TimeFabric: Trusted Time for Hyperledger Fabric

by

Aritra Mitra

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Management Sciences

Waterloo, Ontario, Canada, 2021

© Aritra Mitra 2021

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Since the advent of Bitcoin in 2008, the interest in blockchain technology has surged tremendously. Numerous applications have been proposed in the field of finance, healthcare, and supply chain over the last decade. And, as the popularity of blockchains continue to rise, blockchain platforms must be enhanced to support new application needs.

We propose one such enhancement that is essential for financial applications and online marketplaces – support for time-based logic. Online marketplaces may need to validate transaction time-stamps against a perishable product's expiry date to prevent sale of expired products. Similarly, financial applications in banking may also need a history of recent transactions for extending credit (like an overdraft) to a customer. As nodes in a blockchain operate in a distributed and trustless setting, it is imperative that they can access a global and trusted clock for verifying deadlines or examining a window of recent activity.

In this thesis, we present a lightweight solution that assigns time-stamps to blocks at transaction validation time, which can be referenced as a global clock by all nodes in the network. Moreover, our solution also maintains a cache reflecting the effects of recent transactions. We implement our solution, called TimeFabric, in Hyperledger Fabric, a popular permissioned blockchain platform, and experimentally demonstrate high throughput and minimal overhead (approximately 3%) of maintaining trusted time. We also demonstrate a 2x performance improvement due to the cache, compared to retrieving transaction histories directly from the ledger.

Acknowledgements

First and foremost I would like to thank my supervisor Dr. Lukasz Golab for introducing me to the field of blockchains and providing me with an opportunity to work in this field. I am truly indebted to him for his continuous support and belief in me without which I would not have succeeded in my master's program.

I am also grateful to Dr. Srinivasan Keshav for the countless hours of expert guidance which enhanced my learning experience immensely.

I would like to thank Christian Gorenflo for being a great mentor, providing some great suggestions and pointing me to the right resources throughout my thesis work. I would also like to thank Rishav Agarwal for supporting my work and being a part of all the group discussions.

Finally , all of this would not have been possible without the unending support of my wife and parents.

Table of Contents

List of Figures vii

List of Tables viii

1 Introduction 1

1.1 Contributions 2

2 Background 4

2.1 Properties of Blockchain 4

2.2 Permissioned and Permissionless blockchains 6

2.3 Hyperledger Fabric Overview 6

2.3.1 The Execute Step 7

2.3.2 The Order Step 8

2.3.3 The Validate Step 8

2.3.4 Time and Account Histories in Hyperledger Fabric 9

3 Our Solution: TimeFabric 11

3.1 Trusted Time 11

3.2 API Implementation and Data Layer Support 14

3.3 TimeFabric Failure Model 16

3.4 Summary of Modifications 16

4 Experiments 18
4.1 Block Time Implementation 19
4.1.1 Committer Overhead 19
4.1.2 Block Time Latency 19
4.2 Time Query Performance 20
4.2.1 Endorser Overhead of GetTimenow() 20
4.2.2 Endorser Overhead of GetStateWindow() 21
4.3 Related Work 22
5 Conclusion 25
References 27

List of Figures

1.1 Different sliding windows seen by nodes with different clocks. 2

2.1 Merkle Tree structure of a block (adapted from [12]) . Here 'H' denotes a hash function. 5

2.2 Transaction flow in Hyperledger Fabric 7

3.1 Transaction flow in Hyperledger Fabric. In TimeFabric, we make changes in steps 2 and 6, shown in red. 12

4.1 Smart Contract for executing time based logic 21

4.2 Endorsement time comparison with standard deviation represented in error bars 22

4.3 Endorsement time for window queries 23

List of Tables

4.1 End to End transaction throughput (in transaction per second) for Fabric 1.4 and TimeFabric. TimeFabric shows approx. 3% overhead against Fabric 1.4 19

4.2 Block time latency for various block sizes 20

Chapter 1

Introduction

Blockchain systems have received substantial interest due to their ability to maintain a trusted transaction log in a decentralized environment without a trusted third party. The earliest platform, Bitcoin [1], allowed the exchange of digital currency among peers in a distributed network. Ethereum [2] then introduced smart contracts, which are Turing-complete stored procedures that expanded the applicability of blockchains beyond cryptocurrencies into finance [3] [4], supply chain management [5] and healthcare [6]. Recently, permissioned systems such as Hyperledger Fabric [7] have been proposed for enterprise settings in which only authenticated entities participate in the network.

As permissioned blockchains gain traction in enterprise settings, blockchain systems must be enhanced to support new application needs. In this thesis, we target applications involving time, such as financial transactions and online auctions and marketplaces. For example, assume a decentralized retail setting with a blockchain platform operated by manufacturers, sellers and regulators. The platform must not allow the participating entities to manipulate timestamps in an attempt to sell expired products. Furthermore, in a financial setting, a bank may allow an overdraft (i.e., allow a withdrawal despite insufficient funds) if an account is in good standing based on recent transactions. Thus, access to a sliding window of recent account activity is required when executing these transactions

However, these applications cannot currently be implemented in Hyperledger Fabric for three reasons. First, clients set transaction timestamps, which are not further verified, and are therefore open to manipulation. Second, different nodes in a Fabric network may have different notions of current time, either by using local clocks or consulting external sources (oracles). This makes it potentially impossible to agree on the execution outcome of smart contracts with time-based logic. For example, when processing an overdraft transaction,

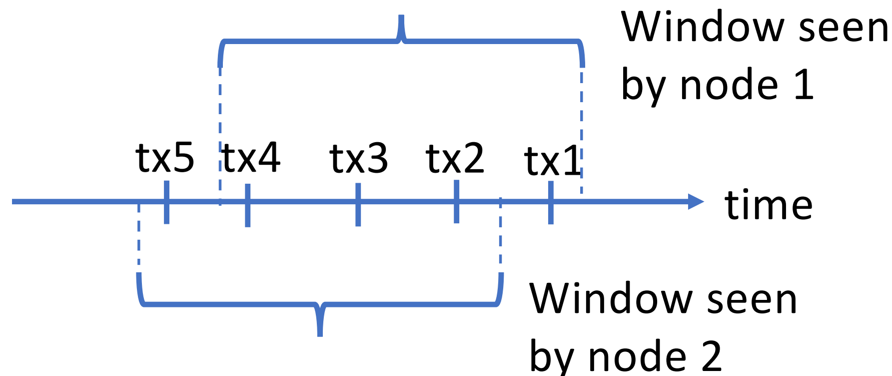

Figure 1.1: Different sliding windows seen by nodes with different clocks.

a node with a late clock will consider a different window of recent activity than nodes with up-to-date clocks. We show an example in Fig. 1.1, with two nodes and five recent transactions. Node 1 has an up-to-date clock and its window contains transactions tx1 through tx4. Node 2's clock is late, and, as a result, it considers a different window, with transactions tx2 through tx5. Another permissioned system - Corda [6] suffers from the same drawback where transactions are time-stamped by a consortium of *notaries*, who do not have synchronized clocks. Thus transactions end up with a time-range rather than a precise time value.

Finally, Fabric uses an account-based data model, in which nodes maintain a *state database* with the current state (e.g., balance) of each account. This makes it easy to verify if an account has sufficient balance to make a purchase. However, if a transaction needs to examine the history of an account, it must extract individual transactions from the ledger, which is expensive. Quorum (another permissioned platform - based on the Ethereum protocol) also maintains an account-based data model due to which access to historical states is expensive.

1.1 Contributions

To mitigate these issues and facilitate smart contracts with time-based logic, we make the following contributions in the context of Hyperledger Fabric:

1. *Trusted time for time-based transactions:* We propose a lightweight solution that enables the Fabric network to validate client-assigned transaction timestamps, and

assigns a tamper-resistant timestamp to each block. Block timestamps can then be used by the network to deterministically execute time-based smart contracts. More importantly, our solution leverages the trust within Fabric network and does not rely on external oracles for block time-stamping.

2. *Data layer support for time-based transactions:* We extend the Fabric state database to store a sliding window of recent states, effectively maintaining a cache reflecting the effects of recent transactions. This modification enables sliding window queries without extracting transactions from the ledger.

3. *Implementation and experimental evaluation:* We implement our solution, called TimeFabric, in Fabric 1.4, and experimentally verify that the overhead of maintaining trusted time is low (under 3%)[1] and that our cache reduces the time to retrieve a sliding window of recent history by a factor of two.

Notably, we make minimal changes to Fabric's transaction processing methodology and we preserve Fabric's modular design, which allows different consensus algorithms to be plugged in without affecting transaction execution.

The remainder of the thesis is organized as follows. Chapter 2 provides background information on blockchains, including an overview of Hyperledger Fabric. Chapter 3 provides a detailed discussion of our solution. In Chapter 4 we present the experiments and provide an overview of existing work. Finally Chapter 5 concludes the thesis with directions for future work.

[1]For a system exhibiting a throughput of 3000 tps (transactions per second), implementing of our solution will result in processing 100 fewer transactions per second.

Chapter 2

Background

In this chapter, we first describe some fundamental properties of blockhains (Sec. 2.1). We then discuss two important categories of blockchain systems: permissioned and permissionless blockchains (Sec 2.2). Finally, we provide an overview of Hyperledger Fabric (a permissioned blockchain system), which is specific to our contributions (Sec 2.3).

2.1 Properties of Blockchain

A blockchain network consists of a set of decentralized nodes that can execute transactions independently. As there is no hierarchy among nodes, they may behave arbitrarily, and hence it is important for them to abide by a set of protocols set within the network. The protocols imposed within the network ensure that only valid transactions are persisted in the blockchain. Furthermore, once validated, all transactions are stored in batches (known as blocks) where each block is linked to the previous block using a cryptographic hash - thus, the term blockchain is coined. The utilization of distributed protocols and cryptographic techniques in blockchains ensure two fundamental properties: *Trust* and *Immutability*:

- **Trust :** A key feature of blockchains is that nodes need not trust one another for executing transactions. As nodes operate independently, they may suffer from byzantine faults or fail-stop faults. In byzantine faults, a node behaves arbitrarily by equivocation i.e., sending different responses to different replicas or performing an incorrect computation. Such failures can tolerate f faulty nodes in the presence of at least 3f+1 nodes [14]. In fail-stop faults, a node may be honest but may stop

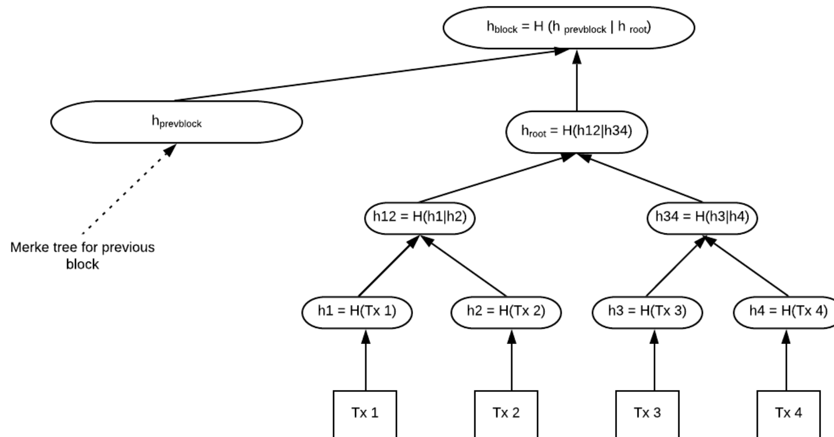

Figure 2.1: Merkle Tree structure of a block (adapted from [12]) . Here 'H' denotes a hash function.

working due to a network breakage or a crash. To mitigate such faults, 2f+1 nodes need to communicate in order to overcome f faulty nodes [22]. Hence, nodes cannot be trusted and specific distributed protocols are necessary to maintain a faultless network. Such protocols, commonly known as consensus protocols, mainly determine the exact order of transactions in a blockchain. Various consensus protocols are used in blockchain systems, with some of the notable ones being - Proof of Work, Proof of Stake, and Proof Of Authority. Bitcoin -the earliest blockchain system - uses a *Proof Of Work* consensus protocol where nodes compete to solve a cryptographic puzzle for building the next batch of transactions (or block). This activity (also known as *mining*) ensures that nodes can operate in a trustless setting.

- **Immutability:** Another essential property of blockchains is *immutability* which prevents tampering of historical records. As transactions are stored in *blocks*, each block computes a root hash from the hash of individual transactions (in that block) using a Merkle tree structure. The root hash is then combined with the hash of the previous block to produce a new block hash, as shown in Fig 2.1. This mechanism of utilizing cryptographic hashes to store records ensures that a node cannot tamper with existing transactions in the blockchain.

  Many blockchain platforms maintain a world state (which is the most updated state of the ledger) for executing transactions. For a financial application exchanging money, the world state may reflect the most recent individual account balances.

The advantage of maintaining a world state is that nodes do not have to recompute balances from historical transactions every time a transaction is requested, resulting in faster execution. It is important to note, that the world state can be derived by replaying historical transactions (from blocks) any time and any change to the world state can be easily recognized and corrected.

2.2 Permissioned and Permissionless blockchains

The earliest blockchain platform - Bitcoin - allows any node to join or leave the network. Network membership is not restricted and all nodes have read and write access to the blockchain, thus guaranteeing data transparency. Such platforms are categorized as public or permissionless blockchains. As the network is open to public and all nodes have equal rights, expensive consensus protocols are employed to prevent byzantine behaviour by nodes which impacts the performance of these platforms.

To mitigate this problem, some blockchains restrict network participation by allowing only authenticated entities to join the network. Membership is managed by a central authority known as the membership service provider (MSP). Such platforms are categorized as private or permissioned blockchains. Although these platforms are not fully decentralized, they bring accountability to the actions of participating nodes and thus reduce the probability of byzantine behaviour. Some examples of permissioned blockchains are Hyperledger Fabric and Corda. Permissioned platforms are commonly used in enterprise collaborations, however, the authenticated entities do not have to fully trust each other.

Public blockchains such as Bitcoin and Ethereum follow an *Order-Execute* (OE) transaction model. Transactions are first ordered using a protocol such as Proof of Work, and then are executed sequentially by each node. In contrast, Fabric follows an *Execute-Order-Validate* (EOV) model, alternatively referred to a *Simulate-Order-Validate-Commit* model [23], in which transactions are executed in parallel in a sandboxed environment, ordered, and validated before being committed to the ledger. We explain the details below, and we summarize the transaction processing workflow in Fig. 2.2.

2.3 Hyperledger Fabric Overview

Entities participating in a Fabric network are called nodes and can be categorized as *peers* and *orderers*. Peers execute smart contracts, called chaincode in Fabric. Orderers,

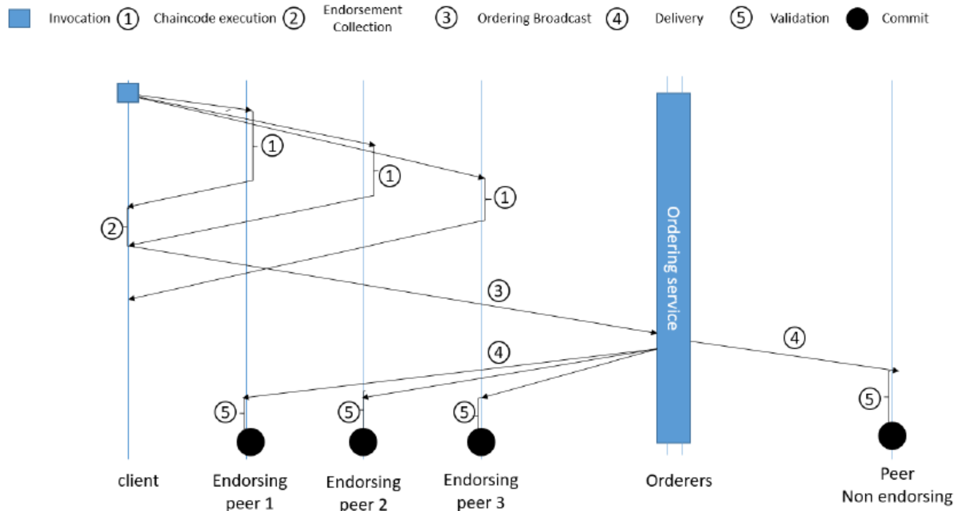

Figure 2.2: Transaction flow in Hyperledger Fabric

collectively referred to as the *ordering service*, are responsible for transaction ordering and creation of blocks. Each peer maintains a local copy of the ledger as well as a *state database* (LevelDB by default), which is a key-value representation of the current state of the ledger. A record in the state database contains three pieces of information: a key (e.g., account ID), a value (e.g., the current account balance), and a version number. The state database is used during transaction processing; for example, it can be used to determine whether a given account has a sufficient balance to make a purchase without having to retrieve all the transactions for this account from the ledger. Whenever a transaction (i.e., the execution of a smart contract) is committed to the ledger, the effects of the transaction are persisted in the state database. That is, the new values are written to the database and the corresponding version numbers are incremented. Old versions are eventually discarded from the state database by a background garbage-collection process.

Fabric's architecture is detailed in [2] and its Execute-Order-Validate transaction processing protocol proceeds as follows.

2.3.1 The Execute Step

Client applications submit *transaction proposals* to the Fabric network (step 1 in Fig. 2.2). A subsets of peers, called *endorsers*, concurrently simulate the execution of the correspond-

ing smart contracts in a sandboxed environment, i.e., without persisting the effects in the state database. Three such endorsers are shown in Fig. 2.2. Each endorser then sends a response to the client application if the corresponding smart contract was successfully simulated. The response contains the endorser's signature as well as a *read set* and *write set*, which consist of the keys and their version numbers that were read from the state database, and keys (plus their new values) that were updated, respectively, during the simulated execution of the transaction proposal. The write sets thus capture the effects of transactions that must eventually be reflected in the state database.

2.3.2 The Order Step

An endorsement policy, set by the network, specifies the number of endorsements a transaction needs. After a client application receives the required number of endorser responses (step 3 in Fig. 2.2), it sends the transaction proposal, with endorsements attached, to the orderers (step 4 in Fig. 3.1). The orderer nodes run a consensus protocol to determine the order of transactions received from various client applications. Fabric allows various consensus protocols to be plugged into the ordering stage (e.g., Kafka or Raft), with crash-fault (rather than Byzantine fault) tolerant protocols used in practice since the participants in a permissioned blockchain system are known and incentivized to behave honestly. Transactions, with endorsements attached, are segmented into blocks; a block is created if the maximum number of transactions per block (set by the application) arrive or if a block timeout period is exceeded (the default block timeout in Fabric is two seconds). Blocks are then disseminated to the peers (step 5 in Fig. 2.2). Note that orderers are only responsible for ordering the transactions and batching them into blocks; they do not examine transaction contents for correctness or validity.

2.3.3 The Validate Step

Finally, peers serially *validate* (endorser signatures and read-write sets of) transactions in a block, and, upon successful validation, persist the effects of transactions in the local state database and append the block to the local copy of the ledger (step 6 in Fig. 2.2; committer peers are the non-endorsing peers). Transaction validation succeeds if the version numbers of the keys in the transaction read sets are the same as the current version numbers in the state database.

Validation is required because transaction proposals are executed in parallel during the initial Execute stage, and thus transaction conflicts may arise. For example, suppose two

client transactions wish to withdraw money from the same account, with key 123, whose current version number in the world state is 100. Suppose no other transactions in this block touch this key. The read sets of both of these transactions include key 123 with version number 100. During validation, the first of these transactions will be committed because key 123 still has version number 100 in the state database (it has not been modified by any other transaction from this block). After the first transaction is committed, the new version of key 123 will be 101. Now, the second transaction fails because the version number of key 123 in its read set is 100, but it is 101 in the state database. Failed (or aborted) transactions are marked as such and remain in the block.

Transaction validation prevents read-write and write-write conflicts. In a given block, at most one transaction can write to a key, and if another transaction only reads this key without writing to it, this transaction must be ordered before the one that writes to this key (otherwise, the version numbers will not match). This prevents double-spending, but may also prevent legitimate transactions from being committed. In the above example, even if there is sufficient balance in account 123 for both withdrawals, only the first transaction will succeed. The second transaction will need to be re-submitted by the client application for re-endorsement, and will be put in a new block for validation.

Note that once the transactions in a block have been ordered, they are sequentially validated by each peer in the same order without re-executing the smart contract. As a result, each peer makes the same commit (or not) decisions, and thus each peer stores the same version of the ledger and the state database. On successful validation, effects of transactions are persisted in the state database.

2.3.4 Time and Account Histories in Hyperledger Fabric

We now outline existing Fabric functionality related to transaction timestamps and transaction histories. Clients can set transaction timestamps when creating transaction proposals, which are recorded in the transaction header and ultimately appear in the blockchain. Fabric exposes a method *GetTxTimestamp(transaction_id)* for chaincode to access transaction timestamps. However, transaction timestamps are not endorsed during the execute step or verified during the validate step.

Furthermore, chaincode can call *GetHistoryForKey(key)* to obtain a history of *all* values for a given key, along with the transaction timestamps corresponding to each update (querying a specific time window is not supported). This is done by consulting an index that points to (the blocks containing) transactions that have modified a given key. These

transactions are then retrieved from the blockchain to compute the history, which is expensive. This index is stored in the state database, in addition to the keys and their latest values.

Chapter 3

Our Solution: TimeFabric

As we noted earlier, transaction timestamps are not endorsed in Fabric. Furthermore, different peers may have different notions of current time, either by using local clocks or consulting external sources. As a result, if a smart contract requires access to a time window of recent transactions for a given key, then different endorsers may access different time windows. In this section, we describe how our solution, TimeFabric, addresses these issues, starting with the notion of time (Sec. 3.1), followed by API implementation details and data layer support for transactions involving time (Sec. 3.2), a discussion of TimeFabric's failure model compared to the underlying Fabric (Sec. 3.3), and concluding with a summary of the required modifications to Fabric (Sec. 3.4). Our design goals are:

1. To provide a trusted and consistent time reference for Fabric peers
2. To process transactions that reference this trusted time efficiently, with minimal overhead
3. To preserve Fabric's modular design that separates endorsement and validation from ordering

3.1 Trusted Time

We begin with design goal #1 to provide a trusted and consistent time reference for all peers. Maintaining a trusted time reference in Fabric may be the responsibility of peers

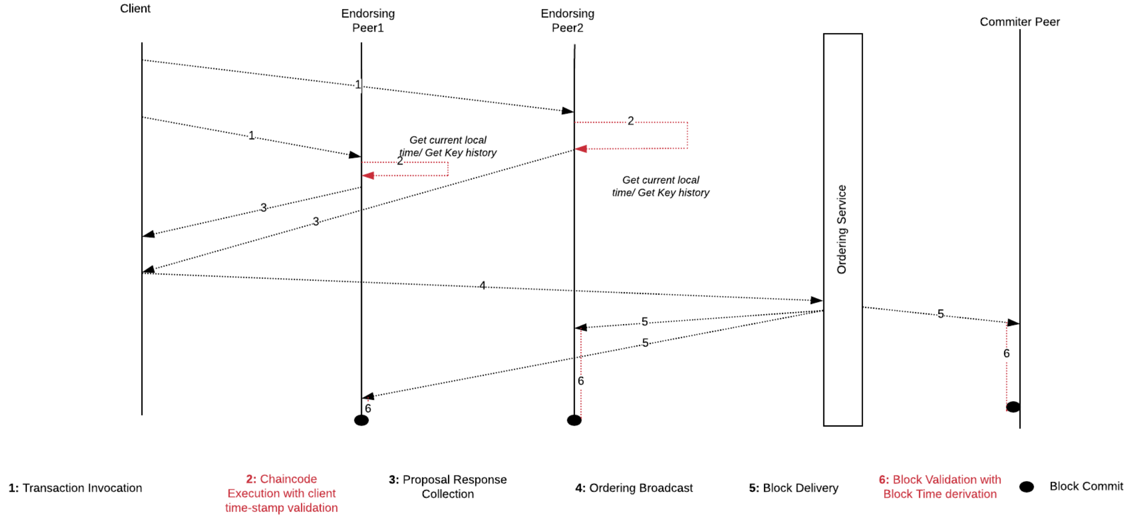

Figure 3.1: Transaction flow in Hyperledger Fabric. In TimeFabric, we make changes in steps 2 and 6, shown in red.

or orderers. However, Fabric's modular design suggests that orderers should only be responsible for ordering transactions. To maintain compatibility with various plug-and-play consensus algorithms for the ordering step (design goal #3), we turn to the other transaction processing steps. Our solution consists of the following modifications to Fabric, shown in red in Fig. 3.1, plus a modification of the data layer to maintain a cache of recent histories that will be discussed in Sec. 3.2.

1. **Validation of transaction timestamps during endorsement**. We modify the execute step such that endorsing peers endorse a transaction during chaincode execution only if the transaction timestamp is within δ time units of the current trusted time (this will be defined shortly). Thus, transactions with timestamps too far into the past or the future will not be endorsed. The value of δ can be set in the corresponding chaincode, and we will discuss setting the value of δ shortly.

2. **Assigning trusted block timestamps**. We modify the validate step such that validating peers assign *block timestamps*. In particular, they set the block timestamp to be the most recent transaction timestamp within the block (among transactions that have been validated and have not been aborted), unless this timestamp is older than the timestamp of the previous block, in which case the timestamp of the new

block equals the timestamp of the previous block plus a small constant ϵ (in our implementation, $\epsilon = 1$ millisecond). To do this, when validating transactions within a block, each peer must keep track of the latest transaction timestamp seen, and finally append it to the block. We extend the block metadata structure to include a timestamp field, which becomes part of the blockchain.

3. **A heartbeat mechanism**[1]. Fabric orderers disseminate a new block when it is full (contains the maximum number of transactions) or if it contains at least one transaction and no other transaction has arrived for two seconds (the default timeout period). However, if no transactions at all arrive for at least two seconds, then no block will be created, and therefore block time will not advance. Suppose no transactions arrive for 60 seconds. Then, when a transaction arrives, its timestamp would be 60 seconds into the future relative to the timestamp of the latest committed block. To ensure that our trusted time moves forward even during periods of inactivity, we set up a "dummy" client that sends one mock transaction every two seconds. This transaction updates a reserved "dummy" key with a random value, and its transaction timestamp equals the local time of the client.

Time thus advances one block at a time, based on *validated* transaction timestamps, giving every peer a common time reference. At any point, the current trusted time, or *block time*, as required during endorsement, is the time of the latest block that has been committed to the ledger. The block time is used for any reference to time in a smart contract.

When setting an appropriate value for δ, note that block time may be over two seconds in the past in the worst case, if no new transactions have arrived and a heartbeat transaction was just generated. To account for this delay and network delays between clients and the Fabric/TimeFabric network, we set δ to four seconds, or twice the timeout period.

TimeFabric uses one timestamp per block rather than one timestamp per transaction for several reasons. The first is efficiency: in general, obtaining consensus on a value in a decentralized setting is expensive. The second is to ensure a monotonically increasing time reference. Recall that we do not modify the ordering step and that orderers do not inspect transaction details when deciding on the transaction order within a block. As a result, transactions within a block may not necessarily be ordered by transaction timestamp. Finally, we observe that block timetamps alone already produce totally ordered key histories because Fabric's validation step ensures that a key can be updated at most once per block.

[1] Implementing the heartbeat mechanism does not need any platform changes.

Given our notion of trusted time, we add three methods to the Fabric API that are accessible to smart contracts (implementation details will follow in Sec. 3.2).

1. *GetTimenow()* returns the current block time.
2. *GetHistoryRangeForKey(key, start, end)* returns a history of values for a given key with block timestamps in the interval $[start, end]$.
3. *GetStateWindow(key,window_length)* is a wrapper over *GetTimenow()* and *GetHistoryRangeForKey()*. It obtains a history of values for a given key with block timestamps in the interval $[current_time - window_length, current_time]$.

GetTimenow() is meant to be used when endorsing transaction timestamps, which can then be used during smart contract execution, e.g., to verify if deadlines are met. *GetWindowForKey()* is meant to be used during smart contract execution to retrieve recent histories.

3.2 API Implementation and Data Layer Support

We now discuss changes in the data layer to speed up the new methods discussed above (design goal #2).

We start with *GetTimenow()*. The implementation is simple: we extract the timestamp from the latest block in the ledger. We considered caching the block time at the endorsers, but the performance gains were minimal[2] since the latest block is already cached in memory by Fabric.

Next, we discuss *GetHistoryRangeForKey()*. A naive implementation, using existing Fabric functionality, is to call *GetHistoryForKey()* to obtain a complete history of values for a given key. We then look up the blocks of the transactions that modified this key, and we retain only those transactions that are in blocks whose timestamps are within the desired time range. This is expensive, due to the need to access the blockchain to retrieve the complete history.

Our solution in TimeFabric is to maintain a cache storing updates from recent transactions. To do this, we add a *cache database* to each peer. Each record in the cache

[2] We compare our cache implementation i.e. storing and retrieving the time-stamp from memory against retrieving the time-stamp from the latest block and observe a performance gain of less than .01%.

database is a key-value pair. The key is a concatenation of the corresponding key in the state database and the block timestamp of the transaction that updated the key. The value is the corresponding updated value. For example, suppose that key 123 is updated to have value 50 by a transaction belonging to a block with Unix timestamp 1607994614. The corresponding key-value pair in the state database is $(123, 50)$, plus the version number. The key-value pair in the cache database is $(123 : 1607994614, 50)$.

To populate the cache database, we make another modification in the validate step. In addition to writing key-value pairs to the state database, we require the validating peers to write key-value pairs (with timestamps concatenated to the key) to the cache database. *GetHistoryRangeForKey()* can then be answered via a range query on the key against the cache database. For example, a query for the history of key 123 between Unix timestamps 1600000000 and 1607994614 becomes a range query against the cache database for keys in the range from $123 : 1600000000$ to $123 : 1607994614$.

We note a subtle but important issue related to read set validation. Assume a transaction that fetches a window of recent account history, including the current balance, for account 123, and updates the account balance if the account history satisfies some condition. This transaction can use *GetHistoryRangeForKey()*, which fetches a window of recent history of key 123 from the cache database. However, we wish to re-use Fabric's transaction conflict logic during transaction validation. For example, this transaction should not be committed if another transaction from the same block had updated account 123. To identify these types of conflicts, we modify *GetHistoryRangeForKey()* to also fetch the latest key-value pair from the state database (in addition to fetching the history of this key from the cache database). Next, only the keys read from the state database are validated; records in the cache database are never updated (only new keys are added), so their version numbers are always '1' and do not need to be validated. However, the transaction's read set contains all keys read from the state database and the cache database for auditability (recall that the read and write sets becomes part of the blockchain).

As mentioned above, there is one important distinction between the state database and the cache database. In the former, values of existing keys are updated (and version numbers are incremented) since only the most recent value needs to be stored. In contrast, the cache database is append only: an update of the state database results in a new key added to the cache database since keys in the cache database include block timestamps. Thus, if not maintained, the cache database will grow indefinitely.

To avoid this problem, we borrow a common solution, similar to the calendar queue, used by data stream management systems to maintain sliding windows [10]. The idea is to partition, or shard, the cache database by time, and, instead of deleting individual records

over time, periodically drop the oldest part. For example, suppose that an application requires a 7-day history. Peers may partition the cache database by day. Every day, a new part is added to store new records generated that day, and the oldest day is dropped. The window length and the number of shards are parameters that may be decided by the Fabric network along with other blockchain configuration parameters. In our implementation, the partitioned cache database consists of separate instances of hashmaps, and *GetHistoryRangeForKey()* is handled by issuing a range query against each instance.

3.3 TimeFabric Failure Model

In this section, we discuss the impact of our modifications on the failure model of the system. In Fabric, the membership service that authenticates the participating entities must be fault-tolerant, and this does not change in TimeFabric. Similarly, we do not change Fabric's ability to plug in various ordering algorithms, which can be crash-fault or Byzantine-fault tolerant, as desired by the application.

We also retain Fabric's endorsement policies, specifying the number of endorser responses required by a client transaction. Having to collect multiple endorser responses prevents collusion between client applications and an endorser, and this extends to TimeFabric's endorsement of transaction timestamps.

Furthermore, the ledger is replicated among the peers, each block contains a hash pointer to the previous block to ensure immutability, and every peer independently validates transactions and appends new blocks to the chain, as in Fabric. TimeFabric adds block timestamping to each peer's responsibilities, resulting in the same failure model: any inconsistencies at one peer can be easily detected by comparing other peers' ledgers. In contrast to Fabric, TimeFabric peers also maintain a cache database. In case of a crash fault, a peer can rebuild its cache database by unpacking transactions from recent blocks. (Similarly, a peer (in Fabric and TimeFabric) recovering from a failure can rebuild its state database from the ledger).

Finally, as for the mock client that implements the heartbeat mechanism, we install one such client at each endorser for crash-fault tolerance.

3.4 Summary of Modifications

We conclude the description of our solution with a summary of the required modifications to Fabric version 1.4, on which TimeFabric is based. In addition to the mock client for

heartbeats, as discussed earlier, the following modifications are required.

In the validation step, peers have two additional tasks:

1. In Fabric, transactions are validated by committer peers once a block is received from the ordering service. Each transaction in the block is unpacked and validated by the committer peer in parallel using multiple Go routines. At this stage, we additionally identify the maximum timestamp across the valid transactions, and we insert this timestamp into the block metadata. Using the maximum value of time-stamps as the block time ensures that our global clock is close to the physical world time.

2. We add a cache database that must by maintained by the peers over time (i.e., periodically create new shards and drop old shards). We modify the block commitment stage to add this new database (which is a hashmap in our implementation). Each transaction in a block is unpacked to extract the write-sets. We then compute new keys to be written to the cache database by concatenating the timestamp to the original key, and we insert this key-value pair to the cache database.

In the execute step, endorsing peers have one additional task: validate transaction timestamps by comparing them to the current block time (via the new method *GetTimenow()*). We implemented this method in the Fabric RPC server by querying the ledger to retrieve the latest block, and extract the block timestamp from the block metadata.

Additionally, smart contracts have access to recent histories via *GetStateWindow()*, which queries the cache database (and the state database for the latest value).

Finally, there are no modifications to the ordering step.

Chapter 4

Experiments

In this chapter, we experimentally evaluate TimeFabric, which we implemented in Fabric version 1.4 (our modifications remain compatible with the recent release of version 2 since we do not change Fabric's modular design). We use six local servers connected through a 1Gbit/s switch. Each server is equipped with two Intel Xeon CPU E5-2620 v2 processors at 2.10 GHz, and 64 GB of RAM. Our experiments are conducted using Fabric binaries and we only use docker containers for the chaincode runtime environment. All tests are conducted with non-conflicting and valid transactions to ensure that all transactions go through the entire life-cycle (endorsement, ordering, validation and commit) without being aborted. This helps us to evaluate the worst-case performance of the system in terms of transaction throughput.

Our experiments have two goals: *1)* evaluating our implementation of trusted block time and *2)* evaluating the performance of the new APIs to obtain the current block time and a recent history for a given key. To evaluate the implementation of block time, we measure the overhead introduced by our changes to the Fabric transaction processing lifecycle, specifically, the overhead incurred by committer peers. To isolate this overhead, we send pre-endorsed transactions to the orderer and measure the transaction throughput at committer peers. We also measure the latency of the block time, i.e., how far back it is compared to the wall clock, for various block sizes. To evaluate the performance of the new API, we measure the runtime overhead of our new method *GetTimenow()*, and we compare our method *GetStateWindow()* to Fabric's *GetHistoryForKey()*.

Table 4.1: End to End transaction throughput (in transaction per second) for Fabric 1.4 and TimeFabric. TimeFabric shows approx. 3% overhead against Fabric 1.4

Fabric 1.4	**TimeFabric**
2927 ± 136	2831 ± 196

4.1 Block Time Implementation

4.1.1 Committer Overhead

In this experiment, we compare the transaction throughput at the committer peer for Fabric 1.4 and TimeFabric. We use a single endorser and a single committer peer, a solo orderer, and four client machines that generate transaction proposals[1]. We first send 25000 transaction proposals from each client to the endorser and obtain the proposal responses. We then set up 25 threads in each client (totaling 100 threads) to send a total of 100000 transactions to the orderer. Subsequently, we measure the total time by the committer peer to commit all the blocks to the ledger and then derive the throughput. Following prior work on improving the throughput of Fabric [11], we set the block size to 100. We conduct 30 runs and report the mean throughput and the standard deviation in Table 4.1. This experiment shows that our changes only add about 3% overhead to the block validation and commit process.

4.1.2 Block Time Latency

In this experiment, we record the time difference between an endorser's local clock and the block time, i.e., the time assigned to the latest committed block. We expect lower latencies for smaller block sizes, with size corresponding to the number of transactions per block. Since we want to measure the latency from the point of view of a single endorsing peer, we use a single peer with a solo orderer and one client node. We execute a smart contract that calls our method, *GetTimenow()*, to obtain the current block time. The smart contract then calculates the difference between its local clock and the block time, and writes this difference to a new key in the state database. That is, the sole purpose of this smart contract is to record block time latencies. We execute 25000 such transactions for varying

[1] The experimental setup consists of nodes from a local cluster. We do not measure network lag in a geo-distributed cluster as our changes do not introduce any new network communication among nodes.

Table 4.2: Block time latency for various block sizes

Block Time Latency					
Block Size	**50**	**75**	**100**	**125**	**150**
Mean (ms)	97	186	192	244	480
Median (ms)	90	131	175	223	285
Range (ms)	51-2343	85-2445	103-2591	104-2603	164-2670

block sizes, and we compute the mean and median latencies as well as the latency range, as seen by these transactions.

We show the results in Table 4.2. We observe that mean latency increases with the block size. However, as we noted earlier, prior work observed the highest throughput at a block size of 100. Given this block size, the mean block time latency is under 200 milliseconds. Note that these results correspond to a scenario in which transactions arrive continuously and blocks fill up naturally, without the need for heartbeat transactions to create new blocks. As we discussed earlier, if transactions stop arriving, then the block time latency increases to just over two seconds, which is the timeout period plus the time to commit the block with the heartbeat transaction.

4.2 Time Query Performance

4.2.1 Endorser Overhead of GetTimenow()

In this experiment, we measure the performance of *GetTimenow()* by monitoring the endorsement time for transactions on a single peer. For this, we implement a smart contract (similar to the one shown in Figure 4.1) that corresponds to a retail purchase transaction for a perishable product. The transaction is endorsed if its timestamp is earlier than product expiry date; if so, the chaincode additionally decrements the available quantity of the product, which involves one key read and one key write. In TimeFabric, the chaincode calls *GetTimenow()* to obtain the time. In Fabric, the chaincode simply obtains the local time at the endorser. We send a series of transactions to the endorsing peer from a single client

```
contract Purchase {
    method CheckExpiry(product, quantity) {
      if (product.ExpiryDateTime < GetTimenow()) {
           return "Transaction Failed";
      }
      product.Qty = product.Qty - quantity;
      return "Transaction Successful";

 } }
```

Figure 4.1: Smart Contract for executing time based logic

and calculate the total time for obtaining all the responses. We repeat this experiment by varying the number of transactions and recording the endorsement time.

We show the results in Figure 4.2, which reveals that the performance overhead of *GetTimenow()* is statistically insignificant.

4.2.2 Endorser Overhead of GetStateWindow()

We compare the performance of *GetStateWindow()* in our implementation against *GetHistoryForKey()* in Fabric 1.4. Since Fabric fetches key histories directly from blocks, we expect a performance improvement in our implementation that uses the cache database for recent history. We start by loading the state database with 500 keys, and then each key is updated between 10 and 200 times, depending on the experiment. The chaincode for this experiment corresponds to a financial overdraft transaction: it reads the full history of the key (between 10 and 200 values, depending on the experiment, to simulate different window lengths) and writes a new value for this key if the history shows that this account has maintained some minimum balance. We use a single client to execute the transactions for all 500 keys and we record the total time for collecting all proposal responses from a single endorser.

We show the results in Figure 4.3. The performance of Fabric's *GetHistoryForKey()*

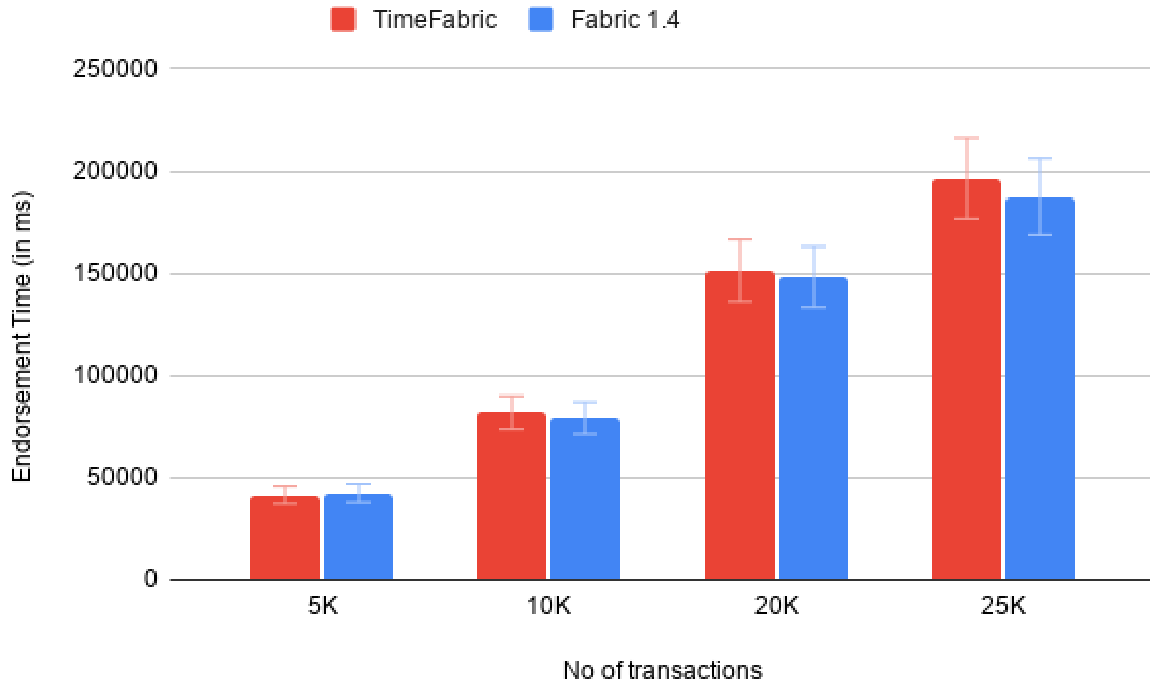

Figure 4.2: Endorsement time comparison with standard deviation represented in error bars

degrades as the window length increases since there is more history to retrieve. On the other hand, the running time of our implementation of *GetStateWindow()* increases only slightly as the window length increases. For a window of 200 historical values, TimeFabric is nearly twice as fast as Fabric 1.4.

4.3 Related Work

Hyperledger Fabric is actively being developed and various performance optimizations have recently been proposed, including adding parallelism and caching to the transaction processing pipeline[11, 23]. Our solution is compatible with these optimizations since our modifications leave Fabric's modular structure intact.

Perhaps the closest work to ours is that of Zan and Xu [31], which proposes to add a

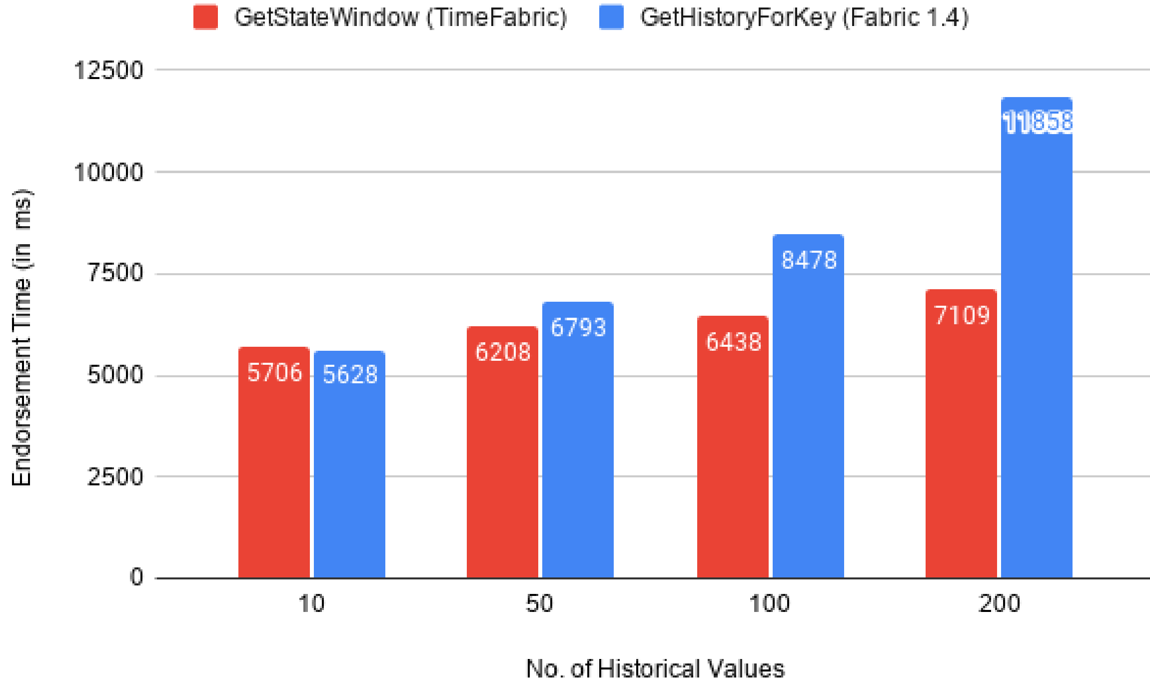

Figure 4.3: Endorsement time for window queries

separate global clock node to Fabric, whose purpose is to periodically synchronize the local clocks of endorsers, orderers and committers during the transaction lifecycle. Although this approach can improve the accuracy of local clocks, it cannot fully synchronize them, as we do using block time. Additionally, our solution goes one step further to ensure that time-related operations such as sliding windows can be done efficiently.

FabricSharp [21] is a proposal to add timestamp-based optimistic concurrency control to Fabric. However, instead of using physical time, FabricSharp uses block sequence numbers and it does not solve our problem of maintaining trusted time for use by smart contracts. This precludes, for example, applications that depend on a time window.

LineageChain [20] extends Fabric by exposing *provenance* information, i.e., key histories, to smart contracts. For efficiency, LineageChain maintains an index over the provenance tree[2]. This is conceptually similar to our use of the cache database to speed up

[2]LineageChain uses a closed-source storage layer - ForkBase [28] where historical values are stored in disk as compared to our solution where the history is maintained in memory.

sliding window queries. However, LineageChain does not offer a notion of time and its provenance queries do not support sliding windows.

Next, we review time-related concepts in permissionless blockchains such as Bitcoin and Ethereum. In systems that use Proof of Work for consensus, block timestamps are usually set by the miners when forming new blocks. Ethereum enforces a protocol to not accept a new block if the timestamp provided by the miner is earlier than timestamp of the previous block. Additionally, if a block timestamp is set in future, other mining nodes may not want to build on that block, resulting in forks. Bitcoin's protocol is to not propagate a block whose miner-assigned timestamp is earlier than the median of the previous 11 blocks or more than two hours into the future. We incorporate similar constraints in our solution: block timestamps must be monotonically increasing, and they are based on verified transaction timestamps that cannot be too far in the past or the future.

While protocols exist in permissionless systems to reject blocks with suspicious timestamps, there has also been work describing attacks related to time manipulation [26],[4],[30],[3]. These works highlight vulnerabilities but do not propose solutions, except [25] – in that work, focusing on Bitcoin, a verifier node requests a timestamping authority (TSA) to validate block timestamps. The verifier node unpacks the block header, has the TSA timestamp the block, and includes the hash of the data in a subsequent transaction that is included in the next block. The next block header is again unpacked, timestamped by TSA and returned to the verifier. As a result, any discrepancy in block time can be found by comparing the block time (set by the miner) against the two timestamps obtained from the TSA. Our solution avoids a timestamping authority and instead leverages the additional trust inherent in permissioned blockchains by using client transaction timestamps (properly verified) as a basis of trusted block timestamping.

Finally, other studies such as [18] and [13] argue that block sequence numbers are intrinsic to blockchains and best represent the temporal progression of a blockchain. Reference [18] specifically states that any reference to an external time oracle violates the decentralized property of a blockchain network. Our solution avoids the use of external time oracles, and, again, leverages the additional trust inherent in permissioned systems to assign block timestamps.

Chapter 5

Conclusion

In this thesis, we presented TimeFabric: a solution to enable smart contracts that reference time in the Hyperledger Fabric permissioned blochain system. We addressed two main issues: implementing a trusted and consistent notion of time that may be referenced by Fabric nodes when executing smart contracts, and data layer support to ensure that operations involving time, such as querying a sliding window of recent history, are efficient. Our light-weight solution assigns block timestamps at transaction validation time and maintains a cache with a sliding window of recent transactions. A probable limitation of our solution is the need of periodic hear-beats by using dummy clients to advance the time for low throughput networks. Secondly, our solution is also contingent upon availability of the latest block to all peers for successful endorsements (as in regular Fabric 1.4), though it remains to be seen if widely distributed nodes may result in failed endorsements. Regardless of these foreseeable limitations, experimental results show that our modifications add little overhead to the transaction processing pipeline in Fabric and that time-based smart contracts can be executed efficiently by fetching account histories from the cache.

As outlined in Chapter 1 our work is useful for any application that needs recent historical states for processing a real-time transaction. Some examples have been described from retail and finance domains. Other applications can be in areas like agriculture where fair allocation of resources among competing individuals is critical. For example, a scarce resource like water can be allocated to various participants in a blockchain based community irrigation system [5] based on their historical usage. As competing parties may manipulate time-stamps to consume more resources, a trusted time-stamp is necessary for ensuring a fair allocation. Another probable use case can be for fraud detection in credit cards where historical transaction patterns are analysed in real-time to approve or reject a new transaction [9]. In future work, we plan to investigate new applications that can leverage

trusted time and access to sliding windows of account histories enabled by TimeFabric, in areas such as finance, retail, supply chains and online auctions.

References

[1] Rishav Raj Agarwal, Dhruv Kumar, Lukasz Golab, and Srinivasan Keshav. Consentio: Managing consent to data access using permissioned blockchains. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC*, pages 1–9, 2020.

[2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. *IACR Cryptology ePrint archive*, 2016:1007, 2016.

[4] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. Findel: Secure derivative contracts for ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 453–467. Springer, 2017.

[5] Borja Bordel, Diego Martin, Ramon Alcarria, and Tomás Robles. A blockchain-based water control system for the automatic management of irrigation communities. In *2019 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–2. IEEE, 2019.

[6] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, August*, 1:15, 2016.

[7] Consumer Financial Protection Bureau. Cfpb study of overdraft programs. *A white paper of initial findings*, 2013.

[8] Luisanna Cocco, Andrea Pinna, and Michele Marchesi. Banking on blockchain: Costs savings thanks to the blockchain technology. *Future internet*, 9(3):25, 2017.

[9] Ivo Correia, Fabiana Fournier, and Inna Skarbovsky. The uncertain case of credit card fraud detection. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 181–192, 2015.

[10] Lukasz Golab and M. Tamer Özsu. *Data Stream Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[11] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 455–463. IEEE, 2019.

[12] Gorenflo, Christian. Towards a new generation of permissioned blockchain systems, 2020.

[13] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*, pages 839–858. IEEE, 2016.

[14] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*, pages 203–226. 2019.

[15] Larissa Lee. New kids on the blockchain: How bitcoin's technology could reinvent the stock market. *Hastings Bus. LJ*, 12:81, 2015.

[16] H-J Lenz and Bernhard Thalheim. Olap databases and aggregation functions. In *Proceedings Thirteenth International Conference on Scientific and Statistical Database Management. SSDBM 2001*, pages 91–100. IEEE, 2001.

[17] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2008.

[18] Ricardo Pérez-Marco. Blockchain time and heisenberg uncertainty principle. In *Science and Information Conference*, pages 849–854. Springer, 2018.

[19] Hubert Pun, Jayashankar M Swaminathan, and Pengwen Hou. Blockchain adoption for combating deceptive counterfeits. *Kenan Institute of Private Enterprise Research Paper*, (18-18), 2018.

[20] Pingcheng Ruan, Gang Chen, Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. Fine-grained, secure and efficient data provenance for blockchain. *Proc. VLDB Endow.*, 12(9):975–988, 2019.

[21] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference*, pages 543–557, 2020.

[22] Fred B Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems (TOCS)*, 2(2):145–154, 1984.

[23] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference*, pages 105–122, 2019.

[24] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 1997.

[25] Pawel Szalachowski. (short paper) towards more reliable bitcoin timestamps. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 101–104. IEEE, 2018.

[26] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16, 2018.

[27] Marko Vukolić. Rethinking permissioned blockchains. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, pages 3–7, 2017.

[28] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Wanzeng Fu, Beng Chin Ooi, and Pingcheng Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. *arXiv preprint arXiv:1802.04949*, 2018.

[29] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[30] Karl Wüst and Arthur Gervais. Ethereum eclipse attacks. Technical report, ETH Zurich, 2016.

[31] Chao Zan and Hai-Chuan Xu. A global clock model for the consortium blockchains. In *International Conference on Blockchain and Trustworthy Systems*, pages 71–80. Springer, 2019.