Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric

Ankur Sharma
ankur.sharma@bigdata.uni-saarland.de
Big Data Analytics Group, Saarland University

Felix Martin Schuhknecht
felix.schuhknecht@bigdata.uni-saarland.de
Big Data Analytics Group, Saarland University

Divya Agrawal
s8diagra@stud.uni-saarland.de
Big Data Analytics Group, Saarland University

Jens Dittrich
jens.dittrich@bigdata.uni-saarland.de
Big Data Analytics Group, Saarland University

ABSTRACT

Within the last few years, a countless number of blockchain systems have emerged on the market, each one claiming to revolutionize the way of distributed transaction processing in one way or the other. Many blockchain features, such as byzantine fault tolerance, are indeed valuable additions in modern environments. However, despite all the hype around the technology, many of the challenges that blockchain systems have to face are fundamental transaction management problems. These are largely shared with traditional database systems, which have been around for decades already.

These similarities become especially visible for systems, that blur the lines between blockchain systems and classical database systems. A great example of this is Hyperledger Fabric, an open-source permissioned blockchain system under development by IBM. By implementing parallel transaction processing, Fabric's workflow is highly motivated by optimistic concurrency control mechanisms in classical database systems. This raises two questions: (1) Which conceptual similarities and differences do actually exist between a system such as Fabric and a classical distributed database system? (2) Is it possible to improve on the performance of Fabric by transitioning technology from the database world to blockchains and thus blurring the lines between these two types of systems even further? To tackle these questions, we first explore Fabric from the perspective of database research, where we observe weaknesses in the transaction pipeline. We then solve these issues by transitioning well-understood database concepts to Fabric, namely transaction reordering as well as early transaction abort. Our experimental evaluation under the Smallbank benchmark as well as under a custom workload shows that our improved version Fabric++ significantly increases the throughput of successful transactions over the vanilla version by up to a factor of 12x, while decreasing the average latency to almost half.

CCS CONCEPTS

• **Information systems → Distributed database transactions**;

ACM Reference Format:
Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, Amsterdam, The Netherlands, 18 pages. https://doi.org/10.1145/3299869.3319883

1 INTRODUCTION

Blockchains are one of the hottest topics in modern distributed transaction processing. However, from the perspective of database research, one could raise the question: what makes these systems so special over classical distributed databases, that have been out there for a long time already?

The answer lies in byzantine fault tolerance: while classical distributed database systems require a trusted set of participants, blockchain systems are able to deal with a certain amount of *maliciously* behaving nodes. This feature opens lots of new application fields such as transactions between organizations, that do not fully trust each other.

Regarding the aspect of byzantine fault tolerance, blockchain systems have a clear advantage over distributed database systems. Unfortunately, with respect to essentially any other aspect of transaction processing, classical database systems are decades ahead of blockchain systems.

A great example for this is the *order-execute* transaction processing model, that prominent systems like Bitcoin [1] and Ethereum [2] implement: In the ordering phase, all peers first agree on a global transaction order, typically using a consensus mechanism. Then, each peer locally executes the

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06.
https://doi.org/10.1145/3299869.3319883

transactions in that order on a replica of the state. While this approach is simple, it has two severe downsides: First, the execution of transactions happens in a sequential fashion. Second, as every transaction must be executed on every peer, the performance of the system does not scale with the number of peers. Of course, both parallel execution as well as scaling capabilities have been well-established properties of distributed database systems since many years.

1.1 Catching up

Still, there are blockchain systems that try to catch up. A prominent example for this is Hyperledger Fabric [10], a popular open-source blockchain system introduced by IBM. Instead of implementing the order-execute model, it follows a sophisticated *simulate-order-validate-commit* model. This model is highly influenced by optimistic concurrency control mechanisms in database systems: Transactions are simulated speculatively in parallel *before* actually ordering them. Then, after ordering, Fabric checks in the validation phase whether the order does not conflict with the previously computed simulation effects. Finally, the effects of non-conflicting transactions are committed. The advantages of this model are clear: parallel transaction execution and therefore the ability to scale — features, which will be mandatory for any upcoming blockchain system, that aims at high performance.

We strongly believe that Fabric's ambitions in transitioning technology from the world of databases to blockchains are a step in the right direction. Unfortunately, its implementation of parallel transaction processing still suffers from certain problems which highly limit the gain, that can be achieved by concurrency. These problems can be identified easily in two simple experiments.

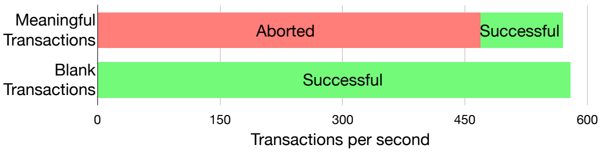


Figure 1: Transactions per second of vanilla Fabric when meaningful transactions are fired as described in Section 6 for the configuration BS=1024, RW=8, HR=40%, HW=10%, HSS=1%. Additionally, we show the throughput when blank transactions are fired.

In the first experiment (Figure 1, top bar), we submit a stream of meaningful transactions, which originate from an asset transfer scenario, and report the throughput, divided into aborted and successful transactions. This experiment reveals a severe problem of Fabric: a large number of transactions end up as being aborted. The reason for all these aborts are serialization conflicts, a negative side-effect of concurrent execution.

If we want to increase the number of successful transactions, we essentially have two options: Either we (a) increase the overall throughput of the system or (b) turn transactions, that would have been aborted by Fabric, to successful ones. Unfortunately, option (a) is hardly applicable in Fabric. We can see this in the second experiment (Figure 1, bottom bar), where we submit blank transactions without any logic. Interestingly, the total throughput of blank and meaningful transactions essentially equals. This reveals, that the overall throughput of the system is not dominated by the core components of transaction processing, but actually by other auxiliary factors: cryptographic computations and networking overhead.

1.2 Fabric++

Thus, option (b) is the key: we have to turn transactions, that would have been aborted by Fabric, to successful ones. We achieve this, by transitioning a well-known technique from database systems to Fabric: *transaction reordering*. Instead of arbitrarily ordering transactions, we inspect the transaction semantics and arrange the transactions in a way such that the number of serialization conflicts is drastically reduced. Furthermore, we remove transactions, that have no chance to commit anymore, as early as possible from the pipeline. This *early abort of transactions* further improves the situation, as transactions, which have no chance to commit, are out of consideration for reordering.

In total, we carry out the following steps to further "databasify" Fabric:

(1) To have a basis for the discussion, we first inspect the transaction flow of Hyperledger Fabric version 1.2 from a conceptual perspective. (Section 2).

(2) We carefully inspect the related work in the area of concurrency control and discuss, which techniques are related to Fabric and which are out of consideration. (Section 3). This leads us directly to the techniques we are integrating.

(3) Based on the analysis of the transaction flow in Fabric, we discuss its weaknesses in detail and describe, how database technology can be utilized to counter them. (Section 4).

(4) We transition database technology to the transaction pipeline of Fabric. Precisely, we first improve on *the ordering of transactions*. By default, the system orders transactions arbitrarily after simulation, leading to unnecessary serialization conflicts. To counter this problem, we introduce an advanced *transaction reordering mechanism*, which aims at reducing the number of serialization conflicts between transactions within a block. This mechanism significantly increases the number of valid transactions, that make it through the system and therefore the overall throughput (Section 5.1).

(5) Next, we advance the *abort of transactions*. By default, Fabric checks whether a transaction is valid right before the commit. This late abort unnecessarily penalizes the system by processing transactions, that have no chance to commit. To tackle this issue, we introduce the concept of *early abort* to various stages of the pipeline. We identify invalid transactions as early as possible and abort them, assuring that the pipeline is not throttled by transactions that have no chance to commit eventually. A requirement for this concept is a *fine-grained concurrency control mechanism*, by which we extend Fabric as well (Section 5.2). These modifications significantly extend the vanilla Fabric, turning it into what we call Fabric++.

(6) We perform an extensive experimental evaluation of the optimizations of Fabric++ under the Smallbank benchmark as well as under a custom workload. We show that we are able to significantly increase the number of successful transactions over the vanilla version. Additionally, we vary the blocksize, the number of channels, and clients to show that our optimizations also have a positive impact on the scaling capabilities of the system. Further, using the Caliper benchmark, we show that Fabric++ also produces a lower transaction latency than vanilla Fabric (Section 6).

2 HYPERLEDGER FABRIC

First, we have to understand the workflow of Fabric. Let us describe in the following section how it behaves in version 1.2.

2.1 Architecture

Fabric is a *permissioned* blockchain system, meaning all peers of the network are known at any point in time. Peers are grouped into *organizations*, which typically host them. Within an organization, all peers trust each other. Each peer runs a local instance of Fabric. This instance includes a copy of the *ledger*, containing the ordered sequence of all transactions that went through the system. This includes both valid and invalid transactions. Apart from the ledger, each peer also contains the *current state* in form of a state database, which represents the state after the *application* of all *valid* transactions in the ledger to the initial state. Apart from the peers, which play an important role both in the simulation phase and the validation phase, there is a separate instance called the *ordering service*, which is the core component of the ordering phase and assumed to be trustworthy.

2.2 High-level Workflow

At its core, Fabric follows a *simulate-order-validate-commit* workflow, as shown in Figure 2.

2.2.1 Simulation Phase. In the simulation phase, a client submits a transaction proposal to a subset of the peers, called the endorsement peers or *endorsers*, for simulation. This subset of endorsement peers is defined in a so called *endorsement policy*. Since organizations do not fully trust each other, it is typically specified, that at least one peer of each involved organization has to simulate the transaction proposal. The endorsers now simulate the transaction proposal against a local copy of the current state in parallel. As the name of this phase suggests, none of the effects of the simulation become durable in the current state at this point. Instead, each endorser builds up a read set and a write set during simulation to capture the effects. After simulation, each endorser returns its read and write set to the client. Along with that, the endorsers also return a cryptographic signature over the sets. If all returned read and write sets are equal, the client forms an actual transaction. It contains the previously computed read set and write set along with all signatures. The client then passes this transaction on to the ordering service.

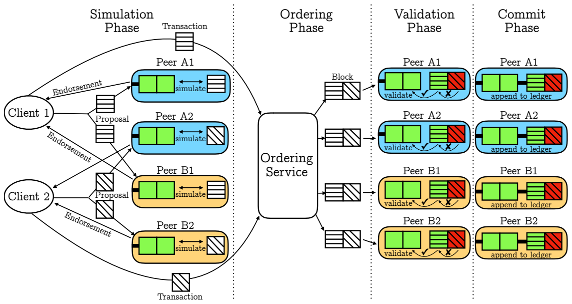


Figure 2: High-level workflow of Fabric.

2.2.2 Ordering Phase. In the ordering phase, the trusted ordering service receives the transactions from the clients. Among all received transactions, it establishes a global order and packs them into blocks containing a certain number of transactions. By default, the transactions are essentially ordered in the way in which they arrive at the service, without inspecting the transaction semantics in any way. The ordering service then distributes each formed block to *all* peers of the network. Note that the system does not guarantee that all peers receive a block at the same time. However, it guarantees that all peers receive the same blocks in the same order.

2.2.3 Validation Phase. As soon as a block arrives at a peer, its validation phase starts. For each transaction within the block, the validation consists of two checks: First, Fabric tests whether the transaction respects the endorsement policy and whether all contained signatures fit to the read and write set. If this is not the case, it means that either an endorser or the client tampered with the transaction in some way. In this case, the systems marks the transaction as invalid. If a transaction passes the first test, Fabric secondly checks, whether any serialization conflicts occur. As the simulation of transactions happens in parallel *before* ordering them, it is possible that the effects of the simulation stand in conflict with the established order. Therefore, Fabric marks transactions, which conflict with previous transactions, as invalid as well.

2.2.4 Commit Phase. In the commit phase, each peer appends the block, which contains both valid and invalid transactions, to its local ledger. Additionally, each peer applies all changes made by the valid transactions to its current state.

3 RELATED WORK

Before diving into the optimizations we apply, we have to discuss related work in the field.

In this work, we transition mature database techniques to the world of blockchains. As mentioned in the introduction, we essentially apply two prominent techniques from the field of database concurrency control to Fabric: *transaction reordering* and *early transaction abort*. Of course, concurrency control is a large and active area of research and the alerted reader might wonder, why we focus on precisely these techniques. The answer lies in the fact, that a blockchain system such as Fabric, despite being a parallel transaction processing system, differs from parallel database systems in four points:

(a) A blockchain system like Fabric commits at the granularity of blocks instead of committing at the granularity of individual transactions. The commit granularity has a significant effect on the type of applicable optimizations.

(b) A blockchain system like Fabric is a distributed system, where the state is fully replicated across the network and where transactions operate on all nodes. In contrast to that, parallel database systems are typically either installed locally on a single node or partition their state across a network, such that transactions operate on a subset of the network. The type of distribution and replication has a significant effect on the optimization options.

(c) In Fabric, a single transaction is simulated in parallel on multiple nodes to establish trust. The state, against which the transaction is simulated potentially differs across nodes. In a parallel database system, the situation is way simpler: a transaction is executed exactly once against the only state present in the system.

(d) The performance of a blockchain system like Fabric is largely dominated by cryptographic signature computations, network communication, and trust validation. In contrast, the performance of parallel database systems is highly influenced by low-level components, such as the choice of the locking mechanism for concurrency control. Therefore, despite being closely related, blockchain optimizations happen on a different level than database optimizations.

Let us now go through the related work. Essentially, concurrency control techniques can be divided into two classes: (1) methods, that aim at improving the overall transactional throughput [13, 17, 20, 26] and (2) methods, that try to turn aborted transactions into successful transactions [24, 28].

3.1 Class 1: Transaction Throughput

The works of [26], [20], and [13] all aim at improving parallel transaction execution. In [26], the authors propose a mechanism, which collects a batch of transactions and analyzes the access dependencies between these transactions. The resulting dependency graph is then partitioned into non-intersecting subgraphs. As the transactions in different subgraphs do not conflict with each other, they can be safely executed in parallel. [20] and [13] go a step further by not only analyzing dependencies between entire transactions, but actually between transaction-fragments. Precisely, they first split the transactions into possible fragments and then analyze the dependencies between these fragments, while respecting existing dependencies between entire transactions. By this, they are able to achieve a higher degree of parallelism. In general, a design as proposed by these three papers allows to equip systems, that would otherwise follow a purely sequential execution, with a partial parallel execution. In the context of blockchains, systems following an order-execute model could benefit from such a technique. Obviously, Fabric is not the right candidate for this method, as it already parallelizes the simulation phase by default.

[17] aims at improving concurrency control from a low-level perspective. It proposes interesting optimizations to MVCC components such as timestamp allocation, version storage, validation, index management, and recovery. In theory, these techniques could be applied to the underlying storage system of Fabric. However, improving low-level components of Fabric will not improve the overall performance, which is largely dominated by top-level components handling cryptography, networking, and trust validation.

Unfortunately, the techniques of class (1) are not suited to improve the transactional throughput of a blockchain system such as Fabric. We saw the reason for this in Figure 1 in the introduction. For blank transactions, the concurrency control mechanism essentially has no work to do. For meaningful transactions, simulation and validation must be synchronized. Still, the throughput equals. This means, that a technique that directly affects transactional processing, such as concurrency control, can *not* lead to an improvement in throughput. Instead, the system is dominated by factors, that are *not* directly related to transactional processing, such as cryptographical computations and networking. As a consequence, optimizations of class (1) are out of consideration.

3.2 Class 2: Transaction Abort & Success

[24] relates to our work, as it shares the same motivation: to reduce the amount of transactions, that are unnecessarily aborted due to serialization conflicts. In the context of a local parallel database system implementing multi-version concurrency control (MVCC) [16, 21, 25, 27], the authors propose to protect each frequently accessed entry additionally with a shared lock. The use of such a lock prevents unnecessary aborts due to read-write conflicts between transactions accessing these hot entries. Effectively, this need to acquire a lock assures for potentially conflicting transactions, that they commit in a non-conflicting order. Unfortunately,

this strategy is hardly applicable to the distributed transaction processing model of Fabric. Since Fabric simulates and commits transactions in parallel on multiple nodes, this technique would require a trusted fine-grained distributed locking service to synchronize accesses, causing excessive network communication and coordination effort.

In [28], the authors also aim at reducing the number of aborted transactions by influencing the commit order. However, they follow a completely different strategy than [24]: When a transaction T wants to commit and detects a read-write conflict with an already committed transaction, then it is not directly aborted. Instead, it is checked whether the commit time of T can be simply changed retrospectively to its begin time. If this change does not trigger a read-write, write-write, or a write-read conflict with another concurrent transaction, then T is allowed to commit at its begin time. While this technique would be applicable in the ordering service of Fabric, its effects would be highly limited. This is caused by the simplicity of the method which allows the commit time of a transaction to be changed only to its begin time, not to other possible points in time within the commit sequence. This wastes a lot of optimization potential. In contrast, our transaction reordering mechanism considers commit reordering for all transactions within a block and aims at finding the best global order.

The benefit of transaction reordering has also recently been studied in [11]. In the context of OLTP systems, the authors identify that the number of successful transaction can be improved by up to a factor of 2.7x via reordering of transaction batches.

Thus, class (2) methods, which aim at increasing the number of successful transactions and clearing the ones that must be aborted, are the key for improving a blockchain system such as Fabric. Our optimizations of transaction reordering and early transaction abort fall into this class. In the following section, we will elaborate the importance of these techniques in detail.

4 BLURRED LINES: FABRIC VS DISTRIBUTED DATABASE SYSTEMS

With an understanding of the workflow of Fabric, we are able to discuss its architecture in relation to distributed database systems. In particular, we are interested in aspects of Fabric, that are (a) conceptually shared with distributed database systems, but (b) have potential for the application of database technology.

4.1 The Importance of Transaction Order

The first component we look at is the ordering mechanism. Such a component is also present in any distributed database system with transaction semantics and therefore a great candidate for transitioning database technology to Fabric.

As described in Section 2, Fabric relies on a single trustworthy ordering service for ordering transactions. Since Fabric simulates the smart contracts bound to proposals *before* performing the ordering, the order actually has an influence on the number of serialization conflicts between transactions. Again, this is a property shared with any parallel database system, that separates transaction execution from transaction commit.

In ordering transactions, various different strategies are possible: The simplest option is to arbitrarily order them, for instance in the order in which they arrive. While this arrival order is fast to establish, it can lead to serialization conflicts, that are *potentially unnecessary*. These conflicts increase the number of invalid transactions, which must be resubmitted by the client. Unfortunately, the vanilla Fabric follows exactly this naive strategy. This is caused by the design decision that the ordering service is not supposed to inspect the transaction semantics, such as the read and write set, in any way. Instead, it simply leaves the transactions in the order in which they arrive. This strategy can be problematic, as the example in Table 1 shows. In this example, four transactions are scheduled in the order in which they arrive, namely $T_1 \Rightarrow T_2 \Rightarrow T_3 \Rightarrow T_4$, where T_1 updates the key k_1 from version v_1 to v_2. Since the transactions T_2, T_3, and T_4 each read k_1 in version v_1 during their simulations, they have no chance to commit, as they operated on an outdated version of the value of k_1. They will be identified as invalid in the validation phase and the corresponding transaction proposals must be resubmitted by the client, resulting in a new round of simulation, ordering, and validation.

Table 1: For the order $T_1 \Rightarrow T_2 \Rightarrow T_3 \Rightarrow T_4$, only one out of four transactions is valid: T_2, T_3, and T_4 read the outdated version v_1 of key k_1, that has been updated by T_1 to v_2 before.

Transaction	Read Set	Write Set	Is Valid?
1. T_1	–	$(k_1, v_1 \to v_2)$	✓
2. T_2	(k_1, v_1), (k_2, v_1)	$(k_2, v_1 \to v_2)$	×
3. T_3	(k_1, v_1), (k_3, v_1)	$(k_3, v_1 \to v_2)$	×
4. T_4	(k_1, v_1), (k_3, v_1)	$(k_4, v_1 \to v_2)$	×

Interestingly, for the four transactions from the previous example, there exists an order that is conflict free. In the schedule $T_4 \Rightarrow T_2 \Rightarrow T_3 \Rightarrow T_1$, as shown in Table 2, all four transactions are valid, as their read and write sets do not conflict with each other in this order.

This example shows that the vanilla orderer of Fabric misses a chance of removing *unnecessary* serialization conflicts. While this problem is new to the blockchain domain, as blockchains typically offer only a serial execution of transactions, within the database community, this problem is actually well known. There exist reordering mechanisms which aim at minimizing the number of serialization conflicts via a

Table 2: The order $T_4 \Rightarrow T_2 \Rightarrow T_3 \Rightarrow T_1$ results in all four transactions being valid.

Transaction	Read Set	Write Set	Is Valid?
1. T_4	$(k_1, v_1), (k_3, v_1)$	$(k_4, v_1 \rightarrow v_2)$	✓
2. T_2	$(k_1, v_1), (k_2, v_1)$	$(k_2, v_1 \rightarrow v_2)$	✓
3. T_3	$(k_1, v_1), (k_3, v_1)$	$(k_3, v_1 \rightarrow v_2)$	✓
4. T_1	–	$(k_1, v_1 \rightarrow v_2)$	✓

reordering of transactions [11, 18, 29, 30]. However, in a database system, it is typically avoided to buffer a large number of incoming transactions before processing as low latency is mandatory. Thus, reordering is not always an option in such a setup. Fortunately, as blockchain systems buffer the incoming transactions anyways to group them into blocks, this gives us the opportunity to apply sophisticated transaction reordering mechanisms without introducing significant overhead.

We will add such a transaction reordering mechanism to Fabric in Section 5.1, which significantly enhances the number of valid transactions, that make it through the system.

4.2 On the Lifetime of Transactions

The second aspect we look at from a database perspective tackles the lifetime of transactions within the pipeline. In Fabric, every transaction that goes through the system is either classified as valid or as invalid with respect to the validation criteria. In the vanilla version, this classification happens in the validation phase right before the commit phase. A severe downside of this form of *late abort* is that a transaction, that violated the validation criteria already in an earlier phase, is still processed and distributed across all peers. This penalizes the whole system with unnecessary work, throttling the performance of valid transactions. Besides, this concept also delays the abort notification to the client.

We have to distinguish in which phase a violation happens. First, a violation can occur already in the simulation phase, in form of so called *cross-block conflicts*, meaning a transaction from a later block, which is currently in the simulation phase, conflicts with a valid transaction from an earlier block. Second, a violation can occur as well as in the ordering phase, in form of *within-block conflicts* between conflicting transactions in a single block.

Let us look at these two scenarios in isolation in Section 4.2.1 and Section 4.2.2, respectively.

4.2.1 Violation in the simulation phase (cross-block conflicts).

To understand the problem in the simulation phase, let us look at the following situation and how the vanilla version of Fabric handles it. Let us assume there are four transactions T_1, T_2, T_3, and T_4 that are currently in the ordering phase and that end up in a block of size four, which is shipped to all peers for validation. Before the validation of that block starts within a peer P, the smart contract of a transaction proposal T_5 starts its simulation in P. To do so, it acquires a read lock[1] on the *entire* current state. While the simulation is running, the block has to wait for the validation, as it has to acquire an *exclusive* write lock on the current state. The problem in this situation is: if T_1, T_2, T_3, or T_4 write the value of a key, that is read by T_5, then T_5 simulates on stale data. Therefore, in the moment of the read, the transaction becomes virtually invalid. Still, in the vanilla version of Fabric, this stale read is not detected before the validation phase of T_5. Thus, T_5 would continue its simulation and go through the ordering phase, just to be invalidated in the very end.

4.2.2 Violation in the ordering phase (within-block conflicts).

Apart from conflicts across blocks, there can be conflicts between transactions within a block. These conflicts appear after putting the transactions into a particular order in the ordering phase. For instance, the example from Table 1 in Section 4.1 showed a schedule, where the three transactions T_2, T_3, and T_4 individually conflict with the previously scheduled transaction T_1 of the same block. Unfortunately, these conflicts are not detected within the orderer of the vanilla version of Fabric. The block containing T_2, T_3, and T_4 would be distributed across all peers of the network for validation, although 3/4 of transactions within the block are virtually invalid. As before, this originates from the design decision that the ordering service does not inspect transaction semantics.

The mentioned situations show that Fabric misses several chances to abort transactions right at the time of violation. In contrast to that, database systems are typically very eager in aborting transactions [14], as it decreases network traffic and saves computing resources. This concept of "cleaning" the pipeline as early as possible is called *early abort* in the context of databases, which apply this concept in various flavors. For instance, besides of the early abort of transactions, that violate certain criteria, database systems eliminate records from the query result set as early as possible by pushing down selection and projection operations in the query plan.

To overcome the mentioned problems, we will apply the concept of early abort at several stages of the transaction processing pipeline of Fabric. By this, we assure to utilize the available resources with meaningful work to the extend. We will detail this in Section 5.2.

5 FABRIC++

We have outlined the problems of Fabric and how they relate to key problems known in the context of database systems. Let us now see precisely how we counter them. First, in Section 5.1, we introduce a transaction reordering mechanism, that aims at minimizing the number of unnecessary within-block conflicts. Second, in Section 5.2, we introduce early

[1] The read lock can be shared by multiple simulation phases, as they do not modify the current state.

transaction abort to several stages of the Fabric pipeline. This also involves the introduction of a fine-grained concurrency control mechanism.

5.1 Transaction Reordering

When reordering a set of transactions S, multiple challenges must be faced. First, we have to identify which transactions of S actually conflict with each other with respect to the actions they perform. Again, these conflicts can occur, as the transactions of S simulated in complete isolation from each other. As the commits of S happen in a later phase, they had no chance to see each other's potentially conflicting modifications. Precisely, we have a conflict between two transactions T_i and T_j (denoted as $T_i \not\twoheadrightarrow T_j$), if T_i writes to a key that is read by T_j. In this case, T_i must be ordered *after* T_j (denoted as $T_j \Rightarrow T_i$) to make the schedule serializable, as otherwise, the read of T_j would be outdated. Unfortunately, the problem is typically more complex as *cycles of conflicts* can occur, such that simple reordering cannot resolve the problem. For example, if we have the cycle of conflicts $T_i \not\twoheadrightarrow T_j \not\twoheadrightarrow T_k \not\twoheadrightarrow T_i$, there is no order of these three transactions that is serializable. Therefore, before reordering transactions, our mechanism must actually first remove certain transactions of S to form a cycle-free subset $S' \subseteq S$, from which a serializable schedule can be generated.

From a high-level perspective, the following five steps must be carried out: (1) First, we build the conflict graph of all transactions of S. (2) Then, we identify all cycles within this conflict graph. (3) Based on that, we identify for each transaction, in which cycles it occurs. (4) Next, we incrementally abort transactions occurring in cycles, starting from the ones that occur in most cycles, until all cycles are resolved. (5) Finally, we build a serializable schedule from the remaining transactions. The pseudo-code of Algorithm 1 implements these five steps.

5.1.1 Example. To understand the principle and to discuss some of the implementation details, let us go through a concrete example. Let us assume we have a set S of six transactions T_0 to T_5 to consider for reordering. These six transactions have read and write sets as shown in Table 3. In total, they access ten unique keys K_0 to K_9.

Step (1): Based on this information, we now have to generate the conflict graph of the transactions as done by the function `buildConflictGraph()` in line 4 of Algorithm 1. To do so in an efficient way, we interpret the rows of Table 3 as bit-vectors of length 10. Let us refer to them as $vec_r(T_i)$ for the reading accesses and $vec_w(T_i)$ for the writing accesses of a transaction T_i. For each transaction T_i, we now perform a bitwise &-operation between $vec_r(T_i)$ and $vec_w(T_j)$ for all $j \neq i$. If the result of an &-operation is not 0, we have identified a read-write conflict and create an edge in the conflict graph between the corresponding transactions. As a result,

```
1  func reordering(Transaction[] S) {
2     // Step 1: For each transaction in S buildConflictGraph()
3     // inspects the read and write set and builds a conflict graph.
4     Graph cg = buildConflictGraph(S)
5     // Step 2: Within the conflict graph, we have to identify all
6     // occurring cycles. We do that by dividing cg into strongly
7     // connected subgraphs using Tarjans algorithm [2] in
8     // divideIntoSubgraphs().
9     Graph[] cg_subgraphs = divideIntoSubgraphs(cg)
10    // In a strongly connected graph, each node is reachable from
11    // every other node. This implies that each strongly connected
12    // subgraph of cg with more than one node must contain at least
13    // one cycle. We identify the cycles within the subgraphs using
14    // Johnsons algorithm [1] in getAllCycles().
15    Cycle[] cycles = emptyList()
16    foreach subgraph in cg_subgraphs:
17       if(subgraph.numNodes() > 1):
18          cycles.add(subgraph.getAllCycles())
19    // Step 3: To remove the cycles in cg, we have to remove
20    // conflicting transactions from S. To identify the
21    // transactions that cause the most problems, for each
22    // transaction of S, we count in how many cycles it occurs.
23    MaxHeap transactions_in_cycles = emptyMaxHeap()
24    foreach Cycle c in cycles:
25       foreach Transaction t in c:
26          if transactions_in_cycles.contains(t)
27             transactions_in_cycles[t]++
28          else
29             transactions_in_cycles[t] = 1
30    // Step 4: Let us define S' as S. We now greedily remove the
31    // transaction from S' that occurs in most cycles, until all
32    // cycles have been resolved.
33    Transaction[] S' = S
34    while not cycles.empty():
35       Transaction t = transactions_in_cycles.popMax()
36       S'.remove(t)
37       foreach Cycle c in cycles:
38          if c.contains(t):
39             c.remove(t)
40             cycles.remove(c)
41          foreach Transaction t' in c:
42             transactions_in_cycles[t']--
43    // Step 5: From S' we have to form the actual serializable
44    // schedule. We start by building the (cycle-free) conflict
45    // graph of S'.
46    Graph cg' = buildConflictGraph(S')
47    // Compute schedule. We start at some node of the graph,
48    // that hasn't been visited yet.
49    Transactions[] order = emptyList()
50    Node startNode = cg'.getNextNode()
51    while order.length() < cg'.numNodes():
52       addNode = true
53       if startNode.alreadyScheduled():
54          startNode = cg'.getNextNode()
55          continue
56       // Traverse upwards to find a source
57       foreach Node parentNode in startNode.parents():
58          if not parentNode.alreadyScheduled():
59             startNode = parentNode
60             addNode = false
61             break
62       // A source has been found, so schedule it and traverse
63       // downwards.
64       if addNode:
65          startNode.scheduled()
66          order.append(startNode)
67          foreach Node childNode in startNode.children():
68             if not childNode.alreadyScheduled():
69                startNode = childNode
70                break
71    return order.invert()
72 }
```

Algorithm 1: Reordering mechanism in pseudo code.

Table 3: Ten unique keys that are accessed by six transactions, separated in read set and write set.

Transactions	Read Set									
	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9
T_0	**1**	**1**	0	0	0	0	0	0	0	0
T_1	0	0	0	**1**	**1**	**1**	0	0	0	0
T_2	0	0	0	0	0	0	**1**	**1**	0	0
T_3	0	0	**1**	0	0	0	0	0	**1**	0
T_4	0	0	0	0	0	0	0	0	0	**1**
T_5	0	0	0	0	0	0	0	0	0	0

Transactions	Write Set									
	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9
T_0	0	0	**1**	0	0	0	0	0	0	0
T_1	**1**	0	0	0	0	0	0	0	0	0
T_2	0	0	0	**1**	0	0	0	0	0	**1**
T_3	0	**1**	0	0	**1**	0	0	0	0	0
T_4	0	0	0	0	0	**1**	**1**	0	**1**	0
T_5	0	0	0	0	0	0	0	**1**	0	0

we obtain the conflict graph $C(S)$ of our six transactions as shown in Figure 3.

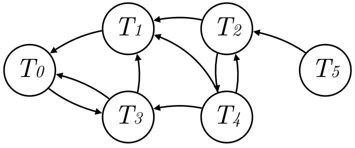


Figure 3: Conflict graph $C(S)$ of the transactions in S.

Note that this algorithm has quadratic complexity on the number of transactions. Still, we apply it as the number of transactions to consider is very small in practice due to the limitation by the block size and therefore, the overhead is negligible.

Step (2): To identify the cycles, we apply Tarjan's algorithm [22] in the function `divideIntoSubgraphs()` in line 9 to identify all strongly connected subgraphs. In general, this can be done in linear time in $O(N + E)$ over the number of nodes N and number of edges E and results in the three subgraphs as shown in Figure 4.

Using Johnson's algorithm [15], we then identify all cycles within the strongly connected subgraphs. Again, this step can be done in linear time in $O((N + E) \cdot (C + 1))$, where C is the number of cycles. Therefore, if there are no cycles in the subgraphs, the overhead of this step is very small.

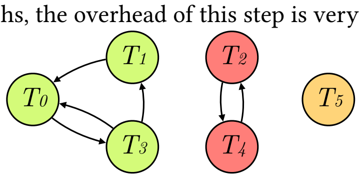


Figure 4: The three strongly connected subgraphs of the conflict graph of Figure 3.

We identify that the first subgraph (colored in green) contains the two cycles $c_1 = T_0 \twoheadrightarrow T_3 \twoheadrightarrow T_0$ and $c_2 = T_0 \twoheadrightarrow T_3 \twoheadrightarrow T_1 \twoheadrightarrow T_0$. The second subgraph (colored in red) contains the cycle $c_3 = T_2 \twoheadrightarrow T_4 \twoheadrightarrow T_2$. The third subgraph (colored in yellow) contains only one node and is thus cycle-free.

Step (3): From this information, we can build a table denoting for every transaction in which cycle it appears as shown in the lines 15 to 18 of Algorithm 1. Table 4 visualizes the result for our example. If a transaction T_i is part of a cycle c_j, the corresponding cell is set to 1, otherwise 0. The last row of the table sums up for every transaction in how many cycles it is contained in total.

Table 4: If a transaction T_i is a part of a cycle c_j, the corresponding cell is set to 1, otherwise 0. The last row contains for every transaction the total number of cycles, in which it appears.

Cycle	T_0	T_1	T_2	T_3	T_4	T_5
c_1	1	0	0	1	0	0
c_2	1	1	0	1	0	0
c_3	0	0	1	0	1	0
Σ	2	1	1	2	1	0

Step (4): We now iteratively remove transactions, that participate in cycles, starting from the ones that appear in most cycles. The lines 33 to 40 of Algorithm 1 show the corresponding pseudo-code. As we can see, T_0 and T_3 both appear in two cycles, so we take care of them first. If we can choose between two transactions, such as T_0 and T_3, we pick the one with the smaller subscript. This assures that our algorithm is deterministic. We remove T_0, which clears all cycles in which T_0 appears, namely c_1 and c_2. The transactions T_2 and T_4 remain with a participation in cycle c_3 each. We remove T_2 which clears c_3 and thereby the last cycle.

From this we now know that from the set $S' = \{T_1, T_3, T_4, T_5\}$ we can generate a serializable schedule, leading to the cycle-free conflict graph $C(S')$ (line 46) as shown in Figure 5.

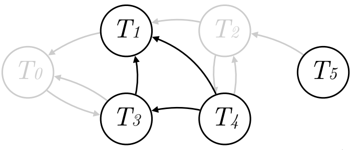


Figure 5: The cycle-free conflict graph $C(S')$, containing only the transactions T_1, T_3, T_4, and T_5.

Step (5): Generating the final schedule is essentially a repetitive execution of two parts until all nodes are scheduled: (a) the locating of the source node in the current subgraph (lines 53 to 61) and (b) the scheduling of all nodes that are reachable from that source (lines 64 to 70).

We start part (a) at the node of $C(S')$ representing the transaction with the smallest subscript, namely T_1. From this starting node, we have to find a source node, as sources have to be scheduled last. T_1 has two parents, namely T_3 and T_4, so it not a source. We follow the edge to T_3, which has not been visited yet but is also not a source, as it has T_4 as a parent as well. We follow the edge to T_4, which has not been visited yet and which is a source. Therefore, we can schedule T_4 safely at the last position in our schedule, to which we refer to as *position* 4. Now, part (b) starts as all nodes that are reachable from T_4 must be scheduled before it. T_4 has two children, namely T_1 and T_3. We follow the edge to T_1, which has not been scheduled yet. However, as T_1 has an incoming edge from T_3, we also cannot directly schedule it. First, we visit T_3 and identify that it has a parent in form of T_4, the source at which we started. With this information, we know that T_3 must be scheduled at position 3 and T_1 must be scheduled at position 2. This ends part (b), as all reachable nodes have been scheduled. Next, we restart at the only remaining node T_5. As T_5 is not only a source but also a sink, we can schedule it instantly at position 1. This results in the final schedule $T_5 \Rightarrow T_1 \Rightarrow T_3 \Rightarrow T_4$, which is returned to the orderer.

Please note that our reordering mechanism is not guaranteed to abort a minimal number of transactions, as this would be a NP-hard problem. However, it offers a very lightweight way to generate a serializable schedule with a small number of aborts.

5.1.2 Batch Cutting. In the context of transaction reordering, we have to discuss and extend a mechanism within the ordering service, that we omitted for simplicity in the description of Fabric in Section 2, namely *batch cutting*. When the ordering service receives the transactions in form of a constant stream, it decides based on multiple criteria when to "cut" a batch of transactions to finalize it and to form the block. In the vanilla version, a batch is cut as soon as one of the following three conditions hold: (a) The batch contains a certain number of transactions. (b) The batch has reached a certain size in terms of bytes. (c) A certain amount of time has passed since the first transaction of this batch was received.

In Fabric++, we extend these criteria by one additional condition. We also cut the batch, if (d) the transactions within the batch access a certain number of unique keys. This condition ensures that the runtime of our reordering mechanism, in particular the time of step (1), remains bounded.

5.2 Early Transaction Abort using Advanced Concurrency Control

The reordering mechanism previously described not only tries to minimize the number of unnecessary aborts, it also enables a form of *early abort*. Transactions, that are removed from S because of their participation in conflict cycles can be aborted already in the ordering phase instead of later on the validation phase. This assures that less transactions are distributed across the network.

In the following, we want to push this concept of aborting transactions as early as possible in the pipeline to the limits. Additionally to early aborting transactions that occur in conflict cycles, we can integrate two more applications of early abort, as we will describe in Section 5.2.1 and Section 5.2.2. The first one is happening already in the simulation phase. Let us see in the following how this works.

5.2.1 Early Abort in the Simulation Phase. To realize early abort in the simulation phase, we first have to extend Fabric by a more fine-grained concurrency control mechanism, that allows for the parallel execution of simulation and validation phase within a peer. With such a mechanism at hand, we have the chance of identifying stale reads *during* the simulation already.

To understand the concept, let us consider the example from Section 4.2.1 again. With a fine-grained concurrency control mechanism, the block containing T_1, T_2, T_3, and T_4 would not have to pend for validation while the smart contract bound to the proposal T_5 is simulating. Instead, the four transactions would apply their updates in an atomic fashion *while* T_5 is simulating. As a consequence of this design, for every read T_5 performs, we can check whether the read value is still up-to-date. As soon as we detect a stale read, we can abort the simulation of the transaction proposal. Additionally, we directly notify the corresponding client about the abort, such that it can resubmit the proposal without delay.

Let us discuss in the following, how exactly our fine-grained concurrency control mechanism works and how we realize it in Fabric++. In the context of modern database systems, advanced concurrency control mechanisms are well established [16, 19, 21, 23, 24, 27]. Instead of locking the entire store, these techniques typically perform a fine-grained locking on the record level or even at the level of individual cells/values. As there is conceptually no difference between the store of a database system and the store used within the Fabric peers, similar techniques can be applied here.

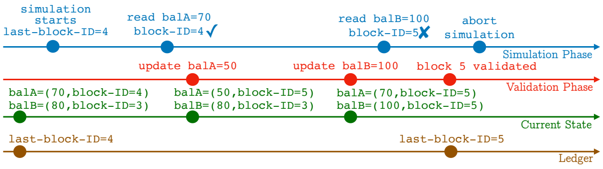

simulation starts last-block-ID=4
read balA=70 block-ID=4 ✓
read balB=100 block-ID=5 ✗
abort simulation
Simulation Phase
update balA=50
update balB=100
block 5 validated
Validation Phase
balA=(70,block-ID=4) balB=(80,block-ID=3)
balA=(50,block-ID=5) balB=(80,block-ID=3)
balA=(70,block-ID=5) balB=(100,block-ID=5)
Current State
last-block-ID=4
last-block-ID=5
Ledger

Figure 6: Parallelization with early abort using our fine-grained concurrency control.

As discussed in Section 2, Fabric implements its current state in form of a key-value store, which maps each individual key to a pair of value and version-number. The version-number is actually composed of the ID of the transaction, that performed the update, as well as the ID of the block that

Correction:
Actually, T6 becomes invalid in this example, not T7. Thus, the former transaction is early aborted, not the latter.

contains the transaction. In the original version of Fabric, the sole purpose of the version-numbers is to identify stale reads. In the validation phase, for every transaction we check whether the version-number of the read value still matches the one in the current state.

We can go one step further and exploit the available version-numbers to implement a lock-free concurrency control mechanism protecting the current state. To do so, in Fabric++, we first remove the read-write lock, that was unnecessarily sequentializing simulation and validation phase. The version-number, that is maintained with each value, is sufficient to ensure the same transaction isolation semantics as the vanilla version. As no lock is acquired anymore, we need a mechanism to ensure that updates performed by the validation phase are not seen by simulation phases running in parallel. To achieve this behavior, during simulation, we have to inspect the version-number of every read value and test whether it is still up-to-date.

Figure 6 visualizes this concept using a concrete example. At the start of the simulation phase, we first identify the `block-ID` of the last block that made it into the ledger. Let us refer to this `block-ID` as the `last-block-ID`. In our example, `last-block-ID = 4`. During the simulation of a smart contract bound to a transaction proposal T_{exec}, no read must encounter a version-number containing a `block-ID` higher than the `last-block-ID`. If it does see a higher `block-ID` it means that during the simulation phase, a validated transaction T_{valid} in the validation phase modified a value in the read set of T_{exec} and thus, the read set is outdated.

In our example, the read of $balA = 70$ in the simulation phase happens *before* the update of $balA$ to 50 in the validation phase. This is reflected by the version-number of $balA$, namely `block-ID = 4`. Therefore, this read is up-to-date and the simulation continues. In contrast to that, the read of $balB$ happens *after* the update of $balB$ to 100 in the validation phase. This is reflected by the version-number of $balB$, namely `block-ID = 5`. As 5 is higher than the `last-block-ID = 4`, we can directly classify T_{exec} as invalid, as the transaction will not have a chance to pass the validation phase later on. Please note that the overall correctness of our lock-free mechanism is ensured by the atomic updates of the version-numbers.

5.2.2 Early Abort in the Ordering Phase. In addition to the early abort in the simulation phase, as explained in Section 5.2.1, we can transition a similar concept also to the ordering phase. As Fabric performs commits at the granularity of whole blocks, two transactions within the same block, that read the same key, must read the same version of that key. For example, let us consider two transactions T_6 and T_7, where T_6 is ordered before T_7 within the same block ($T_6 \Rightarrow T_7$). If T_6 read version v_1 of a key k and T_7 read version v_2 of k in their respective simulations, then T_7 is invalid. Such a version mismatch can happen, if between the simulations of T_6 and T_7 a change to the value of k was committed by a valid transaction from a previous block. Therefore, as soon as we detect a version mismatch between transactions within the same block, we can early abort the latter transaction. Again, this strategy assures that only those transactions end up in a block, that have a realistic chance of commit.

6 EXPERIMENTAL EVALUATION

In the previous section, we have extended and modified core components of Fabric in several ways, turning it into Fabric++. It is now time to evaluate the modifications in terms of effectiveness. Primarily, we are interested in the throughput of valid/successful and invalid/failed transactions, that make it through the system. Secondarily, we are interested on the influence of certain system configurations and the workload characteristics on the system.

6.1 Setup

Before starting with the actual experiments, let us discuss the setup. Our cluster consists of six identical servers, that are located within the same rack and connected via gigabit-ethernet. Four machines serve as peers, one machine runs the ordering service, and one machine serves as the client, which fires transaction proposals. Each server consists of two quad-core Intel Xeon CPU E5-2407 (SandyBridge architecture) running at 2.2 GHz with 32KB of L1 cache, 256KB of L2 cache, and 10MB of a shared L3 cache. 24GB of DDR3 ram are attached to each of the two NUMA regions. The operating system used is a 64-bit Arch Linux with kernel version 4.17. Fabric is set up to use LevelDB as the current state database.

6.2 Benchmark Framework and Workload

In the database community, there exist numerous established benchmarking suites and workloads that can be used to test and to compare systems, such as TPC-C [3], TPC-H [4], or YCSB [5]. Unfortunately, since blockchains are still a relatively young field, there exist only very few benchmarks with standardized workloads.

6.2.1 Framework. First, we have to identify a framework that can be used to run a workload against Fabric. There are essentially three options available right now: Caliper [6], Gauge [7], and BlockBench [12]. Caliper feels like a natural candidate, as it originates from the Hyperledger project just like Fabric. While it is compatible with Fabric 1.2, it suffers from certain limitations: it supports only a single channel, it supports only one transaction type per run, it fires transactions non-uniformly with respect to time, and it is prone to failing with missed events at high firing rates [8]. As a consequence, Gauge was forked from Caliper, which addresses some of these problems. Unfortunately, it lacks compatibility with Fabric 1.2. The same incompatibility holds for BlockBench.

As none of the available frameworks is fully satisfying and since the framework is just a tool for running experiments, we decided to build our own benchmarking framework. It allows us to fire transaction proposals uniformly at a specified rate from multiple clients in multiple channels and reports the throughput of successful and aborted transactions per second. We use our framework for all main experiments in the upcoming evaluation. Still, we include an experimental run on Caliper with a compatible workload in Section 6.7 to ease for other groups the comparison with our work.

6.2.2 Workload. In the following experiments, we use two different types of workloads.

The first one is the Smallbank [9] workload, which is perfectly suited to test a blockchain system, as it simulates a typical asset transfer scenario. Initially, it creates for a certain number of users a checking account and a savings account each and initializes them with random balances. The workload consists of six transactions, where five of them update the account balances in certain ways: `TransactSavings` and `DepositChecking` increase the savings account and the checking account by a certain amount respectively. `SendPayment` transfers money between two checking accounts. `WriteCheck` decreases a checking account, while `Amalgamate` transfers all funds from a savings account to a checking account. Additionally, there is a read-only transaction `Query`, which reads both the checking as well as the savings account of a user. During a single run, we repeatedly fire these six transactions in a random fashion, where we uniformly pick one of the five modifying transactions with a certain probability P_w, and the reading transaction with a probability $1 - P_w$. For each picked transaction, we determine the accounts to access by following a Zipfian distribution, which we can configure in terms of skewness by setting the s-value. Note that an s-value of 0 corresponds to a uniform distribution.

Our second workload consists solely of a single, highly configurable transaction, which performs a certain number of read and write accesses on a set of account balances. Initially, we create a certain number of accounts (N), each initialized with a random integer. Our transaction performs a certain number of reads and writes (RW) on a subset of these accounts. Among the accounts, there exist a certain number of hot accounts (HSS), that are picked for a read respectively write access with a higher probability. This probability for picking a hot account for reading (HR) respectively for writing (HW) can also be configured.

In a single run, we fire a constant stream of transactions for a certain amount of time at a certain firing rate. In the following experiments, we fix the experimental and system configuration to the parameters as shown in Table 5. We identified these parameter values empirically with the goal to find a configuration, that sustains the system without overloading it.

Table 5: Experiment and system configuration.

Experiment Parameters	Values
Fired transaction proposals per second per client	512
Duration in which transaction proposals are fired	90 sec
Number of channels	1
Number of clients per channel	4
System Parameters	**Values**
Maximum time to form a block	1 sec
Maximum number of keys accessed per block	16384
Maximum size per block	2MB
Maximum number of transactions per block (BS)	1024 (see Section 6.3)

6.3 The Impact of the Blocksize

We start our evaluation by investigating the effect of the blocksize on Fabric and Fabric++. By default, Fabric's sample network limits the blocksize to only up to 10 transactions. In the following experiment, we vary the blocksize from 16 transactions to 2048 transactions in logarithmic steps and observe the impact on the number of successful transactions. As workload, we test Smallbank as defined above with 100,000 users under a write heavy workload with $P_w = 95\%$, and a uniform distribution with s-value = 0. Figure 7 shows the average number of successful transactions per second over the entire run of 90 seconds.

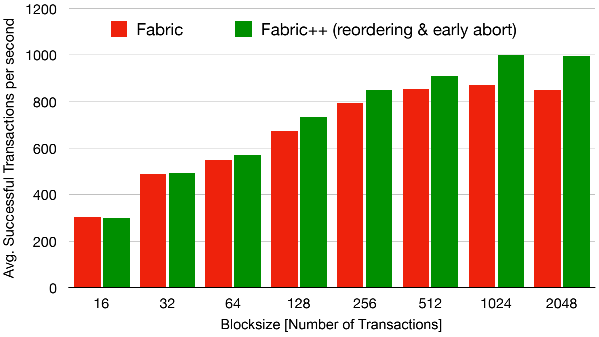


Figure 7: Effect of the blocksize on the average number of successful transactions under Fabric and Fabric++.

As we can see, increasing the blocksize also increases the throughput of successful transactions for both Fabric and Fabric++. This is due to the fact, that the usage of larger blocks causes less network communication. Obviously, Fabric's default setting of 10 transactions per block is clearly too small and severely limits the throughput. We can also observe, that Fabric++ gains more over Fabric with an increase in the blocksize. This already gives us a first impression of the effectiveness of our reordering mechanism, which benefits from larger blocks. As we aim for a high overall throughput, for the remaining experiments, we use a blocksize of 1024 transactions.

6.4 Transactional Throughput

Let us now test Fabric and Fabric++ under probably the most important criterium for a transaction processing system, namely the throughput of successful transactions.

6.4.1 Throughput under Smallbank. First, as workload, we use Smallbank as defined in Section 6.2.2 and configure it as shown in Table 6. Again, we initialize 100,000 users, each equipped with a checking account and a savings account. However, this time we vary the probability of picking a modifying transaction over the reading transaction in three steps: We test $P_w = 95\%$ (write-heavy), $P_w = 50\%$ (balanced), and $P_w = 5\%$ (read-heavy). Further, we vary the skewness of the Zipf distribution, that is used to select the accounts: We go from an s-value = 0.0 (uniform) to an s-value = 2.0 (highly skewed) in steps of 0.2. Table 6 summarizes the configuration again.

Table 6: Smallbank workload configuration.

Workload Parameters	Values
Number of users (two accounts per user)	100,000
Probability for picking a modifying transactions (P_w)	95%, 50%, 5%
s-value of Zipf distribution	$0.0 - 2.0$ in steps of 0.2

In Figure 8, we can see the results. We show one plot each for the read-heavy, balanced, and write-heavy workload. Within each plot, on the x-axis, we vary the s-value as described and report on the y-axis the average number of successful transactions per second over the run. Overall, we can see that Fabric++ shows a higher throughput of successful transactions for all tested runs. We can observe, that for little to no skew (up to an s-value of 0.6), the throughput of both Fabric and Fabric++ is relatively high, as the number of potential conflicts between transactions is small by default. Still, we see that with around 1000 successful transactions per second, the throughput of Fabric++ is a bit higher than for Fabric with around 900 transactions, which is mainly caused by cleaning the pipeline from transactions, that have no chance to commit. For higher skew (s-value ≥ 1.0), we can see that Fabric++ drastically improves over Fabric, especially for the balanced and write-heavy workloads. For an s-value of 1.0, we observe improvement factors between 1.15x and 1.37x, while for an s-value of 2.0, Fabric++ shows an improvement between 2.68x and 12.61x. High skew in the access leads to a large number of potential conflicts, which can be resolved by our optimizations. For such a high contention, our optimizations make the difference between a system, that is essentially jammed (30 successful transactions per second for Fabric under $P_w = 95\%$, s-value = 2.0) and a system, that fluently processes transactions (370 successful transactions per second for Fabric++ under $P_w = 95\%$, s-value = 2.0).

6.4.2 Throughput under custom workload. Let us now investigate the throughput under our custom workload. Again, we use the experimental configuration as well as the system configuration of Table 5. We configure our workload such that we use $N = 10{,}000$ accounts. We test both $RW = 4$ and $RW = 8$ read and write accesses per transaction. The probability of picking a hot account for reading is varied between $HR = 10\%$, $HR = 20\%$, and $HR = 30\%$. The probability of picking a hot account for a writing access is varied from $HW = 5\%$ to $HW = 10\%$. Additionally, we vary the number of hot accounts from $HSS = 1\%$ over $HSS = 2\%$ to $HSS = 4\%$ from the total number of accounts. In total, we test 36 configurations, which are summarized in Table 7.

Table 7: Custom workload configuration.

Workload Parameters	Values
Number of account balances (N)	10,000
Number of read & written balances per transaction (RW)	4, 8
Probability for picking a hot account for reading (HR)	10%, 20%, 40%
Probability for picking a hot account for writing (HW)	5%, 10%
Number of hot account balances (HSS)	1%, 2%, 4%

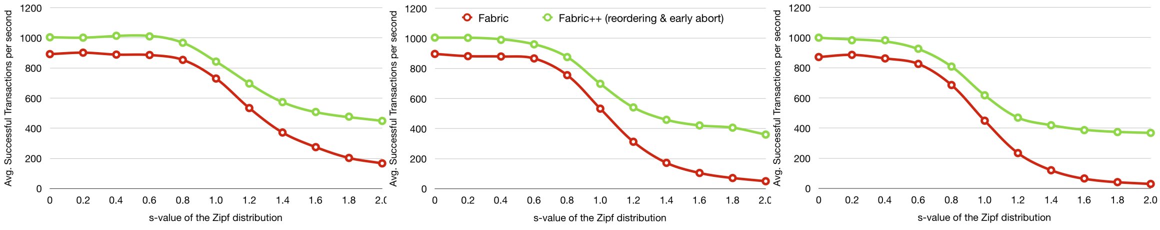
Figure 9 shows the results. We can see that Fabric++ significantly increases the throughput of successful transactions over Fabric for all tested configurations. The largest improvement of Fabric++ over Fabric in terms of successful transactions we observe is around factor 3x for the configuration BS=1024, RW=8, HR=40%, HW=10%, HSS=1%.

6.4.3 Observations. We observe a significant decrease in the throughput of the successful transactions with the increase in the hotness of the transactions in both workloads. Each block b_i roughly updates every hot key. This forces most of the transactions in the next block b_{i+1} to abort because of read-write conflicts. In comparison to Fabric, which suffers heavily from this scenario, Fabric++ reorders the transactions within the block to remove the within-block conflicts to improve the overall throughput of successful transactions.

Fabric++ is also capable of improving the throughput of successful transactions significantly under workloads (Smallbank) which read and modify the same set of keys. Fabric++ prefers to select more transactions that access fewer keys rather than selecting fewer transactions with large number of accesses to improve the end-to-end throughput of successful transactions (as shown in Figure 8). For the workload that potentially has a non-overlapping read and write sets, Fabric++ is able to re-organize the transaction block to minimize the number of unnecessary aborts (as shown in Figure 9).

6.5 Optimization Breakdown

In Section 6.4, we measured the throughput of Fabric++ with both optimizations activated. Let us now see at a sample configuration, how much the individual optimizations of



(a) P_w = 5% (read-heavy)

(b) P_w = 50% (balanced)

(c) P_w = 95% (write-heavy)

Figure 8: Average number of successful transactions per second of Fabric and Fabric++ under the Smallbank workload, as defined in Table 6.

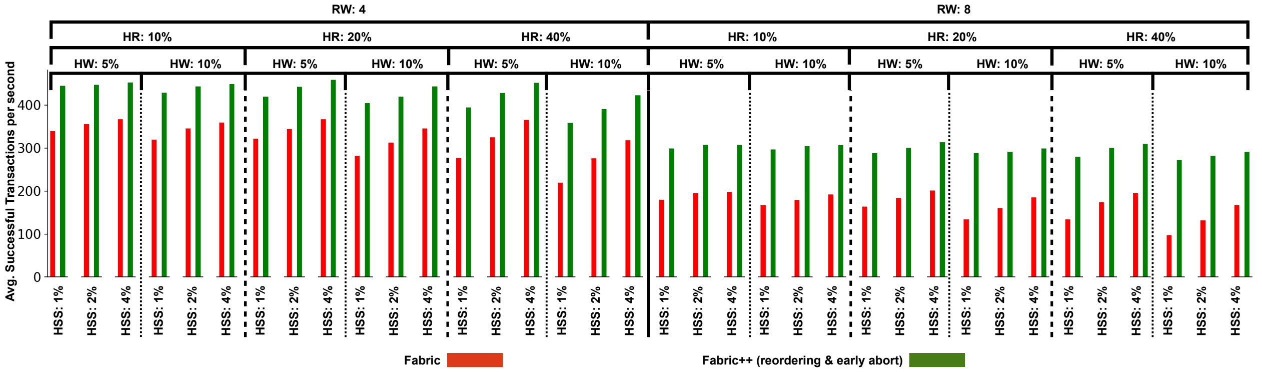


Figure 9: Average number of successful transactions per second of Fabric and Fabric++ under 36 different configurations, as defined in Table 7. We vary the number of read & written balances per transaction (RW), the probability for picking a hot account for reading (HR) and writing (HW), and the number of hot account balances (HSS).

reordering and early abort contribute to the improvement. Figure 10 shows the improvement breakdown for the configuration BS=1024, RW=8, HR=40%, HW=10%, HSS=1% in comparison to standard Fabric. While Fabric achieves only a throughput of around 100 successful transactions per second, activating one of our two optimization techniques alone improves this to around 150 transactions per second. In comparison to that, activating both techniques at the same time results in the highest throughput of successful transactions with around 220 transactions per second. This shows nicely how both techniques work together: Transactions, that are already early aborted in the simulation phase do not end up in a block in the ordering phase. As a consequence, only transactions, that have a realistic chance of being successful, are considered in the reordering process.

6.6 Scaling Channels and Clients

In all of our previous experiments we used four clients to fire transactions on a single channel. We now vary the number of channels, and the number of clients to see the effect on the throughput. We use the configuration BS=1024, RW=8, HR=40%, HW=10%, HSS=1% to evaluate the average throughput of successful transactions for Fabric and Fabric++.

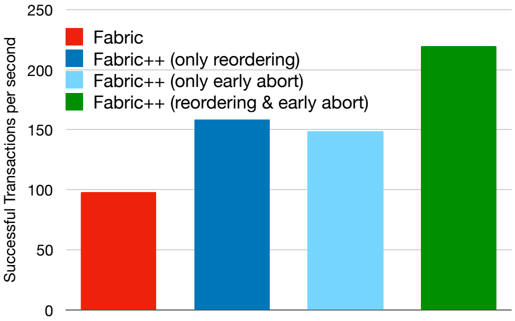
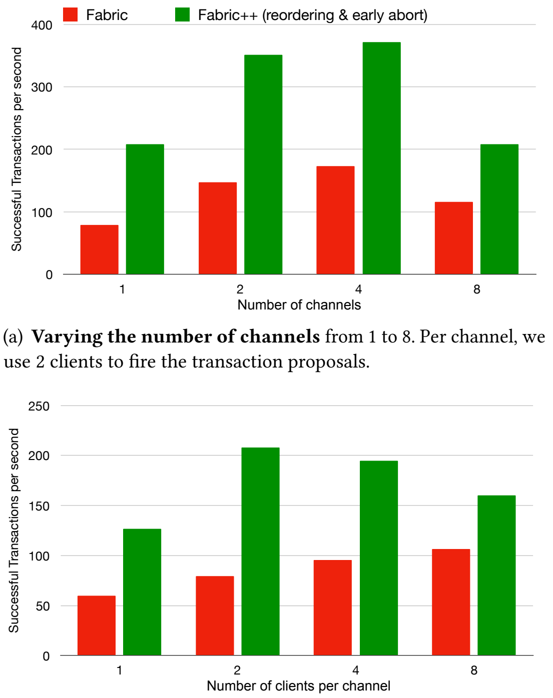


Figure 10: Breakdown of the individual impact of our optimizations on the throughput of successful transactions for the configuration BS=1024, RW=8, HR=40%, HW=10%, HSS=1%.

First, we vary the number of channels in Figure 11(a) from 1 to 8. Per channel, we use 2 clients to fire transaction proposals. We can see that when going from 1 channel to 4 channels, the throughput of both Fabric and Fabric++ significantly increases. Obviously, the additional mechanisms of Fabric++ do not harm the scaling with the number of channels. Only when using 8 channels, the throughput decreases again for



(a) **Varying the number of channels** from 1 to 8. Per channel, we use 2 clients to fire the transaction proposals.

(b) **Varying the number of clients per channel** from 1 to 8. All clients fire their transaction proposals in a single channel.

Figure 11: The impact of the number of channels as well as the number of clients per channel on the throughput of successful transactions for the configuration BS=1024, RW=8, HR=40%, HW=10%, HSS=1%.

both Fabric and Fabric++. This is simply the case because individual channels start competing for resources. This also increases the number of failed transactions: Scaling from 1 to 8 channels increases the number of failed transactions from 213 TPS to 837 TPS for Fabric and from 81 TPS to 704 TPS for Fabric++. Due to the competition for resources, individual simulations phase take longer and increase the chance of working on stale data.

After varying the number of channels, let us now vary the number of clients per channel in Figure 11(b). We test 1, 2, 4, and 8 clients, where all clients fire their transaction proposals into a single channel. Here, the picture is a slightly different to the behavior when scaling channels. The throughput of Fabric increases very gently with the number of clients, and we see an improvement from around 60 to 105 successful transactions per seconds when going from 1 to 8 clients. For Fabric++, we see the highest throughput with around 205 successful transactions per second already for 2 clients. For 8 clients, the throughput drops by around factor 2 to the throughput of Fabric, clearly showing that the firing clients also compete for resources. This is also visible in an increase in failed transactions when going from 1 to 8 clients per channel, which increase from 86 TPS to 928 TPS for Fabric and from 20 TPS to 841 TPS for Fabric++.

6.7 Hyperledger Caliper

For completeness, let us finally see how Fabric and Fabric++ perform under a run of the Hyperledger Caliper benchmarking framework. As said, Caliper severely struggles with high transaction firing rates, so we cannot use the configuration of Table 5 as before. Instead, we fire at a lower rate of 150 transactions per second per client, resulting in 600 transactions per second in total. As a consequence of this low firing rate, we also tune down the block size to 512 transactions. We test our custom workload with N = 10000, RW = 4, HR = 40%, HW = 10%, HSS = 1%. Table 8 shows the results.

Table 8: Latency and Throughput as measured by Caliper for Fabric and Fabric++.

Metric	Fabric	Fabric++
Max. Latency [seconds]	1.44	1.14
Min. Latency [seconds]	0.26	0.12
Avg. Latency [seconds]	0.47	0.28
Avg. Successful Transactions per second	188	299

Interestingly, Caliper also produces latency numbers additionally to the measured throughput of successful transactions. We can see that the average latency of Fabric++ is almost half the latency of the vanilla Fabric. As less virtually invalid transactions trash the pipeline in Fabric++, valid transactions can commit earlier. The run of Caliper also confirms our findings on the throughput: Fabric++ significantly increases the number of successful transactions per second.

7 CONCLUSION

In this work, we identified strong similarities of the transaction pipeline of contemporary blockchain systems at the case of Hyperledger Fabric and distributed database systems in general. We analyzed these similarities in detail and exploited them to transition mature techniques from the context of database systems to Fabric, namely transaction reordering to remove serialization conflicts as well as early abort of transactions, that have no chance to commit. In an extended experimental evaluation, where we tested Fabric++ and the vanilla version under the Smallbank benchmark as well as under a custom workload, we show that Fabric++ is able to significantly outperform Fabric by up to a factor of 12x for the number of successful transactions per second. Further, we are able to almost half the transaction latency, while keeping the scaling capabilities of the system intact.

8 ACKNOWLEDGEMENT

This work was funded by the German Research Foundation (DFG) via the collaborative research center “Methods and Tools for Understanding and Controlling Privacy” (SFB 1223).

REFERENCES

[1] 2019. https://bitcoin.org/bitcoin.pdf

[2] 2019. https://github.com/ethereum/wiki/wiki/White-Paper

[3] 2019. http://www.tpc.org/tpcc/

[4] 2019. http://www.tpc.org/tpch/

[5] 2019. https://github.com/brianfrankcooper/YCSB

[6] 2019. https://github.com/hyperledger/caliper

[7] 2019. https://github.com/persistentsystems/gauge

[8] 2019. https://github.com/persistentsystems/gauge/blob/master/docs/caliper-changes.md

[9] 2019. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/

[10] Elli Androulaki, Artem Barger, Vita Bortnikov, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *EuroSys 2018, Porto, Portugal, April 23-26*. 30:1–30:15. https://doi.org/10.1145/3190508.3190538

[11] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving Optimistic Concurrency Control Through Transaction Batching and Operation Reordering. *PVLDB* 12, 2 (2018), 169–182. http://www.vldb.org/pvldb/vol12/p169-ding.pdf

[12] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. 2018. Untangling Blockchain: A Data Processing View of Blockchain Systems. *IEEE Trans. Knowl. Data Eng.* 30, 7 (2018), 1366–1385. https://doi.org/10.1109/TKDE.2017.2781227

[13] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *PVLDB* 10, 5 (2017), 613–624. https://doi.org/10.14778/3055540.3055553

[14] Zhengyu He and Bo Hong. 2009. Impact of early abort mechanisms on lock-based software transactional memory. In *16th International Conference on High Performance Computing, HiPC 2009, December 16-19, 2009, Kochi, India, Proceedings*. 225–234. https://doi.org/10.1109/HIPC.2009.5433207

[15] Donald B. Johnson. 1975. Finding All the Elementary Circuits of a Directed Graph. *SIAM J. Comput.* 4, 1 (1975), 77–84. https://doi.org/10.1137/0204007

[16] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB* 5, 4 (2011), 298–309.

[17] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 21–35. https://doi.org/10.1145/3035918.3064015

[18] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael Watzke. 2010. Transaction reordering. *Data Knowl. Eng.* 69, 1 (2010), 29–49. https://doi.org/10.1016/j.datak.2009.08.007

[19] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD 2015, Melbourne, Victoria, Australia, May 31 - June 4*. 677–689. https://doi.org/10.1145/2723372.2749436

[20] Thamir M. Qadah and Mohammad Sadoghi. 2018. QueCC: A Queue-oriented, Control-free Concurrency Architecture. In *Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10-14, 2018*. 13–25. https://doi.org/10.1145/3274808.3274810

[21] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual Snapshotting. In *SIGMOD 2018, Houston, TX, USA, June 10-15, 2018*. 245–258. https://doi.org/10.1145/3183713.3196904

[22] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. https://doi.org/10.1137/0201010

[23] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. 2017. Efficiently making (almost) any concurrency control mechanism serializable. *The VLDB Journal* 26, 4 (01 Aug 2017), 537–562. https://doi.org/10.1007/s00778-017-0463-8

[24] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *PVLDB* 10, 2 (2016), 49–60. http://www.vldb.org/pvldb/vol10/p49-wang.pdf

[25] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *PVLDB* 10, 7 (2017), 781–792. https://doi.org/10.14778/3067421.3067427

[26] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. 2016. Exploiting Single-Threaded Model in Multi-Core In-Memory Systems. *IEEE Trans. Knowl. Data Eng.* 28, 10 (2016), 2635–2650. https://doi.org/10.1109/TKDE.2016.2578319

[27] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *PVLDB* 8, 3 (2014), 209–220.

[28] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-Memory Databases. *PVLDB* 9, 6 (2016), 504–515. https://doi.org/10.14778/2904121.2904126

[29] Bo Zhang, Binoy Ravindran, and Roberto Palmieri. 2015. Reducing Aborts in Distributed Transactional Systems through Dependency Detection. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*. 13:1–13:10. https://doi.org/10.1145/2684464.2684475

[30] Ningnan Zhou, Xuan Zhou, Xiao Zhang, et al. 2017. Reordering Transaction Execution to Boost High-Frequency Trading Applications. *Data Science and Engineering* 2, 4 (2017), 301–315. https://doi.org/10.1007/s41019-017-0054-0

A HYPERLEDGER FABRIC: A RUNNING EXAMPLE

In the following, the interested reader finds a complete running example of Fabric's workflow in Figure 12 (simulation phase), Figure 13 (ordering phase), and Figure 14 (validation and commit phase), where two organizations A and B want to transfer money between each other.

Each organization contributes two peers to the network. The balances of the organizations are stored by two variables *BalA* and *BalB*, where *BalA* stores the value 100 in its current version v_3 and *BalB* stores 50 in version v_2. We can also see that the ledger already contains six transactions T_1 to T_6, where the four transactions T_1, T_2, T_4, and T_6 were valid ones and lead to the current state. The transactions T_3 and T_5 were invalid transactions. They are still stored in the ledger, although they did not pass the validation phase.

A.1 Simulation Phase

Transaction processing starts with the *simulation* phase in Figure 12. In **step ❶**, a client proposes a *transaction proposal* (or short *proposal*) to the system. In our example, the proposal intends to transfer the amount of 30 from *BalA* to *BalB*. The two involved operations *BalA*-=30 and *BalB*+=30 are expressed in a smart contract[2], an arbitrary program, that is bound to the proposal. Additionally to the smart contract, an endorsement policy must be specified. It determines which and/or how many peers have to endorse the proposal. In our example of money transfer between two organizations, a reasonable endorsement policy is to request endorsement from one peer of each organization — like two lawyers, preserving and defending the individual rights of their clients.

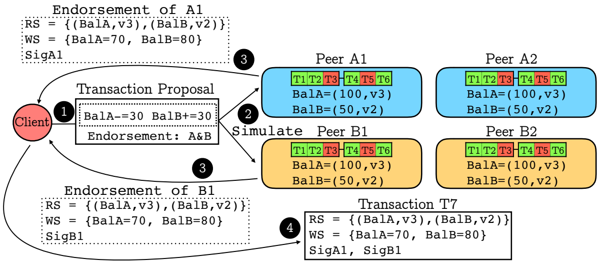


Figure 12: simulation phase.

Therefore, in **step ❷**, the proposal is sent to the two endorsement peers $A1$ and $B1$ according to the policy. These two peers now individually simulate the smart contract (*BalA*-=30, *BalB*+=30), that is bound to the proposal, against their local current state. Note that, as the name suggests, the simulation of the smart contract against the current state does *not change* the current state in any way. Instead, each endorsement peer builds an auxiliary read set RS and a write set WS during the simulation to keep track of all accesses that happen. In our case of money transfer over the amount of 30, the smart contract first reads the two current balances *BalA* and *BalB* along with their current version-numbers. Second, the smart contract updates the two balances according to the transferred amount, resulting in the new balances $BalA = 70$ and $BalB = 80$. Overall, this builds the following read and write set:

$$RS = \{(BalA, v_3), (BalB, v_2)\} \quad WS = \{BalA = 70, BalB = 80\}$$

In this sense, the simulation of the smart contract is actually only a monitoring of the execution effects. The reason for performing only a simulation is that in this phase, we cannot be sure yet whether this transaction will be allowed to commit eventually – this check will be performed later in the validation phase.

After the simulation of the smart contract on all endorsement peers, in **step ❸**, the endorsement peers return their individually computed read and write sets to the client, that sent the transaction proposal. Additionally, they return a signature of their simulation, that will be relevant in the validation phase in Section A.3. If all read sets and write sets match[3], in **step ❹**, the actual *transaction* (called T_7 in the following) is formed from the results of the endorsement. This transaction T_7 now contains the effects of the execution in form of the read and write set as well as all signatures and can be passed on to the ordering service.

A.2 Ordering Phase

As mentioned, the central component of the *ordering* phase is the ordering service, that we visualize in Figure 13. It receives all transactions, that made it through the simulation phase. Consequently, it receives in **step ❺** our transaction T_7, that we followed through the simulation phase in Section A.1. In **step ❻**, we assume that it also receives two other transactions T_8 and T_9, that were endorsed in parallel to T_7.

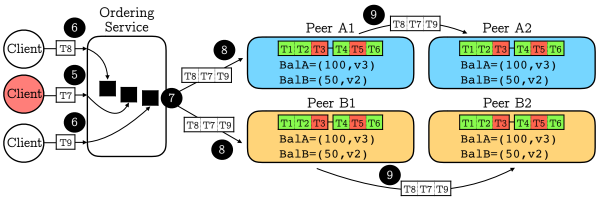


Figure 13: Ordering Phase.

The ordering service now has the sole purpose of establishing a global order among the transactions. It treats the transactions in a black box fashion and does not inspect the transaction semantics, such as the read and write set, in any way. By default, it essentially arranges the transactions in the order in which they arrive, resulting in what we call for the rest of the paper the *arrival order.* In **step ❼**, the ordering service now outputs the ordered stream of transactions in form of *blocks*, containing a certain number of transactions. Outputting whole blocks instead of individual transactions reduces the pressure on the network, as less communication overhead is produced.

Finally, the generated block is distributed to *all* four peers of the network to start the validation phase. Note that there is no guarantee that all peers receive a block at the same time, as the distribution happens partially from ordering service to peers directly as shown in **step ❽** and partially between the peers using a gossip protocol as shown in **step ❾**. However, the service assures that all peers receive the blocks in the same order.

[2] Smart contracts are typically called *chaincodes* in Fabric. However, as they do not conceptually differ from smart contracts in blockchain systems such as Ethereum, we stick to this term throughout the paper.

[3] They might not match due to non-determinism in the smart contract or due to malicious behavior of the endorsement peer(s).

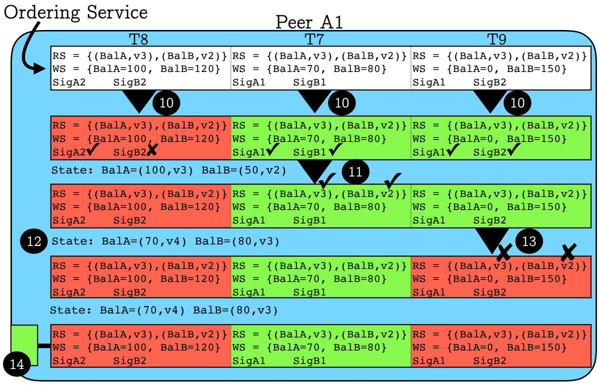

Ordering Service
Peer A1
T8
T7
T9
RS = {(BalA,v3),(BalB,v2)}
WS = {BalA=100, BalB=120}
SigA2 SigB2
RS = {(BalA,v3),(BalB,v2)}
WS = {BalA=70, BalB=80}
SigA1 SigB1
RS = {(BalA,v3),(BalB,v2)}
WS = {BalA=0, BalB=150}
SigA1 SigB2
10
10
10
SigA2✓ SigB2✗
SigA1✓ SigB1✓
SigA1✓ SigB2✓
State: BalA=(100,v3) BalB=(50,v2)
11
12
State: BalA=(70,v4) BalB=(80,v3)
13
State: BalA=(70,v4) BalB=(80,v3)
14

Figure 14: Validation and Commit Phase.

A.3 Validation and Commit Phase[4]

When a block arrives at a peer, the *validation* phase starts, visualized in Figure 14 for peer *A*1. The three remaining peers execute the same validation process. Overall, the validation phase has two purposes.

A.3.1 Endorsement Policy Evaluation. The first purpose is to validate the transactions in the block with respect to the *endorsement policy*. For example, it is possible that a malicious transaction was generated by a malicious client and a malicious peer in conspiracy to take advantage of the money transfer. Let us assume that transaction T_8 is such a malicious transaction and that the malicious client, which proposed T_8, works together with peer *A*2, which is also malicious. Instead of using the legit write set $WS_{B2} = \{BalA = 30, BalB = 120\}$ from B2, the client creates a proposal with the write set $WS_{A2} = \{BalA = 100, BalB = 120\}$, that it received from its collaborator A2.

How is this transaction T_8 now detected in the validation phase? The key to this lies in the signatures Sig_{A2} and Sig_{B2}, that the endorsement peers generate at the end of the simulation phase. The signature is computed over the read and write set, the executed smart contract, and the used endorsement policy. The client receives these cryptographically secure signatures and must pack them into the transaction along with the read and write set. The peers that validate the transaction recompute the signatures of all endorsement peers, that were responsible for transaction T_8 and compare the signatures with the received ones Sig_{A2} and Sig_{B2}. In our example, in **step** ❿, the peers detect that the signature of the honest peer Sig_{B2} does not match to the one they computed from the received write set and thus, would classify T_8 as invalid. T_7 and T_9, the remaining transactions in the block, are evaluated in parallel. Their signatures match the ones computed from the read and write set and therefore, these transactions are valid with respect to the endorsement policy.

A.3.2 Serializability Conflict Check. The second purpose of the validation is to analyze the transactions with respect to *serializability conflicts*, that can arise from the order of transactions. For every transaction, it must be checked whether the version-numbers of all keys in the read set match the version-numbers in the current state. Only if this is the case, a transaction operates on an up-to-date state. Considering our example, let us perform the serializability conflict check for the received block. T_8 is already marked as invalid as it did not pass the endorsement policy evaluation, so it is not checked again. T_7 passed the endorsement policy evaluation and is now tested for serialization conflicts in **step** ⓫. Its read set is $RS = \{(BalA, v_3), (BalB, v_2)\}$. The version numbers of *BalA* and *BalB* in the read set match the ones of the current state and therefore, T_7 is marked as valid. As a consequence, in **step** ⓬, the write set of T_7, namely $WS = \{BalA = 70, BalB = 80\}$ is written to the current state. This changes the current state to $BalA = (70, v_4)$ and $BalB = (80, v_3)$. Note that the version-numbers of the modified variables are incremented.

The next transaction to be checked is T_9 in **step** ⓭. Let us assume it also performs a money transfer and has the following read and write set:

$$RS = \{(BalA, v_3), (BalB, v_2)\} \quad WS = \{BalA = 0, BalB = 150\}$$

This transaction will not pass the conflict check, as it read *BalA* in version v_3 and *BalB* in version v_2, while the current state already contains *BalA* in version v_4 and *BalB* in version v_3. Therefore, it operated on outdated data and is marked as invalid. As a consequence, its write set is not applied to the current state and simply discarded.

Finally, after validating all transactions of the block, in **step** ⓮ the entire block is appended to the ledger along with the information about which transactions are valid or invalid.

B REORDERING: MICRO-BENCHMARKS

To analyze the effectiveness of our reordering mechanism, we also evaluate it in a stand-alone micro-benchmark in isolation of Fabric. For a given sequence of input transactions we compute the number of valid transactions for this particular sequence (called "arrival order" in the following plots) as well as for the sequence that is generated by our reordering mechanism (called "reordered" in the following plots). Additionally, we measure the time to compute the reordered schedule. In Figure 15, we test a workload pattern with varying number of conflicts. Additionally, we evaluate the effect of varying the length of the cycles (Figure 16) and see how well our reordering mechanism performs in comparison to the naive arrival order.

[4] Logically, these are two separate phases. However, as Fabric interleaves these phases, we describe them together.

B.1 Micro-Benchmark 1: Interleave reads and writes to vary the number of conflicts

The first input sequence we test consists of two equal sized sub sequences, where one subsequence contains only transactions that perform writes (colored in red) and the other sequence only transactions that read (colored in blue). Each transaction performs only one operation (either read or write). Neither two writes nor two reads happen to the same key. For the example of $n = 6$ transactions, we start with the following sequence S_1:

$$S_1 = T[w(k_1)], T[w(k_2)], T[w(k_3)], T[r(k_1)], T[r(k_2)], T[r(k_3)]$$

To generate S_i, we move the last transaction of S_{i-1} to the front, leading to the following sequences S_2, S_3, and S_4.

$$S_2 = T[r(k_3)], T[w(k_1)], T[w(k_2)], T[w(k_3)], T[r(k_1)], T[r(k_2)]$$

$$S_3 = T[r(k_2)], T[r(k_3)], T[w(k_1)], T[w(k_2)], T[w(k_3)], T[r(k_1)]$$

$$S_4 = T[r(k_1)], T[r(k_2)], T[r(k_3)], T[w(k_1)], T[w(k_2)], T[w(k_3)]$$

The more writing transactions happen before the corresponding reading transactions, the more conflicts happen. We want to find out whether our reordering mechanism can solve this problem.

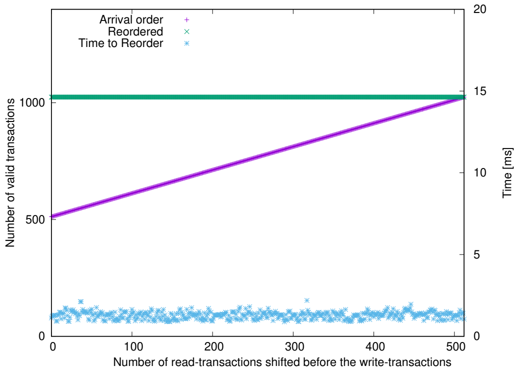


Figure 15: Workload 1: Varying the number of conflicts within the transactions.

Figure 15 shows the results for $n = 1024$ transactions. As we can see, our reordering mechanism is able to reorder the transactions for every input sequence in a way such that all transactions are valid. In contrast to that, the arrival order suffers under a lot of invalid reading transactions, if writing transactions happen before. We can also see that our reordering mechanism is computationally cheap: it takes only around 1 to 2 ms to rearrange the transactions on a Macbook Pro with Intel Core i7 running at 3.1 GHz.

B.2 Micro-Benchmark 2: Vary the length of cycles

In the following experiment, we want to analyze the impact of cycles on the arrival order and on our reordering mechanism. To do so, we again form a sequence of n transactions, that contains n/t cycles of size t transactions of the form

$$T[r(k_0), w(k_0)], T[r(k_0), w(k_1)], T[r(k_1), w(k_2)], T[r(k_2), w(k_0)]$$

Again, we want to identify how many transactions are valid under the arrival order and when using our reordering mechanism. Figure 16 shows the results for 1024 transactions. For the arrival order, only half of transactions are valid, no matter of the cycle length. This is because aborting every second transaction breaks the cycles. In comparison to that, our reordering mechanism is able to achieve a high number of valid transactions, if the cycles are sufficiently long respectively, there are not too many cycles to cancel. Of course, our algorithm becomes more expensive with the length of the cycles to break. However, since extremely long cycles are very unlikely to occur in reality, the runtime of our mechanism will in general remain low in the ordering phase, as we see our evaluation presented in Section 6.

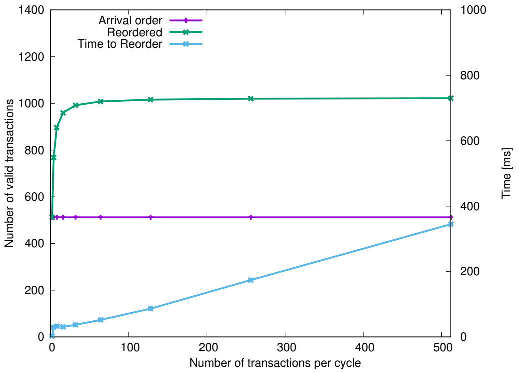


Figure 16: Workload 2: Varying the size of the cycles.