The system, however, suffers from several drawbacks. First, running fault-tolerant protocols among 80 nodes results in high latency. Second, the protocol requires an extra set of nodes to form the reference committee resulting in significant communication overhead between nodes and the reference committee. Finally, since a single reference committee processes cross-shard transactions, the protocol is not able to process cross-shard transactions with non-overlapping clusters in parallel. In SharPer and in contrast to AHL, there is no need for an extra set of nodes to process cross-application transactions. In addition, cross-shard transactions are ordered in only three communication phases. Furthermore, cross-shard transactions with non-overlapping committees can be processed simultaneously. Note that since intra-shard consensus is pluggable, the trusted hardware technique can be employed to decrease the number of required nodes within each cluster.

8.4 Fault Tolerance

Many practical large-scale data management systems such as ISIS [160], Eternal [161], Google's Spanner [35], Amazon's Dynamo [36], and Facebook's Tao [37], use consensus protocols to provide fault tolerance. Consensus algorithms are a form of State Machine Replication [50]. SMR regulates the deterministic execution of client requests on multiple copies of a server, called replicas, such that every non-faulty replica must execute every request in the same order [51] [50].

Several approaches [51] [20] [108] generalize SMR to support crash failures among which Paxos [20] is the most well-known. Paxos guarantees safety in an asynchronous network using $2f+1$ processors despite the simultaneous crash failure of any f processors.

Many protocols are proposed to either reduce the number of phases, e.g., Multi-Paxos which assumes the leader is relatively stable or Fast Paxos [90] and Brasileiro et al. [91] which add f more replicas, or reduce the number of replicas, e.g., Cheap Paxos [162] which tolerates f failures with $f+1$ active and f passive processors.

Byzantine fault tolerance refers to servers that behave arbitrarily after the seminal work by Lamport, et al. [163]. Early Byzantine fault-tolerant protocols (SecureRing [164] and Rampart [165]) were synchronous where a round based algorithm is developed to exclude faulty nodes from the group. Such systems are vulnerable to denial-of-service attack where an attacker may compromise the safety of service by delaying non-faulty nodes or the communication between them until they are tagged as faulty and excluded from the group.

Practical Byzantine fault tolerance protocol (PBFT) [21] is one of the first and the most known state machine replication protocol to deal with Byzantine failures in an asynchronous network. Although practical, the cost of implementing PBFT is quite high, requiring at least $3f+1$ replicas, 3 communication phases, and a quadratic number of messages in terms of the number of replicas. Thus, numerous approaches have been proposed to explore a spectrum of trade-offs between the number of phases/messages (latency), number of processors, the activity level of participants (replicas and clients), and types of failures.

FaB [38] and Bosco [166] reduce the communication phases by adding more replicas. Speculative protocols, e.g., Zyzzyva [39], HQ [106], and Q/U [167], also reduce the communication by executing requests without running any agreement between replicas and optimistically rely on clients to detect inconsistencies between replicas. To reduce the number of replicas, some approaches rely on a trusted component (a counter in A2M-PBFT-EA [87], MinBFT [88], and, EBAWA [89], a hypervisor [168], or a whole operating-system instance [169]) that prevents a faulty replica from sending conflicting

(i.e., asymmetric) messages to different replicas without being detected. SBFT [170] and Hotstuff [171] attain linear communication overhead by increasing the number of communication phases and using advanced encryption techniques, e.g., signature aggregation [172]. Finally, MultiBFT [173] uses multiple parallel primary nodes to parallelize transaction processing and hence improve performance. In addition, optimistic approaches reduce the required number of replicas during the normal-case operation by either utilizing the Cheap Paxos [162] solution and keeping f replicas in a passive mode (REPBFT [174]), or by separating agreement from execution [104]. In ZZ [40] both passive replicas and separating agreement from execution are employed. Note that all these approaches need $3f + 1$ replicas upon occurrence of failures. REMINBFT [174] and CheapBFT [41] use a trusted component to reduce the network size to $2f + 1$ and then keep f of those replicas passive during the normal-case operation. In contrast to optimistic approaches, robust protocols (Prime [175], Aardvark [176], Spinning [177], RBFT [178]) consider the system to be under attack by a very strong adversary and try to enhance the performance of the protocol during periods of failure.

Consensus with multiple failure modes were initially addressed in synchronous protocols [179] [180] [181] [182]. Recent protocols such as VFT [183], XFT [43], and SBFT [170] have focused on partial synchrony, a technique that defines a threshold on the number of slow (partitioned) processes. VFT is similar to PBFT regarding the number of phases and massage exchanges, however, it optimistically assumes that an adversary cannot fully control the malicious nodes and as a result, reduces the phases of communication and message exchanges. SBFT also reduces the number of message exchanges by assuming the adversary controls only crash failures. Scrooge [44], as an asynchronous hybrid protocol, uses a speculative technique to reduce the latency. UpRight [45], which is the closest protocol to SeeMoRe, requires $3m+2c+1$ nodes as the minimum network size from which $2m + c + 1$ are required to participate in each communication quorum. UpRight

also utilizes the agreement routines of PBFT [21], Aardvark [176], and Zyzzyva [39] and, similar to [104], separates agreement from execution. However, UpRight is not aware which nodes may crash and which may be malicious, therefore, does not take advantage of this knowledge by placing particular processes executing specific protocol roles on crash-only or Byzantine sites. On the other hand, SeeMoRe knows where the crash or malicious faults may occur, thus, it either reduces the number of communication phases and message exchanges by placing the primary in the crash-only private cloud, or decreases the number of required nodes by placing the primary in the untrusted public cloud.

Storing data on multiple clouds to enhance fault tolerance is addressed for both crash (ICStore [102], SPANStore [101]) and malicious (DepSky [184], SCFS [185]) failures. DAPCC [186] solves the consensus in a dual failure mode assuming a synchronous environment. Hypris [98] reduces the number of required servers to $2f + 1$ ($f + 1$ when the system is synchronous and no faults happen) by keeping the metadata in a private cloud assumed to be partially synchronous.

Chapter 9

Conclusion

9.1 Summary and Concluding Remarks

Large-Scale Data Management systems have increasingly utilized permissioned blockchains over the past several years. The unique features of blockchain such as transparency, provenance, and authenticity, are appealing to a wide range of large-scale data management applications in permissioned settings. However, these applications deal with five main challenges, *confidentiality*, *verifiability*, *performance*, *scalability*, and *fault tolerance* that need to be supported by blockchain systems. While confidentiality and verifiability are needed in multi-application systems (to preserve the confidentiality of application data and to enable verifiability of the transactions of an application for other applications without revealing any information), performance, scalability, and fault tolerance are required in systems with either a single application or multiple applications. In this dessertation, we propose different techniques to address these challenges. A permissioned blockchain system can utilize any of these techniques to address its challenges.

9.1.1 Large-Scale Data Management Challenges

Confidentiality. Confidentiality of data is required in distributed applications consisting of a set of collaborating enterprises. In such applications, enterprises collaborate with each other following Service Level Agreements (SLAs) to provide different services. While collaboration between enterprises, e.g., cross-enterprise transactions, should be visible to all enterprise, the internal data of each enterprise, e.g, internal transactions, might be confidential. In this dissertation, we proposed CAPER, a permissioned blockchain system that supports both internal and cross-enterprise transactions of collaborating enterprises. In CAPER, the blockchain ledger is not maintained by any node and each enterprise maintains its own local view of the ledger including its internal and all cross-enterprise transactions. CAPER also distinguishes between trust at the node level and trust at the enterprise level and allows an enterprise to behave maliciously for its benefit while its nodes are non-malicious. Furthermore, CAPER introduces three consensus protocols to globally order cross-enterprise transactions: using a separate set of orderers, hierarchical consensus, and one-level consensus.

Verifiability. Verifiability deals with checking the satisfaction of predefined global constraints on the entire system in a privacy-preserving manner in many multi-application systems, participants need to verify all or some of the transactions that are initiated by other applications (platforms/enterprises) to ensure satisfaction of some predefined constraints. In particular, crowdworking platforms need to interface with legal and social institutions. Global regulations must be enforced, such as minimal and maximal work hours that participants can spend on crowdworking platforms. Moreover, while collaborating to enforce global regulations or while processing complex tasks that require the *transparent* sharing of information about the tasks, the system needs to preserve the *privacy* of all participants. In this dissertation, we presented SEPAR, a blockchain-based

multi-platform crowdworking system that enforces global constraints on distributed independent entities. In SEPAR, Privacy is ensured using lightweight and anonymous tokens, while transparency is achieved using a permissioned blockchain shared across multiple platforms. To support fault tolerance and support collaboration among platforms, SEPAR provided a suite of distributed consensus protocols.

Performance. distributed applications require high performance in terms of throughput and latency, e.g., financial applications need to process tens of thousands of requests every second with very low latency. Existing blockchain systems, however, suffer from architectural limitations resulting in performance issues. While recent permissioned blockchain systems, e.g., Hyperledger Fabric, have tried to overcome these limitations, their focus has mainly been on workloads with no-contention, i.e., no conflicting transactions. In this dissertation, we introduced OXII, a new paradigm for permissioned blockchains to support distributed applications that execute concurrently. OXII is designed for workloads with (different degrees of) contention. We then presented ParBlockchain, a permissioned blockchain designed specifically in the OXII paradigm.

Scalability. Scalability is one of the main obstacles to business adoption of blockchain systems. Scalability is the ability of a blockchain system to process a growing number of transactions by adding resources to the system. Partitioning the data into multiple shards that are maintained by different subsets of nodes is a proven approach to enhance the scalability of databases [35]. Despite recent intensive research on using sharding techniques to enhance the scalability of blockchain systems, existing solutions do not efficiently address the efficient processing of cross-shard transactions. In this dissertation, we introduce *SharPer*, a permissioned blockchain system that improves scalability by clustering (partitioning) the nodes and assigning different data shards to different clusters where each data shard is replicated on the nodes of a cluster. SharPer supports both intra-

shard and cross-shard transactions and processes intra-shard transactions of different clusters as well as cross-shard transactions with non-overlapping clusters simultaneously. SharPer also incorporates a *flattened* protocol to establish consensus among clusters on the order of cross-shard transactions.

Fault Tolerance. Large scale data management systems utilize State Machine Replication to provide fault tolerance. Fault-tolerant protocols are the main building block of permissioned blockchain systems. Fault-tolerant protocols are also extensively used in the distributed database infrastructure of large enterprises such as Google, Amazon, and Facebook. However, and in spite of years of intensive research, existing fault-tolerant protocols do not adequately address hybrid environments, e,g., cloud or cluster, consisting of trusted and untrusted environments which are widely used by enterprises. In this dissertation, we considered a trusted environment consisting of non-malicious nodes (crash-only failures) and an untrusted environment with possible malicious failures. We introduced *SeeMoRe*, a hybrid State Machine Replication protocol that uses the knowledge of *where* crash and malicious failures may occur in a hybrid environment to improve overall performance. SeeMoRe has three different modes that can be used depending on the private cloud load and the communication latency between the public and private clouds. SeeMoRe can dynamically transition from one mode to another.

9.1.2 Large-Scale Data Management Applications

We now discuss several large-scale data management applications and show how proposed techniques can be used to satisfy the requirements of these applications.

Supply Chain Management. Lack of trust between different parties is one of the most important problems in supply chain management. To tackle such an issue, a permissioned blockchain can be used to *monitor* the execution of the collaborative process and *check*

conformance between the execution and SLAs. The utilized blockchain system needs to support both internal and cross-enterprise transactions where in contrast to the cross-enterprise transactions which are visible by all participants, the internal transactions of each enterprise is confidential, e.g., the internal transactions of the Manufacturer show its internal process for producing a product which the Manufacturer might intend to keep as a secret. To address the confidentiality challenge of Supply Chain Management applications, CAPER can be used.

Large-Scale Databases. Sharding techniques are extensively used in distributed databases such as Google's Spanner [35] and Facebook's Tao [37] to address the scalability issue. Such systems mainly assume a crash failure model and rely on a trusted coordinator to process cross-shard transaction. In a blockchain-based data management system that needs to tolerate malicious failures, however, the scalability issue can be addressed using SharPer. SharPer can also be integrated with CAPER if the confidentiality of shards is required. In a hybrid environment where different shards have different failure models, SeeMoRe can also be used to provide fault tolerance.

Multi-Platform Crowdsourcing. Crowdsourcing empowers open collaboration over the Internet. A crowdsourcing system includes platforms, requesters, and workers where requesters submit their tasks and workers send their contributions for a particular task to the platform. A crowdsourcing system might need to perform thousands of transactions per second, thus, has to be high *performance*. In addition, a multi-platform crowdsourcing system should scale appropriately with the increasing number of platforms, workers, and requesters, thus requires *scalability*. Moreover, the system has to provide *verifiability* of transactions against the predefined global constraints, e.g., a worker can work *no more* than 40 hours per week or a framework should process *no less than* 100 tasks per week, while preserving the *confidentiality* of transactions. To implement such a complex

system, we have presented SEPAR. While the main focus of SEPAR was on verifiability requirement, it uses the blockchain ledger view that is presented in CAPER to ensure confidentiality. In addition, the sharding technique of SharPer as well as the parallel execution of ParBlockchain can be integrated with SEPAR to provide higher performance and scalability.

9.2 Future Directions

This section summarizes some of the future directions that we think will be areas of active research pursuit in the coming years.

A general framework for large-scale data management applications. In this dissertation, we have focused on five different requirements of large-scale data management systems and proposed five techniques to address these five requirements. We then implemented these techniques within five different systems, i.e., CAPER, SEPAR, SharPer, ParBlockchain, and SeeMoRe. While these systems can used by different applications, a future direction is to develop a general framework to enable users to choose a right set of techniques depending of the requirement of their large-scale data management application. The framework should then integrate the requested techniques and develop a permissioned blockchain that satisfies the desired requirements.

Supporting non-deterministic execution in contentious workloads. Hyperledger Fabric [27] supports non-deterministic execution of transactions by employing execute-order-validate paradigm. In this paradigm, since transactions are executed at the first phase by a set of nodes (endorsers), any non-deterministic execution of transactions can be detected easily. Fabric, in the presence of any contention in the workload, however, has to disregard the effects of conflicting transactions which negatively impacts the performance of the system. ParBlockchain, on the other hand, orders transaction first

and generates a dependency graph exposing conflicts between transactions. As a result, ParBlockchain is able to handle contentious workloads without rolling back the processed transactions or executing transactions sequentially. However, since transactions are executed in the last phase, any non-deterministic execution of transactions negatively impacts the performance of the system. A research problem is to design a permissioned blockchain system that supports non-deterministic execution in contentious workloads.

Enhancing performance by parallel ordering of transactions. Ordering and execution are the two main phases of transaction processing. While parallel execution of transactions, as mentioned earlier, improves the performance of blockchain systems, the sequential ordering of transactions is still a reason for poor performance. Nodes in a blockchain system establish consensus on the total order of transaction blocks to guarantee data consistency in the presence of data dependency among transactions. In many applications, however, there is no need for the sequential ordering of all transactions. A research goal is to enhance the performance of permissioned blockchain systems by supporting parallel ordering of transactions.

Data Analytics in Blockchain. Blockchain technology is rapidly transforming the Big Data Analytics landscape. Blockchain has brought a whole new way of managing and operating with data. Blockchain systems could help data scientists in different ways such as ensuring trust, making predictions, and enabling real-time data analysis. Exploring the connection between blockchain and data analytics is another direction to pursue.

Designing the Right set of Fault-tolerant Protocols. Fault-tolerant protocols are an essential component of any large-scale data management system. On one hand, while fault-tolerant protocols have explored a spectrum of performance trade-offs between the number of required participants, number of phases/messages (latency), and message complexity, they have not adequately addressed all the characteristics of large-scale data

management applications. On the other hand, many large-scale data management applications still rely on basic protocols that might not be adapted for such applications. In particular, the increasing number of malicious attacks in data management systems, which results in changing the trust assumptions about the underlying infrastructure, as well as introducing more complex data management systems, emphasizes the need for developing the right set of protocols for enterprises to manage their data on untrusted infrastructures.

Bibliography

[1] M. J. Amiri, J. Duguépéroux, T. Allard, D. Agrawal, and A. El Abbadi, *Separ: Towards regulating future of work multi-platform crowdworking environments with privacy guarantees*, in *Proceedings of The Web Conf. (WWW)*, pp. 1891–1903, 2021.

[2] M. J. Amiri, D. Agrawal, and A. El Abbadi, *SharPer: Sharding permissioned blockchains over network clusters*, in *SIGMOD Int. Conf. on Management of Data*, pp. 76–88, ACM, 2021.

[3] M. J. Amiri, D. Agrawal, and A. El Abbadi, *Permissioned blockchains: Properties, techniques and applications*, in *SIGMOD Int. Conf. on Management of Data*, pp. 2813–2820, 2021.

[4] M. J. Amiri, S. Maiyya, D. Agrawal, and A. El Abbadi, *SeeMoRe: A fault-tolerant protocol for hybrid cloud environments*, in *Int. Conf. on Data Engineering (ICDE)*, pp. 1345–1356, IEEE, 2020.

[5] M. J. Amiri, D. Agrawal, and A. El Abbadi, *Modern large-scale data management systems after 40 years of consensus*, in *Int. Conf. on Data Engineering (ICDE)*, pp. 1794–1797, IEEE, 2020.

[6] D. Agrawal, A. El Abbadi, M. J. Amiri, S. Maiyya, and V. Zakhary, *Blockchains and databases: Opportunities and challenges for the permissioned and the permissionless*, in *European Conf. on Advances in Databases and Information Systems*, pp. 3–7, Springer, 2020.

[7] M. J. Amiri, D. Agrawal, and A. El Abbadi, *CAPER: a cross-application permissioned blockchain*, *Proc. of the VLDB Endowment* **12** (2019), no. 11 1385–1398.

[8] M. J. Amiri, D. Agrawal, and A. El Abbadi, *ParBlockchain: Leveraging transaction parallelism in permissioned blockchain systems*, in *Int. Conf. on Distributed Computing Systems (ICDCS)*, pp. 1337–1347, IEEE, 2019.

[9] M. J. Amiri and D. Agrawal, *View: an incremental approach to verify evolving workflows*, in *ACM/SIGAPP Symposium on Applied Computing*, pp. 85–93, ACM, 2019.

[10] M. J. Amiri, D. Agrawal, and A. El Abbadi, *On sharding permissioned blockchains*, in *Int. Conf. on Blockchain*, pp. 282–285, IEEE, 2019.

[11] V. Arora, M. J. Amiri, D. Agrawal, and A. El Abbadi, *M-db: A continuous data processing and monitoring framework for iot applications*, in *Int. Conf. on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 1096–1105, IEEE, 2019.

[12] V. Zakhary, M. J. Amiri, S. Maiyya, D. Agrawal, and A. El Abbadi, *Towards global asset management in blockchain systems*, *arXiv preprint arXiv:1905.09359* (2019).

[13] S. Maiyya, V. Zakhary, M. J. Amiri, D. Agrawal, and A. El Abbadi, *Database and distributed computing foundations of blockchains*, in *SIGMOD Int. Conf. on Management of Data*, pp. 2036–2041, ACM, 2019.

[14] M. J. Amiri, M. Koupaee, and D. Agrawal, *On similarity of object-aware workflows*, in *Int. Conf. on Service-Oriented System Engineering (SOSE)*, pp. 84–845, IEEE, 2019.

[15] A. Y. Seqerloo, M. J. Amiri, S. Parsa, and M. Koupaee, *Automatic test cases generation from business process models*, *Requirements Engineering* **24** (2019), no. 1 119–132.

[16] M. J. Amiri, *Object-aware identification of microservices*, in *Int. Conf. on Services Computing (SCC)*, pp. 253–256, IEEE, 2018.

[17] M. J. Amiri and M. Koupaee, *Data-driven business process similarity*, *IET Software* **11** (2017), no. 6 309–318.

[18] M. Koupaee, M. R. Kangavari, and M. J. Amiri, *Scalable structure-free data fusion on wireless sensor networks*, *The Journal of Supercomputing* **73** (2017), no. 12 5105–5124.

[19] M. J. Amiri, S. Parsa, and A. M. Lajevardi, *Multifaceted service identification: process, requirement and data*, *Computer Science and Information Systems* **13** (2016), no. 2 335–358.

[20] L. Lamport, *Paxos made simple*, *ACM Sigact News* **32** (2001), no. 4 18–25.

[21] M. Castro, B. Liskov, *et. al.*, *Practical byzantine fault tolerance*, in *Symposium on Operating systems design and implementation (OSDI)*, vol. 99, pp. 173–186, USENIX Association, 1999.

[22] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, .

[23] C. Cachin and M. Vukolić, *Blockchain consensus protocols in the wild*, in *Int. Symposium on Distributed Computing (DISC)*, pp. 1–16, 2017.

[24] M. Herlihy, *Atomic cross-chain swaps*, in *Symposium on Principles of Distributed Computing (PODC)*, pp. 245–254, ACM, 2018.

[25] V. Zakhary, D. Agrawal, and A. El Abbadi, *Atomic commitment across blockchains*, *Proc. of the VLDB Endowment* **13** (2020) 1319–1331.

[26] S. Thomas and E. Schwartz, *A protocol for interledger payments*, *URL https://interledger.org/interledger.pdf* (2015).

[27] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, *et. al.*, *Hyperledger fabric: a distributed operating system for permissioned blockchains*, in *European Conf. on Computer Systems (EuroSys)*, pp. 30:1–30:15, ACM, 2018.

[28] E. Androulaki, C. Cachin, A. De Caro, and E. Kokoris-Kogias, *Channels: Horizontal scaling and confidentiality on permissioned blockchains*, in *European Symposium on Research in Computer Security (ESORICS)*, pp. 111–131, Springer, 2018.

[29] Hyperledger, "Private data collections: A high-level overview." https://www.hyperledger.org/blog/2018/10/23/private-data-collections-a-high-level-overview.

[30] Y. Lu, Q. Tang, and G. Wang, *Zebralancer: Private and anonymous crowdsourcing system atop open blockchain*, in *Int. Conf. on Distributed Computing Systems (ICDCS)*, pp. 853–865, IEEE, 2018.

[31] J. M. Chase, *Quorum white paper*, 2016.

[32] G. Greenspan, *Multichain private blockchain-white paper*, *URl: http://www. multichain. com/download/MultiChain-White-Paper. pdf* (2015).

[33] J. Kwon, *Tendermint: Consensus without mining*, .

[34] J. Poon and T. Dryja, *The bitcoin lightning network: Scalable off-chain instant payments*, *https://lightning.network/lightning-network-paper.pdf* (2016).

[35] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, *et. al.*, *Spanner: Google's globally distributed database*, *Transactions on Computer Systems (TOCS)* **31** (2013), no. 3 8.

[36] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, *Dynamo: amazon's highly available key-value store*, in *Operating Systems Review (OSR)*, vol. 41, pp. 205–220, ACM SIGOPS, 2007.

[37] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, *et. al.*, *Tao: Facebook's distributed data store for the social graph*, in *Annual Technical Conf. (ATC)*, pp. 49–60, USENIX Association, 2013.

[38] J.-P. Martin and L. Alvisi, *Fast byzantine consensus*, *Transactions on Dependable and Secure Computing* **3** (2006), no. 3 202–215.

[39] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, *Zyzzyva: speculative byzantine fault tolerance*, *Operating Systems Review (OSR)* **41** (2007), no. 6 45–58.

[40] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, *Zz and the art of practical bft execution*, in *Conf. on Computer systems*, pp. 123–138, ACM, 2011.

[41] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, *Cheapbft: resource-efficient byzantine fault tolerance*, in *European Conf. on Computer Systems (EuroSys)*, pp. 295–308, ACM, 2012.

[42] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, *The next 700 bft protocols*, *Transactions on Computer Systems (TOCS)* **32** (2015), no. 4 12.

[43] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic, *Xft: Practical fault tolerance beyond crashes.*, in *Symposium on Operating systems design and implementation (OSDI)*, pp. 485–500, USENIX Association, 2016.

[44] M. Serafini, P. Bokor, D. Dobre, M. Majuntke, and N. Suri, *Scrooge: Reducing the costs of fast byzantine replication in presence of unresponsive replicas*, in *Int. Conf. on Dependable Systems and Networks (DSN)*, pp. 353–362, IEEE, 2010.

[45] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, *Upright cluster services*, in *Symposium on Operating systems principles (SOSP)*, pp. 277–290, ACM, 2009.

[46] "Ethereum blockchain app platform." https://www.ethereum.org. 2017.

[47] G. Wood, *Ethereum: A secure decentralised generalised transaction ledger*, *Ethereum project yellow paper* **151** (2014) 1–32.

[48] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling, *Untrusted business process monitoring and execution using blockchain*, in *Int. Conf. on Business Process Management (BPM)*, pp. 329–347, Springer, 2016.

[49] N. A. Lynch, *Distributed algorithms*. Elsevier, 1996.

[50] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, *Communications of the ACM* **21** (1978), no. 7 558–565.

[51] F. B. Schneider, *Implementing fault-tolerant services using the state machine approach: A tutorial*, *Computing Surveys (CSUR)* **22** (1990), no. 4 299–319.

[52] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[53] M. J. Fischer, N. A. Lynch, and M. S. Paterson, *Impossibility of distributed consensus with one faulty process*, *Journal of the ACM (JACM)* **32** (1985), no. 2 374–382.

[54] M. Castro and B. Liskov, *Practical byzantine fault tolerance and proactive recovery*, *Transactions on Computer Systems (TOCS)* **20** (2002), no. 4 398–461.

[55] G. Bracha and S. Toueg, *Asynchronous consensus and broadcast protocols*, *Journal of the ACM (JACM)* **32** (1985), no. 4 824–840.

[56] K. Korpela, J. Hallikas, and T. Dahlberg, *Digital supply chain transformation toward blockchain integration*, in *Hawaii Int. Conf. on system sciences (HICSS)*, 2017.

[57] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, *Medrec: Using blockchain for medical data access and permission management*, in *Int. Conf. on Open and Big Data (OBD)*, pp. 25–30, IEEE, 2016.

[58] H. Jin, X. Dai, and J. Xiao, *Towards a novel architecture for enabling interoperability amongst multiple blockchains*, in *Int. Conf. on Distributed Computing Systems (ICDCS)*, pp. 1203–1211, IEEE, 2018.

[59] Z. István, A. Sorniotti, and M. Vukolić, *Streamchain: Do blockchains need blocks?*, in *Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL)*, pp. 1–6, ACM, 2018.

[60] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, *Calvin: fast distributed transactions for partitioned database systems*, in *SIGMOD Int. Conf. on Management of Data*, pp. 1–12, ACM, 2012.

[61] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker, *E-store: Fine-grained elastic partitioning for distributed transaction processing systems*, *Proc. of the VLDB Endowment* **8** (2014), no. 3 245–256.

[62] J. E. Cohen, *Law for the platform economy*, *UCDL Rev.* **51** (2017) 133.

[63] M. Kenney and J. Zysman, *The rise of the platform economy*, *Issues in science and technology* **32** (2016), no. 3 61.

[64] J. Berg, M. Furrer, E. Harmon, U. Rani, and M. S. Silberman, *Digital labour platforms and the future of work: Towards decent work in the online world*. Int. Labour Office Geneva, 2018.

[65] G. C. on the Future of Work, *Work for a brighter future*, Tech. Rep. ISBN 978-92-2-132796-7, Int. Labour Organization, 2019.

[66] F. participants, "Imagine all the people and ai in the future of work." ACM SIGMOD blog post, September, 2019.

[67] C. on International Labour Legislation, *Constitution of the international labour organization*, 1919.

[68] J. Camenisch and J. Groth, *Group signatures: Better efficiency and new theoretical aspects*, in *Int. Conf. on Security in Communication Networks*, pp. 120–133, Springer, 2004.

[69] Y. Aumann and Y. Lindell, *Security against covert adversaries: Efficient protocols for realistic adversaries*, in *Theory of Cryptography Conf.*, pp. 137–156, Springer, 2007.

[70] D. Chaum and E. Van Heyst, *Group signatures*, in *Workshop on the Theory and Application of of Cryptographic Techniques*, pp. 257–265, Springer, 1991.

[71] G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik, *A practical and provably secure coalition-resistant group signature scheme*, in *Annual Int. Cryptology Conf.*, pp. 255–270, Springer, 2000.

[72] C. Cachin, *Architecture of the hyperledger blockchain fabric*, in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, vol. 310, 2016.

[73] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, *et. al.*, *All about eve: Execute-verify replication for multi-core servers.*, in *Symposium on Operating systems design and implementation (OSDI)*, vol. 12, pp. 237–250, USENIX Association, 2012.

[74] H.-T. Kung and J. T. Robinson, *On optimistic methods for concurrency control*, *Transactions on Database Systems (TODS)* **6** (1981), no. 2 213–226.

[75] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein, *High performance transactions via early write visibility*, *Proc. of the VLDB Endowment* **10** (2017), no. 5 613–624.

[76] P. A. Bernstein and N. Goodman, *Multiversion concurrency control—theory and algorithms*, *Transactions on Database Systems (TODS)* **8** (1983), no. 4 465–483.

[77] C. Yao, D. Agrawal, P. Chang, G. Chen, B. C. Ooi, W.-F. Wong, and M. Zhang, *Dgcc: A new dependency graph based concurrency control protocol for multicore database systems*, *arXiv preprint arXiv:1503.03642* (2015).

[78] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry, *Sprites and state channels: Payment networks that go faster than lightning*, in *Int. Conf. on Financial Cryptography and Data Security (FC)*, pp. 508–526, Springer, 2019.

[79] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, *et. al.*, *On scaling decentralized blockchains*, in *Int. Conf. on Financial Cryptography and Data Security (FC)*, pp. 106–125, Springer, 2016.

[80] I. A. Seres, L. Gulyás, D. A. Nagy, and P. Burcsi, *Topological analysis of bitcoin's lightning network*, *arXiv preprint arXiv:1901.04972* (2019).

[81] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, *A secure sharding protocol for open blockchains*, in *SIGSAC Conf. on Computer and Communications Security (CCS)*, pp. 17–30, ACM, 2016.

[82] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, *Omniledger: A secure, scale-out, decentralized ledger via sharding*, in *Symposium on Security and Privacy (SP)*, pp. 583–598, IEEE, 2018.

[83] M. Zamani, M. Movahedi, and M. Raykova, *Rapidchain: Scaling blockchain via full sharding*, in *SIGSAC Conf. on Computer and Communications Security*, pp. 931–948, ACM, 2018.

[84] E. Frey and C. Goes, "Cosmos inter-blockchain communication (ibc) protocol." https://cosmos.network. 2018.

[85] D. George and S. Meiklejohn, *Centrally banked cryptocurrencies*, in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[86] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, *Towards scaling blockchain systems via sharding*, in *SIGMOD Int. Conf. on Management of Data*, ACM, 2019.

[87] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, *Attested append-only memory: Making adversaries stick to their word*, in *Operating Systems Review (OSR)*, vol. 41-6, pp. 189–204, ACM SIGOPS, 2007.

[88] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, *Efficient byzantine fault-tolerance*, *Transactions on Computers* **62** (2013), no. 1 16–30.

[89] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, *Ebawa: Efficient byzantine agreement for wide-area networks*, in *Int. Symposium on High Assurance Systems Engineering (HASE)*, pp. 10–19, IEEE, 2010.

[90] L. Lamport, *Fast paxos*, *Distributed Computing* **19** (2006), no. 2 79–103.

[91] F. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal, *Consensus in one communication step*, in *Int. Conf. on Parallel Computing Technologies (PaCT)*, pp. 42–50, Springer, 2001.

[92] C. Curino, E. Jones, Y. Zhang, and S. Madden, *Schism: a workload-driven approach to database replication and partitioning*, *Proc. of the VLDB Endowment* **3** (2010), no. 1-2 48–57.

[93] A. Pavlo, C. Curino, and S. Zdonik, *Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems*, in *SIGMOD Int. Conf. on Management of Data*, pp. 61–72, ACM, 2012.

[94] A. El Abbadi, D. Skeen, and F. Cristian, *An efficient, fault-tolerant protocol for replicated data management*, in *SIGACT-SIGMOD symposium on Principles of database systems*, pp. 215–229, ACM, 1985.

[95] A. El Abbadi and S. Toueg, *Availability in partitioned replicated databases*, in *SIGACT-SIGMOD symposium on Principles of database systems*, pp. 240–251, ACM, 1985.

[96] Q. Zhang, L. Cheng, and R. Boutaba, *Cloud computing: state-of-the-art and research challenges*, *Journal of internet services and applications* **1** (2010), no. 1 7–18.

[97] C. Cachin, I. Keidar, and A. Shraer, *Trusting the cloud*, *Sigact News* **40** (2009), no. 2 81–86.

[98] D. Dobre, P. Viotti, and M. Vukolić, *Hybris: Robust hybrid cloud storage*, in *Symposium on Cloud Computing (SoCC)*, pp. 1–14, ACM, 2014.

[99] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, *Why does the cloud stop computing?: Lessons from hundreds of service outages*, in *Symposium on Cloud Computing (SoCC)*, pp. 1–16, ACM, 2016.

[100] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et. al.*, *A view of cloud computing*, *Communications of the ACM* **53** (2010), no. 4 50–58.

[101] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, *Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services*, in *Symposium on Operating Systems Principles (SOSP)*, pp. 292–308, ACM, 2013.

[102] C. Bǎsescu, C. Cachin, I. Eyal, R. Haas, A. Sorniotti, M. Vukolić, and I. Zachevsky, *Robust data sharing with key-value stores*, in *Int. Conf. on Dependable Systems and Networks (DSN)*, pp. 1–12, IEEE, 2012.

[103] M. A. AlZain, B. Soh, and E. Pardede, *Mcdb: Using multi-clouds to ensure security in cloud computing*, in *Int. Conf. on Dependable, autonomic and secure computing (DASC)*, pp. 784–791, IEEE, 2011.

[104] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, *Separating agreement from execution for byzantine fault tolerant services*, *Operating Systems Review (OSR)* **37** (2003), no. 5 253–267.

[105] M. Castro, R. Rodrigues, and B. Liskov, *Base: Using abstraction to improve fault tolerance*, *Transactions on Computer Systems (TOCS)* **21** (2003), no. 3 236–269.

[106] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, *Hq replication: A hybrid quorum protocol for byzantine fault tolerance*, in *Symposium on Operating systems design and implementation (OSDI)*, pp. 177–190, USENIX Association, 2006.

[107] J. Sousa, E. Alchieri, and A. Bessani, *State machine replication for the masses with bft-smart*, .

[108] D. Ongaro and J. K. Ousterhout, *In search of an understandable consensus algorithm*, in *Annual Technical Conf. (ATC)*, pp. 305–319, USENIX Association, 2014.

[109] "Chain." http://chain.com.

[110] "Hyperledger iroha." https://github.com/hyperledger/iroha.

[111] "Corda." https://github.com/corda/corda.

[112] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, *Resilientdb: Global scale resilient blockchain fabric*, *Proceedings of the VLDB Endowment* **13** (2020), no. 6 868–883.

[113] J. Sousa, A. Bessani, and M. Vukolic, *A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform*, in *Int. Conf. on Dependable Systems and Networks (DSN)*, pp. 51–58, IEEE, 2018.

[114] R. K. Raman, R. Vaculin, M. Hind, S. L. Remy, E. K. Pissadaki, N. K. Bore, R. Daneshvar, B. Srivastava, and K. R. Varshney, *Trusted multi-party computation and verifiable simulations: A scalable blockchain approach*, *arXiv preprint arXiv:1809.08438* (2018).

[115] P. Thakkar, S. Nathan, and B. Vishwanathan, *Performance benchmarking and optimizing hyperledger fabric blockchain platform*, *arXiv preprint arXiv:1805.11390* (2018).

[116] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, *Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second*, in *Int. Conf. on Blockchain and Cryptocurrency (ICBC)*, pp. 455–463, IEEE, 2019.

[117] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, *Blurring the lines between blockchains and database systems: the case of hyperledger fabric*, in *SIGMOD Int. Conf. on Management of Data*, pp. 105–122, ACM, 2019.

[118] P. Thakkar and S. Nathan, *Scaling hyperledger fabric using pipelined execution and sparse peers*, *arXiv preprint arXiv:2003.05113* (2020).

[119] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi, *A transactional perspective on execute-order-validate blockchains*, in *SIGMOD Int. Conf. on Management of Data*, pp. 543–557, ACM, 2020.

[120] A. Miller, I. Bentov, R. Kumaresan, C. Cordi, and P. McCorry, *Sprites and state channels: Payment networks that go faster than lightning*, *arXiv preprint arXiv:1702.05812* (2017).

[121] A. Culwick and D. Metcalf, "The blocknet design specification." https://www.blocknet.co/wp-content/uploads/2018/04/whitepaper.pdf.

[122] V. Buterin, *Chain interoperability*, *R3 Research Paper* (2016).

[123] A. Zamyatin, D. Harz, J. Lind, P. Panayiotou, A. Gervais, and W. J. Knottenbelt, *Xclaim: A framework for blockchain interoperability*, .

[124] "Poa bridge." https://github.com/poanetwork/token-bridge.

[125] "Wanchain: Building super financial markets for the new digital economy." https://www.wanchain.org/files/Wanchain-Whitepaper-EN-version.pdf.

[126] "Fusion whitepaper: An inclusive cryptofinance platform based on blockchain." https://docs.wixstatic.com/ugd/76b9ac_be5c61ff0e3048b3a21456223d542687.pdf.

[127] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, *Enabling blockchain innovations with pegged sidechains*, *URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains* (2014).

[128] J. Dilley, A. Poelstra, J. Wilkins, M. Piekarska, B. Gorlick, and M. Friedenbach, *Strong federations: An interoperable blockchain solution to centralized third-party risks*, *arXiv preprint arXiv:1612.05491* (2016).

[129] J. Poon and V. Buterin, *Plasma: Scalable autonomous smart contracts*, *White paper* (2017).

[130] A. Garoffolo and R. Viglione, *Sidechains: Decoupled consensus between chains*, *arXiv preprint arXiv:1812.05441* (2018).

[131] S. D. Lerner, *Rootstock: Bitcoin powered smart contracts*, 2015.

[132] G. Wood, *Polkadot: Vision for a heterogeneous multi-chain framework*, *White Paper* (2016).

[133] J. Kwon and E. Buchman, *Cosmos: A network of distributed ledgers*, *URL https://cosmos. network/whitepaper* (2016).

[134] A. Churyumov, *Byteball: A decentralized system for storage and transfer of value*, *URL https://byteball. org/Byteball. pdf* (2016).

[135] S. Popov, *The tangle*, *URL https://iota.org/IOTA Whitepaper.pdf* (2018).

[136] L. Baird, *The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance*, *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep* (2016).

[137] K. Karlsson, W. Jiang, S. Wicker, D. Adams, E. Ma, R. van Renesse, and H. Weatherspoon, *Vegvisir: A partition-tolerant blockchain for the internet-of-things*, in *Int. Conf. on Distributed Computing Systems (ICDCS)*, pp. 1150–1158, IEEE, 2018.

[138] Y. Sompolinsky and A. Zohar, *Accelerating bitcoin's transaction processing*, *Fast Money Grows on Trees, Not Chains* (2013).

[139] Y. Lewenberg, Y. Sompolinsky, and A. Zohar, *Inclusive block chain protocols*, in *Int. Conf. on Financial Cryptography and Data Security (FC)*, pp. 528–547, Springer, 2015.

[140] S. D. Lerner, *Dagcoin: a cryptocurrency without blocks*, 2015.

[141] Y. Sompolinsky and A. Zohar, *Phantom: A scalable blockdag protocol.*, 2018.

[142] Y. Sompolinsky, Y. Lewenberg, and A. Zohar, *Spectre: A fast and scalable cryptocurrency protocol.*, *IACR Cryptology ePrint Archive* **2016** (2016) 1159.

[143] I. Bentov, P. Hubácek, T. Moran, and A. Nadler, *Tortoise and hares consensus: the meshcash framework for incentive-compatible, scalable cryptocurrencies.*, *IACR Cryptology ePrint Archive* **2017** (2017) 300.

[144] H. To, C. Shahabi, and L. Xiong, *Privacy-preserving online task assignment in spatial crowdsourcing with untrusted server*, in *Int. Conf. on Data Engineering (ICDE)*, pp. 833–844, IEEE, 2018.

[145] H. To, G. Ghinita, L. Fan, and C. Shahabi, *Differentially private location protection for worker datasets in spatial crowdsourcing*, *Transactions on Mobile Computing* **16** (2016), no. 4 934–949.

[146] B. Liu, L. Chen, X. Zhu, Y. Zhang, C. Zhang, and W. Qiu, *Protecting location privacy in spatial crowdsourcing using encrypted data*, .

[147] A. Liu, Z.-X. Li, G.-F. Liu, K. Zheng, M. Zhang, Q. Li, and X. Zhang, *Privacy-preserving task assignment in spatial crowdsourcing*, .

[148] A. Liu, W. Wang, S. Shang, Q. Li, and X. Zhang, *Efficient task assignment in spatial crowdsourcing with worker and task privacy protection*, *GeoInformatica* **22** (2018), no. 2 335–362.

[149] S. Zhu, Z. Cai, H. Hu, Y. Li, and W. Li, *zkcrowd: a hybrid blockchain-based crowdsourcing platform*, *Transactions on Industrial Informatics* (2019).

[150] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, *Zcash protocol specification*, *GitHub: San Francisco, CA, USA* (2016).

[151] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, *Hawk: The blockchain model of cryptography and privacy-preserving smart contracts*, in *Symposium on security and privacy (SP)*, pp. 839–858, IEEE, 2016.

[152] D. C. Sánchez, *Raziel: Private and verifiable smart contracts on blockchains*, *arXiv preprint arXiv:1807.09484* (2018).

[153] E. Cecchetti, F. Zhang, Y. Ji, A. Kosba, A. Juels, and E. Shi, *Solidus: Confidential distributed ledger transactions via pvorm*, in *SIGSAC Conf. on Computer and Communications Security*, pp. 701–717, ACM, 2017.

[154] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, *et. al.*, *H-store: a high-performance, distributed main memory transaction processing system*, *Proc. of the VLDB Endowment* **1** (2008), no. 2 1496–1499.

[155] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, *Scalable consistency in scatter*, in *Symposium on Operating Systems Principles (SOSP)*, pp. 15–28, ACM, 2011.

[156] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, *Megastore: Providing scalable, highly available storage for interactive services*, in *Conf. on Innovative Data Systems Research (CIDR)*, 2011.

[157] D. Agrawal, A. El Abbadi, and K. Salem, *A taxonomy of partitioned replicated cloud-based database systems.*, *IEEE Data Eng. Bull.* **38** (2015), no. 1 4–9.

[158] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi, *Low-latency multi-datacenter databases using replicated commit*, *Proc. of the VLDB Endowment* **6** (2013), no. 9 661–672.

[159] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, *Enhancing bitcoin security and performance with strong consistency via collective signing*, in *Security Symposium*, pp. 279–296, USENIX Association, 2016.

[160] K. P. Birman, T. A. Joseph, T. Raeuchle, and A. El Abbadi, *Implementing fault-tolerant distributed objects*, *Trans. on Software Engineering* (1985), no. 6 502–508.

[161] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. A. Tewksbury, and V. Kalogeraki, *The eternal system: An architecture for enterprise applications*, in *Int. Enterprise Distributed Object Computing Conf. (EDOC)*, pp. 214–222, IEEE, 1999.

[162] L. Lamport and M. Massa, *Cheap paxos*, in *Int. Conf. on Dependable Systems and Networks (DSN)*, pp. 307–314, IEEE, 2004.

[163] L. Lamport, R. Shostak, and M. Pease, *The byzantine generals problem*, *Transactions on Programming Languages and Systems (TOPLAS)* **4** (1982), no. 3 382–401.

[164] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, *The securering protocols for securing group communication*, in *Hawaii Int. Conf. on System Sciences (HICSS)*, vol. 3, pp. 317–326, IEEE, 1998.

[165] M. K. Reiter, *The rampart toolkit for building high-integrity services*, in *Theory and Practice in Distributed Sys.*, pp. 99–110. Springer, 1995.

[166] Y. J. Song and R. van Renesse, *Bosco: One-step byzantine asynchronous consensus*, in *Int. Symposium on Distributed Computing (DISC)*, pp. 438–450, Springer, 2008.

[167] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, *Fault-scalable byzantine fault-tolerant services*, *Operating Systems Review (OSR)* **39** (2005), no. 5 59–74.

[168] H. P. Reiser and R. Kapitza, *Hypervisor-based efficient proactive recovery*, in *Int. Symposium on Reliable Distributed Systems (SRDS)*, pp. 83–92, IEEE, 2007.

[169] M. Correia, N. F. Neves, and P. Verissimo, *How to tolerate half less one byzantine nodes in practical distributed systems*, in *Int. Symposium on Reliable Distributed Systems (SRDS)*, pp. 174–183, IEEE, 2004.

[170] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, *Sbft: a scalable decentralized trust infrastructure for blockchains*, in *Int. Conf. on Dependable Systems and Networks (DSN)*, pp. 568–580, IEEE/IFIP, 2019.

[171] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, *Hotstuff: Bft consensus with linearity and responsiveness*, in *Symposium on Principles of Distributed Computing (PODC)*, pp. 347–356, ACM, 2019.

[172] D. Boneh, B. Lynn, and H. Shacham, *Short signatures from the weil pairing*, *Journal of cryptology* **17** (2004), no. 4 297–319.

[173] S. Gupta, J. Hellings, and M. Sadoghi, *Scaling blockchain databases through parallel resilient consensus paradigm*, *arXiv preprint arXiv:1911.00837* (2019).

[174] T. Distler, C. Cachin, and R. Kapitza, *Resource-efficient byzantine fault tolerance*, *Transactions on Computers* **65** (2016), no. 9 2807–2819.

[175] Y. Amir, B. Coan, J. Kirsch, and J. Lane, *Prime: Byzantine replication under attack*, *Transactions on Dependable and Secure Computing* **8** (2011), no. 4 564–577.

[176] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, *Making byzantine fault tolerant systems tolerate byzantine faults.*, in *Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 9, pp. 153–168, USENIX Association, 2009.

[177] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, *Spin one's wheels? byzantine fault tolerance with a spinning primary*, in *Int. Symposium on Reliable Distributed Systems (SRDS)*, pp. 135–144, IEEE, 2009.

[178] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, *Rbft: Redundant byzantine fault tolerance*, in *Int. Conf. on Distributed Computing Systems (ICDCS)*, pp. 297–306, IEEE, 2013.

[179] P. Thambidurai, Y.-K. Park, *et. al.*, *Interactive consistency with multiple failure modes*, in *Symposium on Reliable Distributed Systems (SRDS)*, pp. 93–100, IEEE, 1988.

[180] F. J. Meyer and D. K. Pradhan, *Consensus with dual failure modes*, *Transactions on Parallel and Distributed Systems* (1991), no. 2 214–222.

[181] R. M. Kieckhafer and M. H. Azadmanesh, *Reaching approximate agreement with mixed-mode faults*, *Transactions on Parallel and Distributed Systems* **5** (1994), no. 1 53–63.

[182] H.-S. Siu, Y.-H. Chin, and W.-P. Yang, *A note on consensus on dual failure modes*, *Transactions on Parallel and Distributed Systems* **7** (1996), no. 3 225–230.

[183] D. Porto, J. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues, *Visigoth fault tolerance*, in *European Conf. on Computer Systems (EuroSys)*, p. 8, ACM, 2015.

[184] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, *Depsky: dependable and secure storage in a cloud-of-clouds*, *Transactions on Storage (TOS)* **9** (2013), no. 4 12.

[185] A. N. Bessani, R. Mendes, T. Oliveira, N. F. Neves, M. Correia, M. Pasin, and P. Verissimo, *Scfs: A shared cloud-backed file system.*, in *Annual Technical Conf. (ATC)*, pp. 169–180, USENIX Association, 2014.

[186] S.-S. Wang, K.-Q. Yan, and S.-C. Wang, *Achieving efficient agreement within a dual-failure cloud-computing environment*, *Expert Systems with Applications* **38** (2011), no. 1 906–915.