TimeFabric: Trusted Time for Permissioned Blockchains

Aritra Mitra
University of Waterloo, Canada

Christian Gorenflo
University of Waterloo, Canada

Lukasz Golab
University of Waterloo, Canada

S. Keshav
University of Cambridge, UK

Abstract

As the popularity of blockchains continues to rise, blockchain platforms must be enhanced to support new application needs. In this paper, we propose one such enhancement that is essential for financial applications and online marketplaces – support for time-based logic such as verifying deadlines or expiry dates and examining a time window of recent account activity. We present a lightweight solution to reach consensus on the current time without relying on external time oracles. Our solution assigns timestamps to blocks at transaction validation time and maintains a cache reflecting the effects of recent transactions. We implement a proof-of-concept prototype, 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 reconstructing account histories from the ledger.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Permissioned Blockchain, Timestamp, Clock, Sliding Window, Hyerpleger Fabric

Digital Object Identifier 10.4230/OASIcs.FAB.2021.4

Supplementary Material *Software (Source Code)*:
https://github.com/aritramitra14/fabric/tree/timefabric

1 Introduction

Blockchain systems have received substantial interest due to their ability to maintain a trusted transaction log in a decentralized environment. The earliest platform, Bitcoin [12], allowed the exchange of digital currency among peers in a distributed network. Ethereum [20] then introduced smart contracts, which are Turing-complete stored procedures that expanded the applicability of blockchains beyond cryptocurrencies into finance [11] [6], supply chain management [14] and healthcare [1]. Recently, *permissioned* systems such as Hyperledger Fabric [2] have been proposed for enterprise settings in which only authenticated entities participate in the network.

When processing transactions, blockchain systems must accomplish two goals: consensus on the order of transactions and consensus on the validity of transactions. In early blockchains such as Bitcoin and Ethereum, the miner selected to create a block provides consensus on order, and validity is independently verified by each peer in the network. However, in

4th International Symposium on Foundations and Applications of Blockchain 2021 (FAB 2021).
Editors: Vincent Gramoli and Mohammad Sadoghi; Article No. 4; pp. 4:1–4:15
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

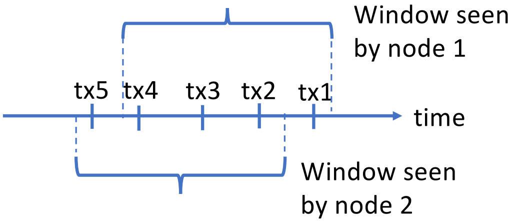

Figure 1 Different sliding windows seen by nodes with different clocks.

permissioned blockchains such as Fabric, consensus on order is obtained by using an ordering service and consensus on validity by using a subset of peers in the network (details in Section 2).

In simple cases, transaction validity can be determined based on account balance alone. Many blockchain systems thus use an account-based data model, in which each peer maintains the current state of each account in a so-called *state database*. This allows peers to validate transactions without having to reconstruct account balances from the entire history stored in the ledger. However, as permissioned blockchains gain traction in enterprise settings, blockchain systems must be enhanced to support new application needs. In this paper, we target applications such as financial services and online auctions and marketplaces, in which determining transaction validity is more complex and depends on *time*.

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 time *window* of recent account activity is required when executing these transactions.

To validate transactions whose correctness depends on time, a common solution is to obtain the current time from an external trusted oracle, along with the oracle's certificate of the current timestamp. This allows each peer to establish validity, and for all peers to come to the same deterministic conclusion. However, this approach breaks down when validity is *independently* determined by multiple peers, as is the case in permissioned blockchains such as Fabric or Corda [5]. In these situations, we need to obtain consensus on the current time among the endorser peers as a precondition to obtaining consensus on validity. Otherwise, it may be impossible to agree on the transaction outcome. For example, when processing an overdraft transaction, peer nodes with different current times may consider a different window of recent activity. We show an example in Figure 1, with two nodes and five recent transactions. Node 1 considers a window with transactions tx1 through tx4. Node 2 uses a different current time and considers a different window, with transactions tx2 through tx5.

To address the above issues in support of smart contracts with time-based logic, we make the following contributions.

1. *Trusted time for time-based transactions:* Instead of relying on external time oracles, we propose a light-weight consensus mechanism for time that assigns a trusted timestamp to each block. Block timestamps can then be used by the network to deterministically execute time-based smart contracts.
2. *Data layer support for time-based transactions:* We extend the account-based data model to store a sliding window of recent states, effectively maintaining a cache reflecting the effects of recent transactions. If a peer node needs to examine the recent history of an account, it can access the cache instead of reconstructing the account history 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%) and that the cache reduces the time to retrieve a 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 TimeFabric source code is publicly available at `https://github.com/aritramitra14/fabric/tree/timefabric`.

While we use Fabric in our proof-of-concept implementation, our solution generally applies to any blockchain in which multiple entities independently judge the validity of transactions. Thus, we allow a migration path for these types of blockchains if they do not wish to trust an external oracle to validate time-based transactions.

The remainder of this paper is organized as follows. Section 2 provides background information, including an overview of Hyperledger Fabric, Section 3 presents our solution, Section 4 discusses the experimental results, Section 5 reviews previous work, and Section 6 concludes the paper with directions for future work.

2 Background

Blockchain platforms can be categorized as public, or permissionless, and private, or permissioned; the former allows anyone to join the network whereas a private blockchain, commonly used in enterprise collaborations, includes a *membership service* that only allows authenticated entities to participate in the network. 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 [17], 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 Figure 2 (ignore the steps marked in red, which correspond to our modifications in TimeFabric and will be discussed later).

2.1 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, 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 Execute-Order-Validate transaction processing protocol proceeds as follows.

2.1.1 The Execute Step

Client applications submit *transaction proposals* to the Fabric network (step 1 in Figure 2). A subsets of peers, called *endorsers*, concurrently simulate the execution of the corresponding smart contracts in a sandboxed environment, i.e., without persisting the effects in the state database. Two such endorsers are shown in Figure 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.

2.1.2 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 Figure 2), it sends the transaction proposal, with endorsements attached, to the orderers (step 4 in Figure 2). 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 Figure 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.

2.1.3 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 Figure 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. As a result, each peer makes the same transaction commit (or not) decisions, and thus each peer stores the same version of the ledger and the state database. Also note that smart contracts are not re-executed during validation; only their effects are persisted in the state database.

2.2 Timestamps 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.

3 Our Solution

In this section, we present our solution to support smart contracts with time-based logic. Our design goals are:

1. to provide a trusted and consistent time reference for peers that validate transactions, without the need to consult external oracles,
2. to process transactions that reference this trusted time efficiently, with minimal overhead,
3. and to preserve the underlying blockchain system architecture as much as possible while making minimal modifications.

We address goal #1 in Section 3.1 and goal #2 in Section 3.2. We then describe implementation details of our proof-of-concept, TimeFabric, which is based on Hyperledger Fabric 1.4 (Section 3.3), followed by a discussion of TimeFabric's failure model compared to the underlying Fabric (Section 3.4).

3.1 Trusted Time

Our approach to maintain trusted time consists of the following steps.

1. **Validation of transaction timestamps**. Peers that validate transactions are given an additional responsibility: to ensure that 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 be rejected. The value of δ can be set in the corresponding smart contract code, and we will discuss how to choose an appropriate value for δ in Section 3.3.

2. **Assigning trusted block timestamps**. Additionally, peers that validate transactions need to assign *block timestamps*. In particular, they set the block timestamp to be the most recent or the median transaction timestamp within the block, among transactions that have been validated and have not been rejected[1]. However, if this timestamp is older than the timestamp of the previous block, then 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. Block timestamps become part of the blockchain and are included in the block hash for immutability.
3. **A heartbeat mechanism**. Suppose no transactions arrive for some time, say, one minute. Then, when a transaction finally arrives, its timestamp would be a minute into the future relative to the timestamp of the latest committed block. To ensure that trusted time moves forward, we require a "dummy" client that sends mock transactions even during periods of inactivity. A mock transaction updates a reserved "dummy" account with a random value, and its transaction timestamp equals the local time of the client. We will discuss how often these mock transactions need to be sent in Section 3.3.

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 for transaction validation, 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.

Our solution 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: transactions within a block may not be ordered by their timestamps.

3.2 Data Layer Support

Given our notion of trusted time, we create three methods that are accessible to smart contracts.

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() can be used when validating transaction timestamps, which can then be used during smart contract execution, e.g., to verify if deadlines are met. A simple implementation of this method is to extract the timestamp from the latest block in the ledger. Another option is to cache the block time at the validating peers.

GetHistoryRangeForKey() is meant to be used during smart contract execution to retrieve recent histories. A naive implementation is to reconstruct the account history from the ledger, which is expensive. Our solution is to maintain a cache capturing the effects of recent

[1] We will discuss the choice between maximum and median transaction timestamps in Section 3.3.

transactions. First, we assume that validating peers use the account-based data model and already maintain a key-value state database with the current account states. Additionally, we require each validating peer to maintain a *cache database*. Each record in the cache 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 to the validating peers. 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 $123 : 1607994614$.

There is one important distinction between the state database and the cache database. In the former, values of existing keys are updated 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 *pruned*, 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 [7]. 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 network along with other blockchain configuration parameters.

Technically, there is no limit on how much history can be stored in the cache database. However, to ensure high transaction throughput, it should be ensured that the cache database (and, of course, the state database) fits in memory.

3.3 TimeFabric Implementation

We now discuss the implementation details of TimeFabric, which is based on Hyperledger Fabric 1.4. Our modifications to Fabric's transaction validation pipeline are shown in red in Figure 2 and are explained below.

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 or median timestamp across the valid transactions, and we insert this timestamp into the block metadata.
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.

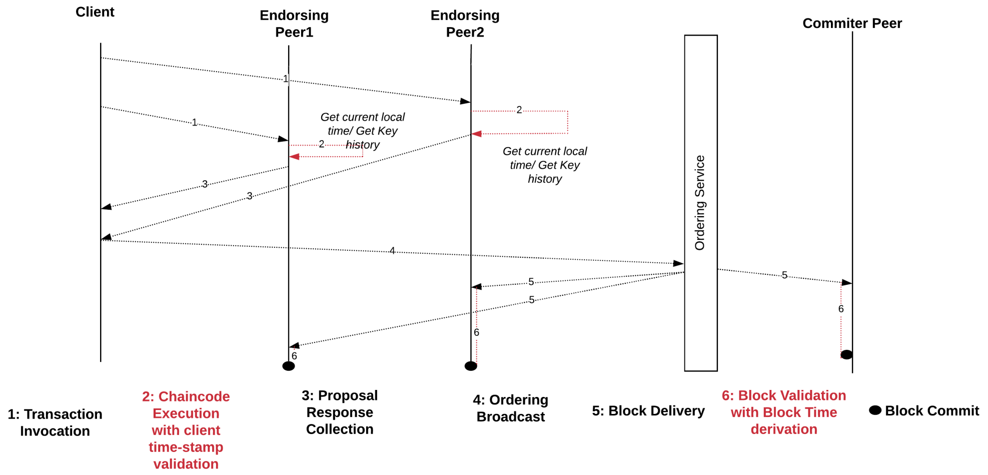
Client
Endorsing Peer1
Endorsing Peer2
Commiter Peer
Ordering Service
1
1
2
Get current local time/ Get Key history
2
Get current local time/ Get Key history
3
3
4
5
5
5
6
6
6
1: Transaction Invocation
2: Chaincode Execution with client time-stamp validation
3: Proposal Response Collection
4: Ordering Broadcast
5: Block Delivery
6: Block Validation with Block Time derivation
● Block Commit

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

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. We considered caching the block time at the endorsers, but the performance gains were minimal since the latest block is already cached in memory by Fabric.

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

We note a subtle but important issue related to read set validation in TimeFabric. 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 records read from the state database are validated to make sure the version numbers match; 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. In our example, only the latest version of key 123 obtained from the state database is validated, and the window of recent history of key 123 obtained from the cache database is not. 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).

Finally, there are no modifications to Fabric's ordering step. This satisfies design goal #3: 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, our modification are restricted to the endorsers and the validators.

In addition to the above changes to Fabric, our implementation of TimeFabric requires the addition of a dummy client for heartbeats (recall Section 3.1). To decide when to send a heartbeat, we observe that 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). Thus, we configure the dummy client to send a mock transaction every two seconds.

Now, recall the δ parameter for transaction timestamp validation. In our implementation, block time is permitted to be 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 two seconds plus the expected network delay. Large values of δ should be avoided to prevent malicious clients from submitting transactions with future timestamps and therefore advancing the block time too quickly. On the other hand, delays must be taken into account to ensure that legitimate transactions are not rejected, although a client who experiences an unusually long delay can always resubmit its transaction. Note that in a permissioned blockchain, even malicious clients need permission to access the blockchain by requesting access credentials from a membership service. Hence, clients exhibiting malicious behaviour can be ejected from the system. Nevertheless, if one wishes to err on the side of caution, malicious behavior can be reduced by setting the block timestamp to the median value of the timestamps in a block, rather than the maximum, since the median is harder to manipulate.

Finally, recall that our solution assigns one timestamp per block rather than one timestamp per transaction. However, observe that block timestamps alone produce totally ordered key histories in TimeFabric because Fabric's validation step ensures that a key can be updated at most once per block (recall Section 2).

3.4 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.

Table 1 End to End Throughput.

Fabric 1.4	TimeFabric
2927 ± 136	2831 ± 196

4 Experiments

In this section, 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. In practice, Fabric deployments may be geo-distributed, but our experiments focus on commit overhead and database access times at individual peers, which are independent of how the peers nodes are distributed. 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 lifecycle (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 API 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()*.

4.1 Block Time Implementation

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. We first send 25,000 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 100,000 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 [8], we set the block size to 100. We conduct 30 runs and report the mean throughput and the standard deviation in Table 1. This experiment shows that our changes only add about 3% overhead to the block validation and commit process.

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

Table 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

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 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 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 result 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

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 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 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 3, which reveals that the performance overhead of *GetTimenow()* is statistically insignificant.

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 an 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.

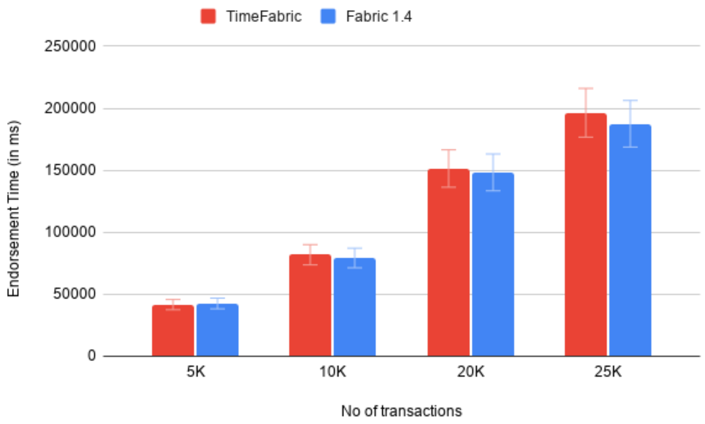

Figure 3 Endorsement time comparison.

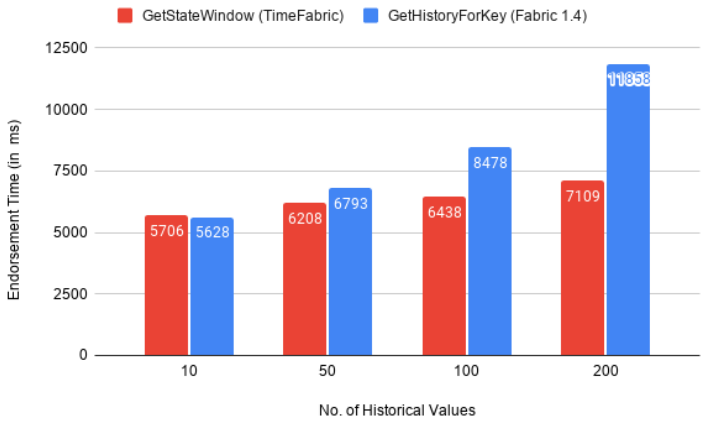

Figure 4 Endorsement time for window queries.

We show the results in Figure 4. The performance of Fabric's *GetHistoryForKey()* 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.

5 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[8, 17]. 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 [22], which proposes to add a 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 [16] 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 [15] extends Fabric by exposing *provenance* information, i.e., key histories, to smart contracts. For efficiency, LineageChain maintains an index over the provenance tree. This is conceptually similar to our use of the cache database to speed up sliding window queries. However, LineageChain does not offer a notion of time and its provenance queries do not support sliding windows.

An index to speed up temporal queries in Fabric was proposed in [9]. Account histories are stored in blocks on the file system and the index consists of pointers stored in the state database. The pointers identify blocks that contain transactions for a given account whose timestamps are within a given interval. The index is meant for off-line analytics over account histories. In contrast, our solution maintains an in-memory time window of the effects of recent transactions for use by smart contracts. Furthermore, our solution includes a notion of trusted time, whereas the index proposed in [9] was based on unverified transaction timestamps.

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 [19],[4],[21],[3]. These works highlight vulnerabilities but do not propose solutions, except [18] – 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 [13] and [10] argue that block sequence numbers are intrinsic to blockchains and best represent the temporal progression of a blockchain. Reference [13] 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.

6 Conclusions

In this paper, we presented a method to support smart contracts with time-based logic referencing current time or a time window of recent history. We showed that existing solutions such as querying an external time oracle, break down for systems in which multiple peers independently validate transactions. Instead, our solution assigns trusted block timestamps at transaction validation time, which can then be used by all peers to reach consensus on time-based transaction validity. To ensure that time-based smart contracts can be executed efficiently, our solution also adds a cache database storing a window of recent transactions. We implemented a proof-of-concept prototype, TimeFabric, on top of Hyperledger Fabric. 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.

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

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

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

8 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.

9 Himanshu Gupta, Sandeep Hans, Kushagra Aggarwal, Sameep Mehta, Bapi Chatterjee, and Praveen Jayachandran. Efficiently processing temporal queries on hyperledger fabric. In *34th IEEE International Conference on Data Engineering, ICDE*, pages 1489–1494, 2018.

10 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.

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

12 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

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

14 Hubert Pun, Jayashankar M Swaminathan, and Pengwen Hou. Blockchain adoption for combating deceptive counterfeits. *Production and Operations Management*, 2021.

15 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.

16 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.

17 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.

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

19 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.

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

21 Karl Wüst and Arthur Gervais. Ethereum eclipse attacks. Technical report, ETH Zurich, 2016.

22 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.