FabricCRDT: A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains

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

Ruben Mayer
Technical University of Munich
ruben.mayer@tum.de

Hans-Arno Jacobsen
Technical University of Munich

Abstract

With the increased adaption of blockchain technologies, permissioned blockchains such as Hyperledger Fabric provide a robust ecosystem for developing production-grade decentralized applications. However, the additional latency between executing and committing transactions, due to Fabric's three-phase transaction lifecycle of Execute-Order-Validate (EOV), is a potential scalability bottleneck. The added latency increases the probability of concurrent updates on the same keys by different transactions, leading to transaction failures caused by Fabric's concurrency control mechanism. The transaction failures increase the application development complexity and decrease Fabric's throughput. Conflict-free Replicated Datatypes (CRDTs) provide a solution for merging and resolving conflicts in the presence of concurrent updates. In this work, we introduce FabricCRDT, an approach for integrating CRDTs to Fabric. Our evaluations show that in general, FabricCRDT offers higher throughput of successful transactions than Fabric, while successfully committing and merging all conflicting transactions without any failures.

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

Keywords Blockchain, Hyperledger Fabric, CRDT, Eventual Consistency, Multi-Version Concurrency Control

ACM Reference Format:
Pezhman Nasirifard, Ruben Mayer, and Hans-Arno Jacobsen. 2019. FabricCRDT: A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains. In *Middleware '19: Middleware '19: 20th International Middleware Conference, December 8–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3361525.3361540

1 Introduction

Since the introduction of Bitcoin [26], new blockchains have been developed that provide their users with novel ways of storing and validating transactions and data in a trustless environment [9, 17]. However, a large number of existing blockchains fall significantly behind existing distributed databases concerning scalability and throughput [39]. This limitation is touted as a fundamental cost for providing security and trust in a decentralized and trustless environment. Unfortunately, the significantly lower transaction throughput of blockchains such as Bitcoin and Ethereum [9] has been a severe obstacle to the widespread adoption of these technologies. Although there have been several scalability approaches introduced [22], the public and permissionless nature of these blockchains make it difficult to find a good solution [13].

For use cases where the identity of users and nodes are known, when developing decentralized enterprise applications, permissioned blockchains constitute a viable alternative. One of the most prominent permissioned blockchains is Hyperledger Fabric [3], which offers significantly higher throughput and transactional guarantees in comparison to Bitcoin and Ethereum while allowing the deployment of Turing complete applications [37]. Fabric follows a three-phase Execute-Order-Validate (EOV) lifecycle for transactions. To ensure the consistency of the ledger, Fabric uses an optimistic concurrency control mechanism that enables concurrent updates. This mechanism is similar to the technique used by several database systems for increasing scalability and throughput [14, 32, 35]. Although the concurrency control mechanism is necessary for ensuring the consistency of the ledger, it constitutes a scalability bottleneck for Fabric. The added latency between executing and committing a transaction in Fabric is on the order of hundreds of milliseconds to seconds, which is significantly higher than the latency for committing transactions in conventional databases.

The considerable latency between the start (execution) and end (commit) of a transaction increases the probability of the concurrent arrival and execution of conflicting transactions, which can result in the failure of a large portion of transactions in the network [34]. Once a transaction fails, the only option for clients is to create a new transaction and resubmit, which adds to the complexity of Fabric application development. Therefore, providing a solution that enables Fabric to manage the conflicting transactions internally without rejecting the transactions can significantly improve the throughput of Fabric and simplify the application development process.

Conflict-free Replicated Datatypes (CRDTs) [33] address a similar concurrent update problem among node replicas. For certain update scenarios, CRDTs provide solutions for merging conflicting values internally and resolving update conflicts automatically. CRDTs are abstract datatypes that allow node replicas to eventually converge to a consistent state without losing updates. CRDTs have been implemented in several production-grade databases such as Redis [40] and Riak [8] and have established themselves as a viable solution in practice [29]. In this work, we introduce FabricCRDT, an extension of Fabric with CRDTs, and we assess the improvements CRDTs introduce to permissioned blockchains.

In doing so, we offer the following contributions in this work:

1. We investigate the applicability of CRDTs to permissioned blockchains and propose a novel approach for integrating CRDTs with Fabric that automatically resolves transaction conflicts without the loss of updates.

Middleware '19, December 8–13, 2019, Davis, CA, USA

ACM ISBN 978-1-4503-7009-7/19/12…$15.00
https://doi.org/10.1145/3361525.3361540

2. We extend Fabric with CRDTs and describe the design of a new system called FabricCRDT. Our extension is implemented without disrupting the standard behavior of Fabric, making it backward compatible with existing chaincodes and transactions. Also, our approach requires only minimal changes to Fabric and reuses its main components.
3. Our solution simplifies the complexity of Fabric application development by eliminating the developer's concerns about transaction failures due to concurrent update conflicts. Also, we offer a simplified CRDT-based programming model for developing CRDT-enabled applications on FabricCRDT, with a minimal learning curve for developers who are already familiar with Fabric.
4. We provide insights into the appropriate use cases for a CRDT-enabled permissioned blockchain. We also perform extensive evaluations to understand the best configuration of FabricCRDT for executing CRDT-compatible applications.

The remainder of the paper is organized as follows. First, we provide an overview of Fabric and CRDTs in Section 2, followed by a detailed discussion of multi-version concurrency control (MVCC) conflicts in Section 3, and our approach in Section 4 and Section 5. We discuss use cases and the potential of CRDT-enabled blockchains in Section 6. We report the results of our evaluation in Section 7 and review related work in Section 8.

2 Background

2.1 Hyperledger Fabric

Hyperledger Fabric (Fabric) is an open-source permissioned blockchain, initially developed by IBM [31]. Fabric provides an ecosystem for hosting and executing blockchain applications, offering a wide range of features and services ranging from storing the application data to a sophisticated identity and membership management. Developers host smart contracts, also known as chaincodes, on Fabric which clients interact with by creating and submitting transactions. Developers use *chaincode shim* that provides APIs for the chaincode to interact with data on the ledger. Shim is available in programming languages such as Go and Node.js.

The two main components of Fabric are *peers* and *orderers*. Peers are responsible for executing transactions and storing the data on their local copy of the ledger. The peer's ledger consists of an append-only blockchain and a world state database, realized by CouchDB [14]. Executing all valid transactions included in the blockchain starting from the genesis block results in the current state of the world state database. Orderers are responsible for defining a total global order for transactions and batching the ordered transactions into blocks. Fabric divides peers into *organizations* and provides them with private communication channels. A complete workflow for committing a transaction is depicted in Figure 1 (adapted from Ref. [34]). Every successfully committed transaction follows these three steps:

1. **Execution and Endorsement:** The client creates a *transaction proposal* containing the name of the chaincode, the input data, and the endorsement policy. An endorsement policy specifies which peers from which organizations are required to execute and sign the proposal (also known as endorsing). The client submits the proposal to the peers specified by the endorsement policy in parallel (Step 1 in Figure 1). Each endorsing peer executes the chaincode against the local copy of the world state database, signs the results, and sends the results back to the client (Step 2). The results are in the form of read and write sets, where read sets contain the keys read during execution and write sets contain the key-value pairs to be written to the ledger. Peers do not modify their local copy of the ledger during this phase and only execute the proposal in an isolated fashion, i.e., peers *simulate* the transaction proposal.
2. **Ordering:** Once the client has received enough endorsements that satisfy the endorsement policy, it creates a transaction containing the proposal's payload, the endorsements, and other metadata, and sends the transaction to the ordering service (Step 3). The ordering service receives transactions from every client in the network, defines a total global order for the transactions and batches them into new blocks (Step 4), which are broadcast to the peers (Step 5).
3. **Validation and Commit:** Peers perform two validations on the transactions in the incoming blocks and then commit the transactions. For the first validation, a peer validates the endorsement policies of the transactions in parallel to ensure that each transaction satisfies the predefined endorsement policy. Second, a peer sequentially compares the read-set with its local copy of the world state database to ensure that the records that were read during the endorsement phase have not changed concurrently in the world state database. The transactions that successfully pass both validations are considered valid. Finally, a peer appends every valid and invalid transaction in the block to the blockchain and updates the world state database with the write-set of the valid transactions.

2.2 Conflict-free Replicated Datatypes

CRDTs are abstract datatypes that can be replicated on several nodes with the guarantee to eventually converge to the same state without requiring a consensus protocol [29, 33]. CRDTs provide well-defined application interfaces, representing specific data structures such as counters, sets, lists, maps, JSON objects, and others. CRDTs provide these interfaces by extending the basic datatypes with some metadata, which makes the updates on these datatypes at least commutative. Thus, commutative updates can be applied in different orders on replicas, resulting in the same state no matter the order of updates, provided no updates are lost or duplicated.

In general, CRDTs are divided into two main categories: state-based CRDTs and operation-based CRDTs. State-based CRDTs exchange the full state or delta state and merge the local state with the received state. For operation-based CRDTs, nodes propagate the state by sending update operations to other nodes. As an example, a counter datatype which only increments a value by one can be converted to a grow-only CRDT counter, by defining an *increment operation* that increments the value of the counter by one. The grow-only CRDT counter is relatively easy to create since the increment operation is inherently commutative, although not idempotent. Therefore, the grow-only CRDT counter converges to the same state no matter in which order the operations are applied, yet the same operation cannot be applied more than once. For interacting with this counter over the network, nodes send increment operations, and given an asynchronous distributed system where the delivery of messages eventually succeeds without message loss

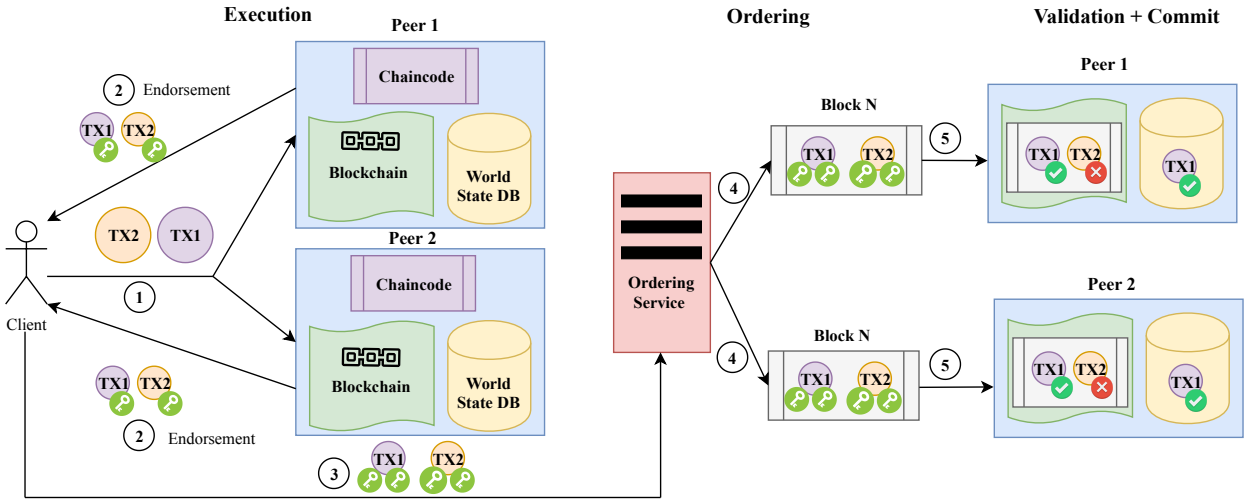


Figure 1. Transaction flow in Hyperledger Fabric.

nor duplication, the counter eventually converges to the same value on all nodes.

Among existing CRDTs, a JSON CRDT represents a complex general-purpose data structure [23]. A JSON object is inherently a tree structure, consisting of other structures like maps and lists. In JSON, a map is a dictionary of key-value pairs where keys are string constants and values are either primitive values like string or numbers or complex structures like other maps and lists. In this work, we assume that maps are unordered structures and the values in the maps are either a string, a map, or a list. In JSON, lists are arrays of objects, which can be a combination of primitive values or complex structures, like string, numbers, maps or lists.

3 MVCC Conflicts

In this section, we analyze Fabric's concurrency control mechanism and the causes of transaction failures in more detail.

A transaction proposal invokes the chaincode on Fabric, which, based on the implemented logic, interacts with the stored ledger data in three ways during the chaincode execution:

- **Read Transaction:** Chaincode only reads key-value pairs from the ledger.
- **Write Transaction:** Chaincode only writes key-value pairs to the ledger without reading any pair.
- **Read-Write Transaction:** Chaincode reads and writes key-value pairs from/to the ledger.

The execution of transaction proposals results in a read-write set. The read set includes a list of keys and the version number of the key's value that a peer retrieved from the ledger during the execution of the chaincode. The write set contains the key-value pairs that will be committed to the ledger at the end. Read transactions do not change the state of the ledger, and clients do not send the transactions for ordering and committing. The write transaction results in a read-write set with an empty read set. Hence, these transactions will not cause any read-write set conflict. However, a read-write transaction with an outdated version number in the read-set fails the validation. To illustrate the problem better, imagine that at time TS_1, peer P_1 has the world state $WS: \{(K_1, VN_1, VL_1), (K_2, VN_2, VL_2), (K_3, VN_3, VL_3)\}$ and receives a block containing five transactions with corresponding read-write sets as follows, where K represent the key of the key-value pairs, VN represents the version number of the retrieved key-value pairs from the world state and VL is the value of the key to be written to the blockchain and world state:

- $\mathbf{T_1}$: < **Read** : $\{(K_2, VN_2)\}$, **Write** : $\{(K_2, VL_1)\}$ >
- $\mathbf{T_2}$: < **Read** : $\{(K_1, VN_1), (K_2, VN_2)\}$, **Write** : $\{(K_3, VL_3)\}$ >
- $\mathbf{T_3}$: < **Read** : $\{(K_2, VN_2)\}$, **Write** : $\{(K_3, VL_1)\}$ >
- $\mathbf{T_4}$: < **Read** : $\{(K_3, VN_2)\}$, **Write** : $\{(K_2, VL_1)\}$ >
- $\mathbf{T_5}$: < **Read** : $\{\}$, **Write** : $\{(K_3, VL_2)\}$ >

Given that all five transactions pass the endorsement policy validation (not explicitly shown here), P_1 sequentially validates the five transactions in the block by comparing the version number of each key in the read set to the version number in the world state. A transaction is considered valid if both version numbers are equal. If the version numbers are not equal, the peer invalidates the transaction as a *multi-version concurrency control conflict (MVCC conflict)*. The key's mismatch is the result of updates committed by preceding valid transactions. The preceding transactions may be included either in the previous blocks or in the same block but are preceding the current position of the conflicting transaction. Committing keys in the write-set of the valid transactions causes the version number of keys in the world state database to change. Therefore, P_1 marks T_1 as valid and T_2 and T_3 as invalid, because the write-set of T_1 updates K_2 so that its new version number VN'_2 and the version number of K_2 in T_2 and T_3's read-set does not match ($VN_2 \neq VN'_2$). Also, P_1 marks T_4 and T_5 as valid.

This multi-version concurrency control mechanism is a commonly used method in database systems to increase the throughput and decrease the latency, instead of using blocking mechanisms such as shared locks [35]. Although this mechanism is necessary for ensuring data consistency and isolation of transactions, the relatively high latency between the creation of the read-write set and the validation of the read-set in Fabric can result in a large number of transactions in a block to fail, especially when a small set of frequently accessed keys are included [15, 34]. This high

latency consists of the endorsement latency, the ordering latency, and the commit latency [37]. The endorsement latency is the time needed for the client to obtain all the required endorsements, which, depending on the endorsement policy and the complexity of chaincodes, varies significantly for different transactions. The ordering latency is the time required for the transaction to be included in one block and to be broadcasted to the peers. The ordering service creates a block based on several criteria, including the maximum number of transactions, the maximum total size of transactions in a block and a timeout period for creating blocks. For higher transaction arrival rates, the ordering service creates a block as soon as the maximum size is reached. However, for lower arrival rates, the transaction can be delayed until the timeout period is reached. The timeout period is a configurable parameter which can be on the order of seconds. Finally, the commit latency is the time taken for a peer to validate and commit transactions in the block. These delays are inherent to the design of Fabric and can not be significantly reduced without fundamental changes to Fabric.

4 FabricCRDT Design

In this section, we discuss the system model and requirements of FabricCRDT and introduce our approach for integrating CRDTs with Fabric.

4.1 System Model

As FabricCRDT is an extension of Fabric, we assume the same system model as Fabric. We consider an asynchronous distributed model where all users and nodes are connected in a way that it is guaranteed for all transactions to be delivered eventually, despite arbitrary delays. However, the order of the transactions in a block is not guaranteed to be the same order that transactions are issued or arrive at the ordering service. As the ordering service does not guarantee to prevent duplicate delivery of transactions, for example, when the client intentionally or unintentionally submits duplicate transactions, we rely on peers to identify the duplicate ones with same transaction identification numbers. In this work, we assume that clients do not submit duplicate transactions. In the case that duplicate transactions are submitted, FabricCRDT also commits duplicate transactions.

4.2 FabricCRDT Requirements

We define a set of requirements that FabricCRDT must satisfy. First, *compatibility*: we aim to extend Fabric with CRDT-enabled functionalities with minimal changes to the original design of Fabric. This way, we keep the learning curve minimal for developers who already designed applications for Fabric. Also, the applications developed for Fabric remain compatible with FabricCRDT. The second requirement is *no failure*: FabricCRDT should be able to commit all valid CRDT-based transactions successfully. We define valid transactions as the transactions submitted by the client which pass the endorsement policy validation successfully. The third requirement is *no update loss*: by committing all valid transactions in a block, FabricCRDT eventually converges to the same state on all peers, and all client's updates are preserved while using CRDT techniques to merge conflicting transactions. The last requirement is *generality*: to accommodate developers with the possibility of realizing a wide range of use cases, the CRDT approach used for merging conflicting transactions in FabricCRDT should provide users with a general-purpose data structure to submit data to the ledger.

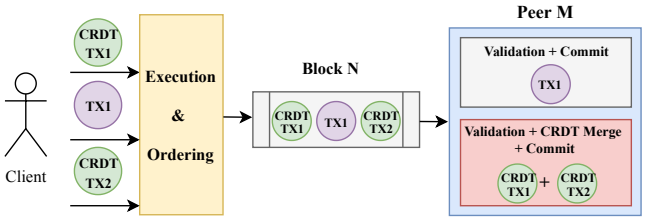


Figure 2. Transaction flows on FabricCRDT.

4.3 Fabric and CRDTs

To achieve the requirement we discussed, we need to identify the right approach for dealing with conflicting transactions internally. As discussed in Section 3, Fabric rejects transactions with an outdated version number of key-value pairs in the read-set and discards these transactions' write-set, as committing a write-set with outdated version can result in data inconsistencies. To avoid the failure of conflicting transactions and data inconsistencies, FabricCRDT does not reject transactions, but merges the values of the conflicting transactions by using CRDT techniques.

Since we aim to keep Fabric applications compatible with FabricCRDT, we define a new type of transaction that encapsulates all CRDT-related functionalities. Figure 2 displays the transaction flow in FabricCRDT, where CRDT and non-CRDT transactions coexist. CRDT transactions have a similar structure as standard Fabric transactions; however, they invoke chaincodes which modify CRDT-encapsulated values on the ledger. The chaincodes that contain CRDTs are executed in the same way as non-CRDT chaincodes, but peers flag the key-value pairs in the resulting transaction's write-set as "CRDT key-values". On FabricCRDT, both types of transactions go through the same ordering steps, but they are treated differently in the final validation and commit phase. Non-CRDT transactions go through the same validation steps as on Fabric, but CRDT transactions only go through the endorsement validation check. Then, instead of going through the MVCC validation, the transaction values of conflicting transactions are merged automatically by using CRDT techniques before being committed to the ledger. The CRDT procedures used for conflict resolution depends on the type of CRDT object; for example, managing grow-only CRDT counters requires different techniques than merging JSON CRDTs. In our prototype of FabricCRDT, we support merging JSON CRDTs.

5 FabricCRDT Implementation

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

5.1 CRDT Transactions in a Block

The CRDT and non-CRDT transactions in a block follow the same workflow until they reach the multi-version concurrency control validation (MVCC validation). The non-CRDT transactions pass

Algorithm 1: Merge CRDT transactions in a block.

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

through the MVCC validation, and the peer commits the valid transactions to the ledger. The CRDT transactions in a block are merged before getting committed. Algorithm 1 explains our approach for managing transactions in a block on FabricCRDT.

For resolving the CRDT transactions in one block, first, we iterate through all transactions in the block, and for each transaction, we iterate through the key-value pairs in the transaction's write-set (lines 3 to 14 in the algorithm). If the key-value pair is not marked as a CRDT, we skip the key-value pair to be handled as a non-CRDT transaction. However, if the key-value pair is flagged as a CRDT, the algorithm first checks if a CRDT object with the same key already exists in a local set containing all CRDT objects. If a CRDT object does not exist, the algorithm instantiates a new CRDT object with the key and adds it to the set. The type of CRDT object depends on the type of CRDT value in the key-value pair. For example, for a JSON CRDT type, an empty JSON CRDT object is instantiated. Afterward, the peer converts the binary value of the key-value pair to the corresponding type and merges it with the CRDT object. Then, the set containing all CRDT objects is updated (lines 7 to 12). We discuss the steps required for merging the individual CRDTs in Section 5.2. After the first iteration, the peer performs MVCC validation on non-CRDT transactions (line 15). Afterward, the algorithm iterates through every transaction's write-set once more to check if there exists a CRDT object for that key in the local CRDT set (lines 16 to 22). If a CRDT object exists, then the CRDT object is converted to the corresponding datatype, for example, for a JSON CRDT, it is converted to a JSON object (line 20). The converted object is a representation of the datatype with all the CRDT-related metadata cleaned up and removed. Finally, the object is converted into a byte array that replaces the value of the key-value pair in the write-set of the transaction (lines 21 and 22). This second iteration through every transaction's write set is necessary because the peer is not aware of all key-value pairs in the CRDT transactions in the block that needs to be merged until the end of the first iteration. Once all CRDT transactions are merged, the peer finalizes and cleans up the CRDT objects and updates the write-values of the corresponding transactions with the new converged value, which is then committed to the ledger.

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

Listing 1. Sample JSON objects in transactions' write-set.

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

The result of merging the two CRDT transactions is shown in Listing 2. The write-set of Transaction 2 is identical to the write-set of Transaction 1.

Listing 2. Result of example JSON merge.

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

5.2 JSON CRDTs on FabricCRDT

Although the approach discussed in Algorithm 1 is independent of the CRDT types, the necessary mechanism for resolving conflicts

and merging different CRDTs varies between different CRDT types and requires specific implementation support. In our prototype of FabricCRDT, we focused on implementing and integrating JSON CRDTs [23], which provide a general-purpose data structure for complex use cases.

We implemented JSON CRDTs based on the theoretical work of Klepmann et al. [23] and a GoLang JSON CRDT implementation [28]. In Ref. [23], the authors introduce the formal semantics and the algorithm for implementing an API for interacting with a JSON CRDT. The algorithm provides an API for modifying JSON objects, such as inserting, assigning, and deleting values, as well as reading from the JSON. The Reading API does not cause any modification to the JSON, but modifying the JSON is represented by *operations* which have globally unique identifiers. The API described by the authors, although necessary for ensuring the automatic resolution among several processes, is cumbersome to use for chaincode developers. In FabricCRDT, every peer observes the transactions in a block in the same order. We exploit this property to simplify the API. To use the JSON CRDTs in the chaincode, similar to chaincodes on Fabric, developers should create JSON objects. However, for submitting the key-value pairs to the ledger, the developer should use the CRDT-specific *putCRDT* command that we implemented in the chaincode shim. This command only informs the peer that this value is a CRDT and does not interact with the CRDT in any way. The operations required for merging the JSON CRDTs are performed on the peers without the interference of the chaincode developer.

Algorithm 2 describes our approach for merging JSON CRDTs. This algorithm is the implementation of the *MergeCRDT* function in line 11 of Algorithm 1. Algorithm 2 iterates through each key-value pair in the JSON object, where the value is either a string, a list, or a map. The items included in the list or map may include nested maps or lists. For each value in the JSON object, first, we create an empty cursor and an empty dependency list (lines 3 and 4). The cursor defines the path from the head of the JSON CRDT to the node where the mutation for modifying the JSON CRDT happens. A mutation defines the modification, such as add or delete, that is applied to the JSON object. The dependencies set contains the unique identifier of all operations which should be performed before the current operation is executed. We ensure that the operations identifiers are globally unique by using an instance of a Lamport Clock [25] for each JSON CRDT instantiation. The Lamport clock is incremented by one with every new operation to ensure the causal order of the operations.

If the value of a key in the JSON object is a string, the algorithm executes lines 6 to 11. First, it extends the cursor with the current key and increments the Lamport clock by one. Then, it creates a mutation for inserting the current string value and the current key. Afterward, an operation is created with the current value of the Lamport clock as the identifier of the operation. The operation also holds the mutation, a dependency list, and the cursor pointing to the location in the JSON CRDT where the mutation occurs. Finally, the operation is applied to the JSON CRDT (line 10). For applying the operation, first, we check if all dependencies in the operation's dependency list are already applied. If some of the operations are missing, we queue the operation until all dependencies are applied. If there is no pending operation, we apply the operation by using the operation's cursor to traverse from the head of the JSON CRDT. For every node in the cursor, if the node already exists, we add

Algorithm 2: Merge a JSON object with JSON CRDT.

```
1  MergeCRDT (JsonCRDT, Json)
       input  : JsonCRDT, An initialized JSON CRDT object.
       input  : Json, A JSON object to be added to the JSON
                CRDT.
2      foreach key_i, value_i ∈ Json do
3          cursor := NewCursorElements()
4          dependencies = set()
5          if value_i.IsString() then
6              AddCursorElement(cursor, key_i)
7              JsonCRDT.TickClock()
8              mutation = NewInsertMutation(key_i, value_i)
9              operation =
                 NewOperation(JsonCRDT.ClockToString(),
                 dependencies, cursor, mutation)
10             ApplyOperationToJSON(JsonCRDT, operation)
11             dependencies.Add(JsonCRDT.ClockToString())
12         else if value_i.IsList() then
13             foreach listValue_j ∈ value_i.GetListItems() do
14                 AddCursorElement(cursor, key_i)
15                 RecursivelyAddListItemToJsonCRDT(
                     JsonCRDT, key_i, listValue_j,
                     dependencies, cursor)
16                 RemoveCursorElement(cursor, key_i)
17         else if value_i.IsMap() then
18             foreach
                 mapKey_j, mapValue_i ∈ value_i.GetMapItems() do
19                 AddCursorElement(cursor, key_i)
20                 RecursivelyAddMapItemToJsonCRDT(
                     JsonCRDT, mapKey_j, mapValue_i,
                     dependencies, cursor)
21                 RemoveCursorElement(cursor, key_i)
```

the identifier of the current operation to the node to record the current operation's node dependencies. If the node from the cursor is missing in the JSON CRDT, we add the node to the JSON CRDT and the operation's identifier to the node. Once we reach the end of the cursor and the location of the node is found, we apply the mutation to the JSON CRDT and insert the node. For adding the node, we insert a dictionary item with the key as the operation identifier, and the value as the string value from the JSON object.

When the value of the JSON object is a list, we iterate through the list's items (lines 14 to 16). For every list item, first, we append the cursor with the current key in the JSON object, then we call a recursive function that extends the JSON CRDT with the content of the list item. We use a recursive function since the value of the list item could either be a string, a list, or a map, which may contain further nested list or map items. The recursive function either extends the JSON CRDT with the string value as described in the algorithm (lines 6 to 11) or if the value is a list or a map, it extends the JSON CRDT (lines 13 to 16 or lines 18 to 21, respectively.) When the value of the JSON object is a map (lines 18 to 21), we follow the same approach as the list type, but we extend the cursor with

the key of the key-value pairs in the map instead of the key of the current JSON object.

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

6 Use Cases and Potentials of a CRDT-enabled Fabric

Numerous use cases have been proposed that benefit from CRDT-enabled database systems, such as data metering, global voting platforms, and shared document editing applications [24, 29]. Similarly, these use cases also benefit from the advantages that permissioned blockchains offer, for instance, decentralized trust. CRDT-enabled blockchains ease the realization of these use cases on blockchains.

FabricCRDT, as an extension to Fabric, supports all use cases that can be implemented on Fabric. However, based on the requirements we explained in Section 4.2, FabricCRDT, by taking advantage of CRDTs, offers two additional properties which are beneficial to the CRDT use cases. FabricCRDT ensures that (1) all submitted transactions that pass endorsement policy validation are committed successfully (*no failure* requirement) and (2) no user updates are lost when concurrent updates on the same keys are submitted (*no update loss* requirement), i.e., it offers eventual strong consistency.

One major use case that benefits from FabricCRDT are collaborative document editing platforms, which provide an environment for users to concurrently work on shared documents. Because of the inherent concurrent nature of these platforms, conflicts from updating the same content can frequently occur. CRDTs are a practical technique for resolving these kind of conflicts [1, 2]. By using CRDT features that FabricCRDT offers, like JSON CRDTs, developers can create blockchain-based document editing applications. On FabricCRDT, documents are stored as JSON objects, and edit updates are committed as CRDT transactions. Now, updates are merged without the loss of user's data (*no update loss* requirement); further, no updates will fail, so that users do not need to redo and resubmit their edits (*no failure* requirement). Furthermore, users benefit from the trust and security of permissioned blockchains when they use FabricCRDT. Ref. [23] discusses how JSON CRDTs are used for representing text documents.

Another prominent application of permissioned blockchains is supply-chain management applications for tracing and ensuring the quality of different products from food to pharma industries [7, 38]. Sensitive goods like drugs and fresh fruits and vegetables should be kept within specific conditions, e.g., regarding temperature, humidity, and light, during transportation and storage. To ensure that these goods are treated in compliance with regulations and policies, sensors continuously monitor the goods and record the readings on the blockchain to keep them secured against manipulations. Although the use case of storing a stream of sensor readings from IoT devices can be implemented on Fabric, we argue that this use case is even a better fit for FabricCRDT. Depending on the design of the system, different readings from different IoT devices may collide, for example, when a temperature sensor and a humidity sensor concurrently submit records to update a shared list of the sensor readings of the same good. Using FabricCRDT, it is ensured that conflicts are merged automatically and that all sensor data end up in the world state (*no update loss* requirement). Due to resource limitations of IoT devices (e.g., regarding energy), the extra effort required for resubmitting failed transactions may be prohibitive. Using FabricCRDT makes it possible for IoT devices to submit transactions *once* without needing to take care of transaction failures and data loss (*no failure* requirement).

There are also limitations to FabricCRDT. Use cases that require transactional isolation of repeatable reads [5] are not a good fit, as FabricCRDT commits transactions even if their read-set is outdated. This includes use cases for transferring assets. For example, financial applications like SmallBank [34] or FabCoin [3], which are developed for Fabric, are bad choices to be adapted as a CRDT-based blockchain application. These applications represent asset creation and transfers between the owner; modeling them as CRDTs results in vulnerabilities, e.g., to the double-spending attack [20], where an attacker creates several transactions to transfer a single asset to multiple owners. On Fabric, only one of the attacker's transactions is successfully committed, and the MVCC validation fails on other transactions, since the committed transaction causes the read-set of other transactions to be outdated. However, FabricCRDT skips the MVCC validation, merges the transactions' values, and successfully commits all of the attacker's transactions.

7 Evaluation

In this section, we provide a comprehensive evaluation of our design. We evaluate the number of successful transactions, latency, and the throughput of FabricCRDT and Fabric under various configurations and workloads.

7.1 Workload Generation

Currently, the blockchain research community lacks a standard workload and benchmarking approach for evaluating different blockchain systems. Benchmarks such as TPC-C [10] and TPC-H [11] from the database community are not directly applicable to blockchains. They have been created for database systems and are not directly compatible with Fabric or FabricCRDT. Adapting these workloads to the transactional structures of Fabric or other blockchain systems requires a steady community effort.

We created a custom workload for evaluating the performance of FabricCRDT, consisting of chaincodes for an IoT use case and use Hyperledger Caliper [30] for generating and submitting the transactions and collecting the performance metrics. When designing the workload, we focused on understanding the limitations and potentials of a CRDT-enabled Fabric. Since standard transactions in Fabric and FabricCRDT go through identical workflows, we argue that for conflict-free workloads both systems show similar performance. For this reason, for most experiments, we evaluate the performance of FabricCRDT on workloads that consist of conflicting read-write transactions. Additionally, we perform one set of experiments with workloads consisting of conflicting and non-conflicting transactions in different ratios.

For our experiments, we implemented a chaincode that receives and stores temperature readings and device identification numbers of IoT devices. When executing a transaction, the chaincode first reads a key-value pair from the ledger, where the key is the device's identification number and the value is a JSON object containing the previous temperature readings of the device. Then, the chaincode adds the new temperature reading to the JSON object and submits

it to be written to the ledger. As an example, Listing 3 shows the JSON object that a transaction submits with one property for the device identification number and a list containing three temperature readings. For each experiment, the structure of the JSON object and the number of submitted JSON objects differ, which we will specify accordingly. However, the logic and behavior of the chaincode are the same for all experiments.

Listing 3. Sample JSON object submitted by a transaction.

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

7.2 Experimental Setup

We deployed a Fabric network on a Kubernetes v1.11.3 cluster consisting of three controller nodes, three worker nodes, one DNS and load balancer node, one NFS node, and one command-line interface (CLI) node. All nodes except the CLI node run on Ubuntu 16.04 virtual machines (VMs) with 16 vCPUs and 41 GB RAM. The CLI node runs on an Ubuntu 16.04 VM with 8 vCPUs and 20 GB RAM. All VMs are installed on top of KVM provided by OpenStack Mitaka and are interconnected by 10 GB Ethernet. We use CouchDB as the world state database and Apache Kafka/Zookeeper for the ordering service. Also, we use Hyperledger Caliper v0.1.0 [30]. All experiments run a Fabric and a FabricCRDT network of three organizations, two peers per organization, one orderer node, and one channel.

For the evaluations, we kept the number of organizations, peers, channels, and clients constant. Since in FabricCRDT, we did not change any Fabric components that are responsible for the communication between different parts over the network, we focus on evaluating the internal behavior of peers in FabricCRDT and Fabric.

For each experiment, we start with an empty ledger and populate the ledger with keys that are read during the experiment as included in the configuration table of each experiment. During an experiment, Caliper uses four clients to submit in total 10,000 transactions. Besides this fixed setup, each experiment employs an additional configuration which we will specify.

7.3 Effect of Different Block Sizes

We examine the impact of the block size on the total number of successful transactions, the throughput of successful transactions, and the average latency of successful transactions. We configured the FabricCRDT and Fabric networks as described in Table 1 and only changed the block size in each experiment. We configured the maximum and preferred number of bytes for a block to 128 MB and the block timeout to 2 seconds. We kept these values fixed for each experiment but gradually increased the maximum allowed number of transactions in a block from 25 to 1000. Each chaincode invocation reads one key-value pair from the ledger and writes one pair back. Also, the JSON object that is written to the ledger has two keys, containing a string constant and a list, as we exemplify in Listing 3.

In order to find the best configuration of FabricCRDT and Fabric under worst-case workloads, all transactions modify the same keys; hence, all transactions are dependent on each other and are conflicting, FabricCRDT will merge all the key-value pairs of all transactions in each block. Therefore, a higher number of transactions in a block induces a higher overhead for the peer to merge a higher number of JSON CRDTs.

Table 1. Configuration for evaluating the impact of the block size.

Parameters	Value
Transaction submission rate per second	300
Number of read keys per transaction	1
Number of write keys per transaction	1
Number of keys per JSON object	2

Observations - The results of the experiments are summarized in Figure 3. In Figure 3(a), we can observe that FabricCRDT has a higher throughput for smaller block sizes. The main reason for the degradation of throughput in FabricCRDT with larger block sizes is the higher overhead required for merging a higher number of JSON CRDTs. For FabricCRDT, the highest throughput overall was 267 transactions per second for a block size of 25 transactions.

We can observe in Figure 3(b) that FabricCRDT experiences a higher transaction commit latency for larger block sizes because of the lower throughput, resulting in more time needed for committing all transactions. In Figure 3(c), we can observe that FabricCRDT successfully commits *all* submitted transactions. In Fabric, there are always some conflicting transactions that cannot be committed, while in FabricCRDT, all conflicts are automatically merged, and all transactions are successfully committed.

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

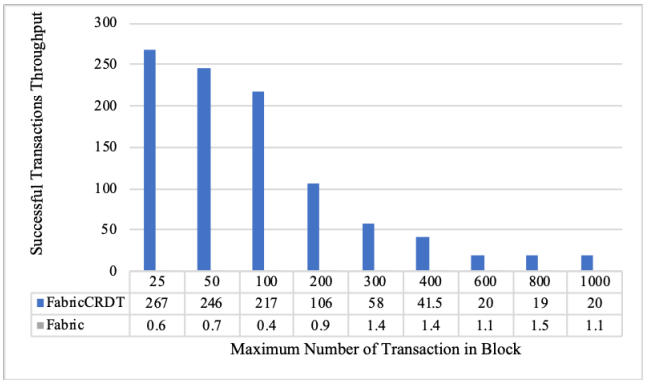
7.4 Effect of Different Number of Reads and Writes

To understand the effect of a higher number of key-value pairs in the transaction's read-write set, we change the number of key-value pairs that were read from and written to the ledger. For each experiment, we chose either 1, 3, or 5 key-value pairs to be read and to be written. Table 2 specifies the experimental configuration. During each experiment, we kept the set of read and write keys identical for all transactions. For example, in the experiment with five read-keys and five write-keys, in every transaction, we read or write the same set of 5 distinct keys.

Table 2. Configuration for evaluating the impact of read and write keys.

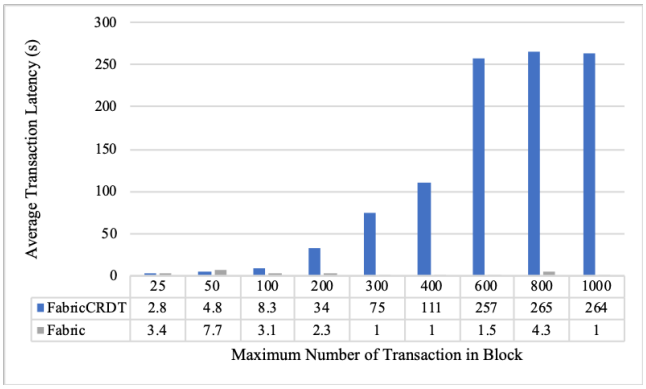
Parameters	Value
Transaction submission rate per second	300
Number of keys per JSON object	2

Observations - Figure 4 summarizes the results of the experiments. As expected, we can observe in Figure 4(a) that the throughput of FabricCRDT decreases as the read-write set grows, because of the increased overhead for merging a larger number of values.



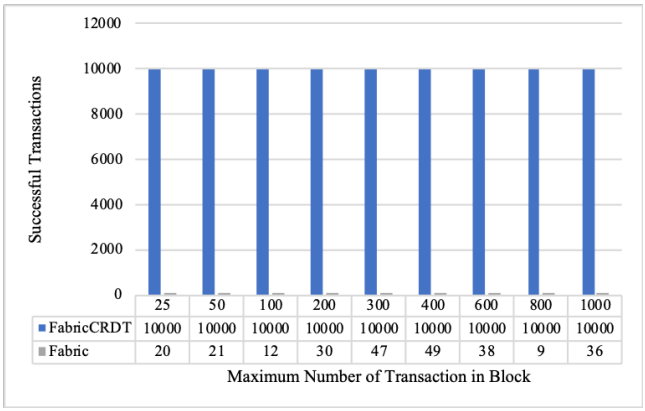
	25	50	100	200	300	400	600	800	1000
FabricCRDT	267	246	217	106	58	41.5	20	19	20
Fabric	0.6	0.7	0.4	0.9	1.4	1.4	1.1	1.5	1.1

(a) Successful transactions throughput per second for different block sizes.



	25	50	100	200	300	400	600	800	1000
FabricCRDT	2.8	4.8	8.3	34	75	111	257	265	264
Fabric	3.4	7.7	3.1	2.3	1	1	1.5	4.3	1

(b) Average latency of successful transactions for different block sizes.

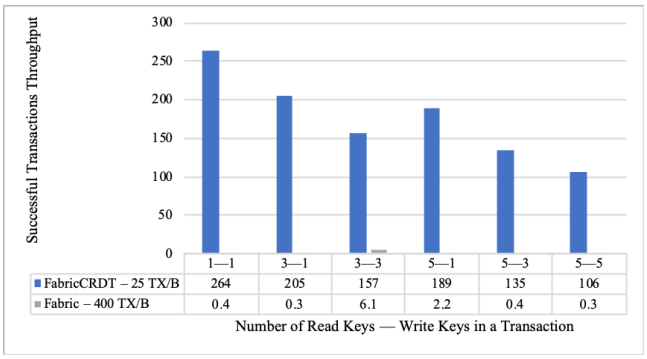


	25	50	100	200	300	400	600	800	1000
FabricCRDT	10000	10000	10000	10000	10000	10000	10000	10000	10000
Fabric	20	21	12	30	47	49	38	9	36

(c) Number of successful transactions for different block sizes.

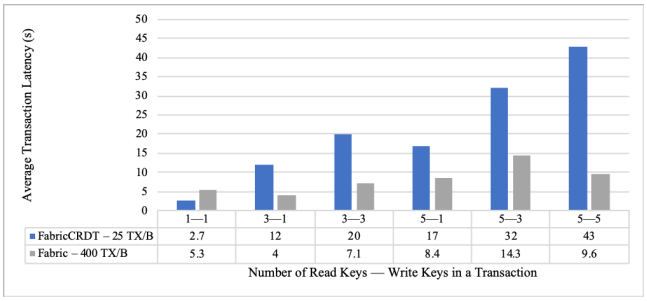
Figure 3. Effect of block size on throughput, latency, and success rate of FabricCRDT and Fabric.

We see that FabricCRDT is affected by both the number of reads and writes in the transactions. In comparison to FabricCRDT, Fabric shows a lower transaction throughput (Figure 4(a)) and a lower total number of successful transactions (Figure 4(c)). On the other hand, FabricCRDT has a higher commit latency in comparison to Fabric (cf. Figure 4(b)).



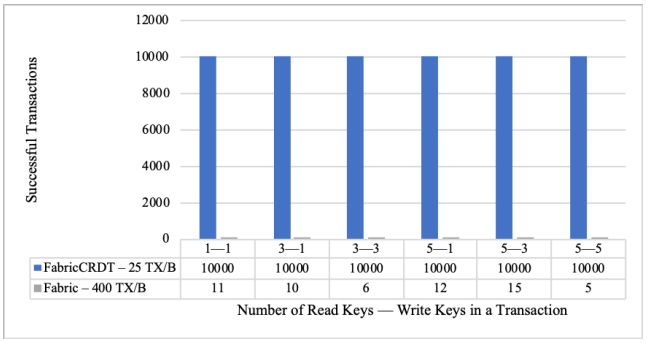
	1—1	3—1	3—3	5—1	5—3	5—5
FabricCRDT – 25 TX/B	264	205	157	189	135	106
Fabric – 400 TX/B	0.4	0.3	6.1	2.2	0.4	0.3

(a) Successful transactions throughput per second for different number of read-write keys.



	1—1	3—1	3—3	5—1	5—3	5—5
FabricCRDT – 25 TX/B	2.7	12	20	17	32	43
Fabric – 400 TX/B	5.3	4	7.1	8.4	14.3	9.6

(b) Average latency of successful transactions for different number of read-write keys.



	1—1	3—1	3—3	5—1	5—3	5—5
FabricCRDT – 25 TX/B	10000	10000	10000	10000	10000	10000
Fabric – 400 TX/B	11	10	6	12	15	5

(c) Number of successful transactions for different number of read-write keys.

Figure 4. Effect of different number of reads and writes in transactions on the throughput, latency, success rate of FabricCRDT and Fabric.

7.5 Impact of Varying Complexity of JSON Objects

In contrast to the experiments in Section 7.4, here, we evaluate the effect of varying complexity of JSON objects that are written to the ledger. In particular, we study how the throughput and latency of FabricCRDT changes, as merging more complex JSON objects induces more overhead. Table 3 shows the configuration of this experiment. Each transaction reads one JSON object from ledger with a certain number of keys and a certain nesting depth of the values; then, the transaction modifies the JSON object and writes it back to the ledger. Listing 4 exemplifies a JSON object with "3-3 complexity", i.e., the transaction submits a JSON object with three

key-value pairs, where each value has a depth of three from the root of the JSON object.

Table 3. Configuration for evaluating the impact of different complexity of JSON objects.

Parameters	Value
Transaction submission rate per second	300
Number of read keys per transaction	1
Number of write keys per transaction	1

Observations - Figure 5 summarizes the results of the experiments. Similar to the experiments in Section 7.4, we observe that the throughput decreases and the latency increases for FabricCRDT with an increasing complexity of JSON objects (cf. Figure 5(a) and Figure 5(b)). Unlike FabricCRDT, Fabric does not interact with the content of the JSON objects. Therefore, the throughput and latency of Fabric are not correlated to the complexity of the JSON objects.

Listing 4. A sample JSON object with "3-3" complexity.

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

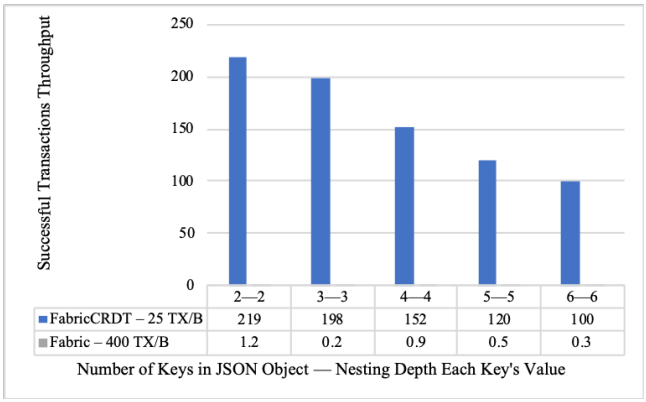
7.6 Impact of Different Transaction Arrival Rates

Further, we evaluate the effect of different transaction arrival rates on FabricCRDT and Fabric. We configured each experiment according to Table 4. We employ four clients in total, where all clients together submit transactions with a rate of 100 to 500 transactions per second.

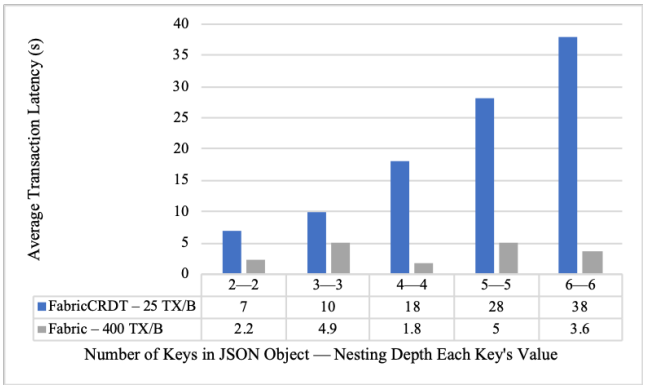
Table 4. Configuration for evaluating the impact of different transaction arrival rates.

Parameters	Value
Number of read keys per transaction	1
Number of write keys per transaction	1
Number of keys in JSON objects	2

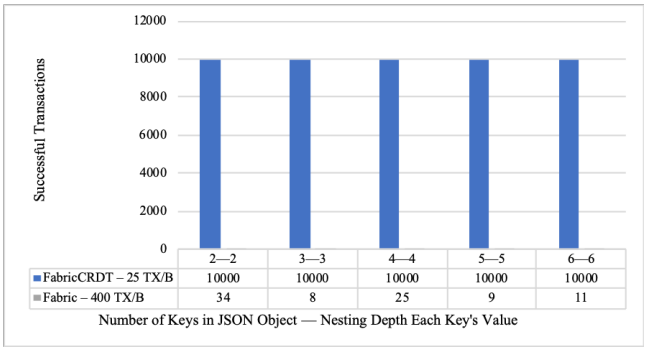
Observations - As the results of the experiments in Figure 6(a) show, FabricCRDT's throughput increases until it reaches a saturation point at about 250 transactions per second. Meanwhile, Figure 6(b) shows that the latency increases as the transaction arrival rate increases for FabricCRDT. The enormous increase in latency in FabricCRDT can be attributed to the effects of queuing when the transaction arrival rate exceeds the throughput.



(a) Successful transactions throughput per second for different JSON complexities.



(b) Average latency of successful transactions for different JSON complexities.

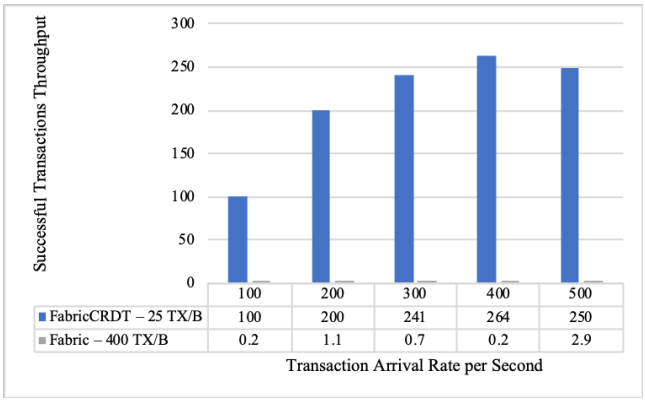


(c) Number of successful transactions for different JSON complexities.

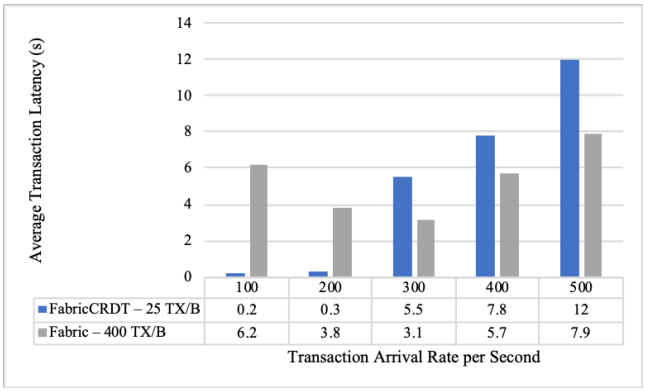
Figure 5. Effect of the complexity of JSON objects in the transactions on the throughput, latency and success rate of FabricCRDT and Fabric.

7.7 Impact of Different Percentage of Conflicting Transactions

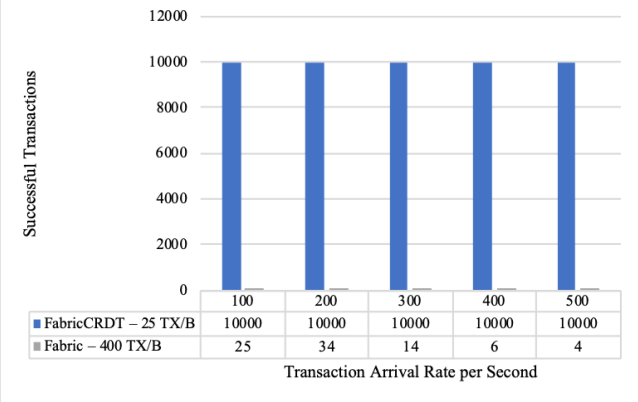
In order to understand the limitations and potentials of FabricCRDT, in the previous experiments, we used workloads where all transaction are conflicting. However, for production deployment of Fabric and FabricCRDT, where different applications are hosted,



(a) Successful transactions throughput per second for different transaction arrival rates.



(b) Average latency of successful transactions for different transaction arrival rates.



(c) Number of successful transactions for different transaction arrival rates.

Figure 6. Effect of different transaction arrival rates on the throughput, latency, and success rate of FabricCRDT and Fabric.

blocks may contain both conflicting and non-conflicting transactions. To study the effects of different percentages of conflicting transaction in the workload, we configured experiments as specified in Table 5. For each experiment, a fixed percentage of transactions are conflicting, where the conflicting transactions are merged in FabricCRDT and rejected in Fabric.

Table 5. Configuration for evaluating the impact of percentage of conflicting transactions in the workload.

Parameters	Value
Transaction submission rate per second	300
Number of read keys per transaction	1
Number of write keys per transaction	1
Number of keys per JSON object	2

Observations - Figure 7 summarizes the results of the experiment. We observe for workloads, where a smaller percentage of transactions are conflicting, that the throughput and latency of FabricCRDT are similar to Fabric (Figure 7(a) and Figure 7(b)). However, when the percentage of conflicting transaction increases, the number of failures also increases in Fabric (Figure 7(c)), while no failures occur in FabricCRDT.

7.8 Summary of Results and Discussion

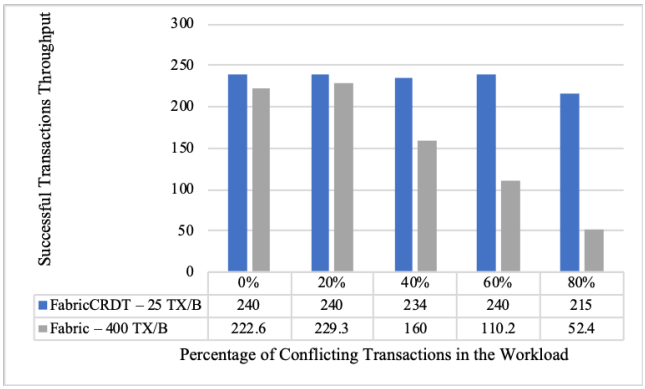
Since FabricCRDT bypasses the MVCC validation and merges the conflicting transactions instead of rejecting them, it manages to commit all transactions in all experiments successfully. In stark contrast to this, Fabric only successfully a very few transactions when all transactions are conflicting. We argue that the performance of FabricCRDT is an improvement to Fabric, since handling such a large amount of the transaction failures in the application may be a significant burden and increases the complexity of developing Fabric applications.

In our experiments, we observed that FabricCRDT, in comparison to Fabric, suffers from a higher latency, which is a direct result of the extra processing required for merging a large number of JSON CRDTs. For adding each key in the JSON object to a JSON CRDTs, metadata has to be created, and the complexity of JSON CRDTs increases when more keys and values are added. However, in most parts of our evaluation, we investigated the worst-case scenarios where all transactions in a block are conflicting and are required to be merged. For scenarios where conflicting and non-conflicting transactions coexist, the results of experiments show that Fabric and FabricCRDT have comparable latency and throughput.

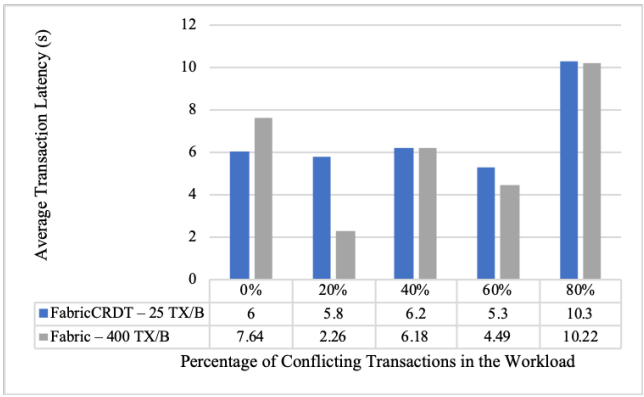
8 Related Work

Various CRDTs have been proposed and implemented in production-grade collaborative editing tools and distributed databases [8, 27, 36, 40]. Although these works offer practical solutions for improving the scalability and enhancing the user experience, only a few works investigate the applicability of CRDTs on blockchains. In Ref. [21], the authors propose Vegvisir, a directed acyclic graph (DAG)-structured blockchain for CRDT-enabled applications. Vegvisir offers a power-efficient blockchain for IoT devices which tolerates network partitioning, but it only supports applications that can be implemented completely in CRDTs. Furthermore, a proposal has been introduced by Fabric developers to enhance Fabric's concurrency control by using built-in plugins for parallel execution of basic updates such as incrementing or decrementing values [19]. However, the implementation of this proposal has not been released, and the available information on the proposal is limited and lacks technical details.

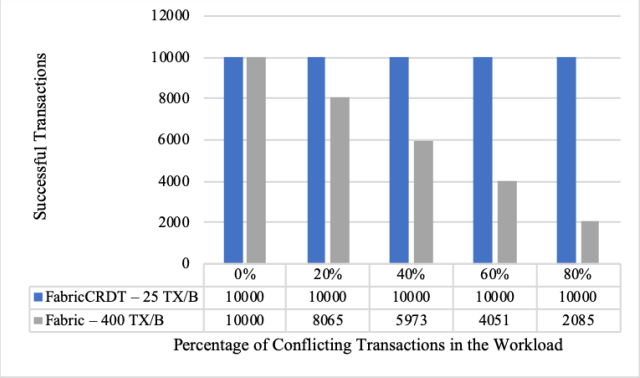
Some works investigated various approaches for improving the performance and throughput of Fabric. In Ref. [34], the authors



(a) Successful transactions throughput per second for different percentage of conflicting transactions in the workload.



(b) Average latency of successful transactions for different percentage of conflicting transactions in the workload.



(c) Number of successful transactions for different percentage of conflicting transactions in the workload.

Figure 7. Effect of different percentage of conflicting transactions in the workload on the throughput, latency, and success rate of FabricCRDT and Fabric.

use transaction reordering techniques [12] inspired by databases to improve the throughput of Fabric and to early abort conflicting transactions. They decrease the number of conflicting transactions by improving the order of the transactions in the ordering service according to a dependency graph. Although they show that reordering is a practical approach for decreasing transaction failures, they do not aim for the total elimination of failures, as FabricCRDT does. Several works focused on identifying different bottlenecks of Fabric and offering solutions [4, 6, 16, 18, 37]. In Ref. [6, 18], the authors offer solutions for improving the performance issues of the ordering service. The authors of StreamChain [18] propose an approach for replacing Fabric's block processing mechanism with stream transaction processing to decrease the end-to-end latency of committing transactions. In Ref. [6], the authors propose a new Byzantine fault-tolerant protocol for the ordering service to increase the throughput of Fabric by decreasing the message communication overhead. The authors of Ref. [16, 37] offer extensive analysis and re-architecting guidelines of Fabric to improve several bottlenecks, including the consensus mechanism, I/O and computational overhead for ordering and validating transactions and repeated validation of certificates for endorsement policies. They implemented improvements such as the parallelization of several Fabric processes, separation of different resources, and caching. All these works provide valuable insights into different approaches that can improve the performance of Fabric. As FabricCRDT reuses several of Fabric's components, these approaches are also applicable to FabricCRDT. However, none of these works provide a solution for dealing with transaction failures of concurrent updates directly.

9 Conclusion

In this work, we introduced an approach for integrating CRDTs with Hyperledger Fabric. We presented FabricCRDT, an extension of Fabric, that successfully commits transactions that perform concurrent updates and automatically merges conflicting transactions by using CRDT techniques without losing updates. We conducted extensive evaluations to understand how FabricCRDT performs in comparison to Fabric. According to our findings, FabricCRDT successfully merges all conflicting transactions without any failures when all transactions use CRDTs. In general, FabricCRDT offers higher throughput than Fabric but also induces higher commit latency due to the added overhead of merging CRDTs.

In future work, we plan to extend FabricCRDT with more CRDTs, such as list, map, and graph CRDTs. We also investigate the effect of eliminating the ordering service to understand the potentials of a purely CRDT-based permissioned blockchain.

Acknowledgments

We would like to sincerely thank Kaiwen Zhang and our shepherd, Marko Vukolić, for their valuable input to this work. We also thank the Alexander von Humboldt Foundation for supporting this project in part.

References

[1] M. Ahmed-Nacer, C. L. Ignat, G. Oster, H. G. Roh, and P. Urso. 2011. Evaluating CRDTs for Real-time Document Editing. In *Proceedings of the 11th ACM Symposium on Document Engineering*. ACM, 103–112. https://doi.org/10.1145/2034691.2034717

[2] M. Ahmed-Nacer, U. Pascal, B. Valter, and P. Nuno. 2014. Merging OT and CRDT Algorithms. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*. ACM, 9:1–9:4. https://doi.org/10.1145/2596631.2596636

[3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 30:1–30:15. https://doi.org/10.1145/3190508.3190538

[4] A. Baliga, N. Solanki, S. Verekar, A. Pednekar, P. Kamat, and S. Chatterjee. 2018. Performance Characterization of Hyperledger Fabric. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. IEEE, 65–74. https://doi.org/10.1109/CVCBT.2018.00013

[5] P. A Bernstein and E. Newcomer. 2009. *Principles of transaction processing*. Morgan Kaufmann.

[6] A. Bessani, J. Sousa, and M. Vukolić. 2017. A Byzantine Fault-tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. ACM, 6:1–6:2. https://doi.org/10.1145/3152824.3152830

[7] T. Bocek, B. B. Rodrigues, T. Strasser, and B. Stiller. 2017. Blockchains everywhere - a use-case of blockchains in the pharma supply-chain. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 772–777. https://doi.org/10.23919/INM.2017.7987376

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

[9] V. Buterin et al. 2013. Ethereum White Paper. *GitHub repository* (2013). https://github.com/ethereum/wiki/wiki/White-Paper Accessed: 2019-05-06.

[10] Transaction Processing Performance Council. 2019. TPC-C. http://www.tpc.org/tpcc/ Accessed: 2019-05-16.

[11] Transaction Processing Performance Council. 2019. TPC-H. http://www.tpc.org/tpch/ Accessed: 2019-05-16.

[12] B. Ding, L. Kot, and J. Gehrke. 2018. Improving Optimistic Concurrency Control Through Transaction Batching and Operation Reordering. *Proc. VLDB Endow.* 12, 2 (2018), 169–182. https://doi.org/10.14778/3282495.3282502

[13] J. Eberhardt and S. Tai. 2017. On or Off the Blockchain? Insights on Off-Chaining Computation and Data. In *European Conference on Service-Oriented and Cloud Computing*. Springer International Publishing, 3–15.

[14] Apache Software Foundation. 2019. CouchDB. https://couchdb.apache.org/ Accessed: 2019-05-06.

[15] C. Gorenflo, L. Golab, and S. Keshav. 2019. XOX Fabric: A hybrid approach to transaction execution. *CoRR* abs/1906.11229 (2019). http://arxiv.org/abs/1906.11229

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

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

[18] Z. István, A. Sorniotti, and M. Vukolić. 2018. StreamChain: Do Blockchains Need Blocks?. In *Proceedings of the 2Nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. ACM, 1–6. https://doi.org/10.1145/3284764.3284765

[19] Fabric Jira. 2019. Enhanced Concurrency Control. https://jira.hyperledger.org/browse/FAB-10711 Accessed: 2019-05-06.

[20] G. O. Karame, E. Androulaki, and S. Capkun. 2012. Two Bitcoins at the Price of One? Double-Spending Attacks on Fast Payments in Bitcoin. *IACR Cryptology ePrint Archive* 2012 (2012), 248.

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

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

[23] M. Kleppmann and A. R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 2733–2746. https://doi.org/10.1109/TPDS.2017.2697382

[24] R. Kumar. 2018. When to Use a CRDT-Based Database. https://www.infoworld.com/article/3305321/when-to-use-a-crdt-based-database.html Accessed: 2019-09-06.

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

[26] S. Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system.

[27] B. Nédelec, P. Molli, A. Mostefaoui, and E. Desmontils. 2013. LSEQ: An Adaptive Structure for Sequences in Distributed Collaborative Editing. In *Proceedings of the 2013 ACM Symposium on Document Engineering*. ACM, 37–46. https://doi.org/10.1145/2494266.2494278

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

[29] N. Preguiça, C. Baquero, and M. Shapiro. 2018. *Conflict-Free Replicated Data Types CRDTs*. Springer International Publishing, 1–10. https://doi.org/10.1007/978-3-319-63962-8_185-1

[30] Hyperledger Project. 2019. Hyperledger Caliper. https://hyperledger.github.io/caliper/ Accessed: 2019-05-06.

[31] The Linux Foundation Project. 2019. Hyperledger Fabric. https://www.hyperledger.org/projects/fabric Accessed: 2019-05-06.

[32] SAP. 2019. SAP HANA. https://www.sap.com/products/hana.html Accessed: 2019-05-06.

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

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

[35] A. Sharma, F. M. Schuhknecht, and J. Dittrich. 2018. Accelerating Analytical Processing in MVCC Using Fine-Granular High-Frequency Virtual Snapshotting. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 245–258. https://doi.org/10.1145/3183713.3196904

[36] SoundCloud. 2019. Roshi: A Large-scale CRDT Set Implementation for Timestamped Events. https://github.com/soundcloud/roshi Accessed: 2019-05-06.

[37] P. Thakkar, S. Nathan, and B. Viswanathan. 2018. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform. *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2018). https://doi.org/10.1109/mascots.2018.00034

[38] F. Tian. 2017. A supply chain traceability system for food safety based on HACCP, blockchain amp; Internet of things. In *2017 International Conference on Service Systems and Service Management*. IEEE, 1–6. https://doi.org/10.1109/ICSSSM.2017.7996119

[39] K. Wüst and A. Gervais. 2018. Do you Need a Blockchain?. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. IEEE, 45–54. https://doi.org/10.1109/CVCBT.2018.00011

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