StreamChain: Do Blockchains Need Blocks?

Zsolt István[*]
IMDEA Software Institute
zsolt.istvan@imdea.org

Alessandro Sorniotti
IBM Research - Zurich
aso@zurich.ibm.com

Marko Vukolić
IBM Research - Zurich
mvu@zurich.ibm.com

ABSTRACT

Processing at block granularity and blockchains seem inseparable. The original role of blocks is to amortize the cost of cryptography (e.g., solving proof-of-work) and to make data transfers more efficient in a geo-distributed setting. While blocks are simple and powerful tool for amortizing these costs, today in permissioned distributed ledgers, that are often neither geo-distributed, nor require proof-of-work, the benefits of operating on blocks are overshadowed by the large latencies they introduce.

Our proposal is to switch the distributed ledger processing paradigm from block processing to *stream transaction processing* and rely on batching (i.e., block formation) only for amortizing the cost of disk accesses for commit operations. This paradigm shift enables shaving off end-to-end latencies by more than an order of magnitude and opens up new use-cases for permissioned ledgers. We demonstrate a proof-of-concept of our idea using Hyperledger Fabric, achieving end-to-end latencies of less than 10ms while maintaining relatively high throughput, namely close to 1500 tps.

CCS CONCEPTS

• **Information systems** → *Data management systems*; • **Applied computing** → *Electronic commerce*; • **Computer systems organization** → *Distributed architectures*;

ACM Reference Format:
Zsolt István, Alessandro Sorniotti, and Marko Vukolić. 2018. StreamChain: Do Blockchains Need Blocks?. In *2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL'18), December 10–14, 2018, Rennes, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3284764.3284765

1 INTRODUCTION

Permissioned distributed ledgers, such as Hyperledger Fabric [1] or Corda [3], inherit their core design from public blockchains and, as a result, their latencies are typically in the order of half a second, even when running on a local area network. In these settings, the high latency of committing operations can hinder wide-spread adoption, especially given that these systems do not require costly proof-of-work, and the fundamental problem distributed ledgers solve, namely reaching consensus among nodes, can be performed at the order of milliseconds even if we assume Byzantine faults [2, 4].

[*]Work done in IBM Research - Zurich.

SERIAL'18, December 10–14, 2018, Rennes, France

ACM ISBN 978-1-4503-6110-1/18/12…$15.00
https://doi.org/10.1145/3284764.3284765

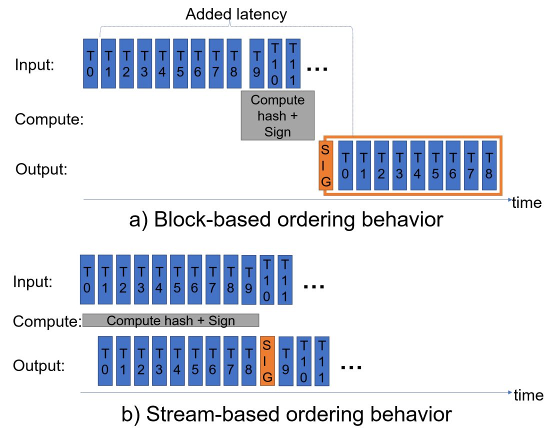

Figure 1: In StreamChain, transactions are relayed to peers as soon as they are ordered, removing batching delays.

The main reason for this high latency comes from the fact that blockchains batch transactions into blocks. For instance, in Hyperledger Fabric [1], transactions are grouped into blocks by an ordering service (see Figure 1*a*), and then validated as a block at the peers. Of course, in public distributed ledgers (blockchains) that were designed for geo-distributed use, blocks are an important tool in amortizing the cost of producing proof-of-work and the communication delays across the network. Today, however, there is a growing opportunity for distributed ledgers operated in a permissioned way, with nodes physically close to each other, e.g. in datacenters close to a stock exchange in an asset trading use cases, or close to a shipping port in a supply-chain use case.

In this work we revisit the relationship between distributed ledgers and blocks with the goal of reducing end-to-end latencies by an order of magnitude. We propose **StreamChain**, a design for distributed ledgers that handles transactions in a streaming manner both during agreement on their order and during their processing (e.g., execution and validation of a distributed application). In particular, our proof-of-concept (PoC) design, built on Hyperledger Fabric, implements streaming both at the orderer (Figure 1*b*) and the peers. In fact, StreamChain selectively uses batching (blocks) when writing to the persistent ledger state, because this allows amortizing disk access costs without introducing significant latency penalties. As a result, StreamChain can deliver low latencies and high throughput at the same time. It is important to note that these changes do not require modifications to the system and threat model, even if the underlying ordering service is Byzantine fault tolerant.

Beyond the intuitive, non-functional desirability of low latency in distributed ledgers, which directly translates to very fast consensus *finality* [11], low end-to-end latencies have an additional positive effect in *execute-order-validate* (EOV) distributed ledgers [1], such as Hyperledger Fabric. In these systems, executing transactions on stale data results in failing transactions in the validation phase (more details are in Section 2.2). Although some of these validation failures could be avoided with additional work in the execution step [7] or by using specific programming models, reducing latencies in general will reduce data staleness and benefit any application running on top.

Our StreamChain PoC on Hyperledger Fabric demonstrates the feasibility and benefits of our approach. Our StreamChain PoC delivers less than 10ms end-to-end latencies and close to 1500 transaction/s throughput, having to our knowledge the lowest latency of all existing distributed ledgers, while maintaining very competitive throughput compared to today's best performing systems. Based on our experience creating the PoC, we list several recommendations on how to implement a full-fledged version of StreamChain that could deliver even lower latencies and higher throughput.

In the following, in Section 2 we detail the motivation behind this work, in particular in the context of Hyperledger Fabric. Section 3 overviews the core idea behind StreamChain, whereas Section 4 discusses our implementation. Section 5 contains evaluation results. Section 6 discusses other techniques that can be applied to StreamChain to further reduce the latency. Finally, Section 7 concludes the paper.

2 BACKGROUND AND MOTIVATION

StreamChain's objective is to deliver low end-to-end latency for distributed ledger transactions. Low latency is often understood as a performance, non-functional, requirement; in this section, using Hyperledger Fabric architecture as motivation, we argue that it may sometimes also be a functional requirement in building permissioned blockchains.

2.1 Hyperledger Fabric

Hyperledger Fabric (or, simply, Fabric) is an open source implementation of a general-purpose permissioned blockchain system. As its design is substantially different from most other related blockchain and distributed ledger systems [1, 12], the following sections will review the design approaches that differ more radically and have influenced StreamChain.

Separation of execution and consensus. In most distributed ledger platforms, nodes play a dual role: they both take part in the consensus protocol to agree on a total order of transactions in the system, and they provide an environment for smart contracts (distributed applications, or chaincodes) to be executed and for maintaining their state. In Fabric on the other hand, two types of nodes exist: 1) ordering service nodes, which are stateless with respect to smart contracts and only implement a broadcast/deliver interface whereby incoming transactions are ordered and delivered back to the rest of the network as a sequence of blocks; 2) peer nodes, which are consumers of the ordering service, and which maintain the global state of all smart contracts and handle smart contract invocations from clients.

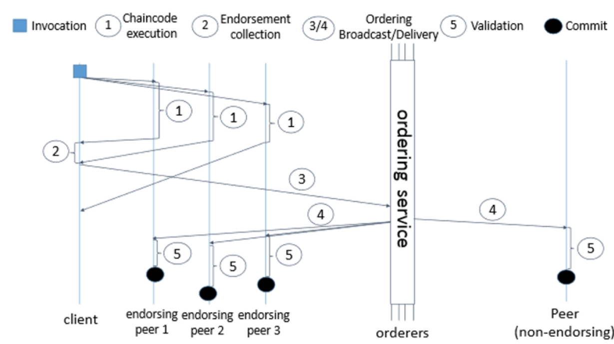

Figure 2: Fabric 5-step transaction flow. Nodes that execute and validate transactions are separate from ordering nodes.

Endorsing and committing. In Fabric, chaincodes are not executed on every node in the system, as is the case in most other platforms (Figure 2). The developer of chaincode is free to select a non-empty subset of all the peers in the network, called endorsing peers, and require that each invocation be performed on those. The set of endorsing peers is defined by an endorsement policy, defined per chaincode, which also defines a (possibly nested) threshold expression over endorsing peer identities required for validity of a transaction. At the same time, all peers maintain the state of all chaincodes (on a particular shard called a *channel*). Non-endorsing peers are called committing peers (for a given chaincode).

Execute-order-validate. Fabric adopts an execute-order-validate (EOV) programming model, whereby smart contracts are executed (or more accurately, simulated) by endorsing peers. The simulation of the smart contract does not lead to immediate updates to the world state. Rather, state updates are serialized and signed by each endorsing peer, resulting in a transaction. In the ordering step, transactions are broadcast to the ordering service, which establishes a total order among them. After ordering, transactions are inspected by all peers, which proceed to validate them by checking compliance against the endorsement policy: if a transaction contains sufficiently many signatures to satisfy the endorsement policy of the chaincode that is invoked, the corresponding state update is applied to the ledger.

The EOV model is instrumental in building a system where chaincode execution can be trivially parallelized without affecting the correctness of results even in presence of non-deterministic smart contracts. Furthermore, the EOV paradigm ensures that system throughput is not gated by the slowest smart contract, and that the system can scale up by simply adding more endorsing peers.

2.2 Concurrency

The EOV model is best suited to handle UTXO workloads. The workload takes the name from Bitcoin's Unspent Transaction Outputs. In the Bitcoin case, a transaction spends one or more outputs (or, coins) by destroying the corresponding state variables and creating new, unique ones for the recipients of that spend operation. The concept can be generalized to workloads where execution operates over variables that are written once at their creation and then modified only once at the time of their consumption, after which they are either destroyed or simply archived and never used again. UTXO workloads are particularly relevant for execute-order-validate systems since they all but eliminate concurrency issues. Consider for instance a system that handles the ownership changes of asset-backed digital tokens. If assets are modelled as one-time-use state variables that track their current owner, transfer operations can take advantage of the full parallelism afforded by the execute-order-validate model since no two legal operations can touch the same variable.

However in our experience, developers of smart contracts tend to apply programming patterns reminiscent of declarative sequential programming and relational databases, generating read/modify/write workloads on a handful of state variables. The latter are not designed for the execute-order-validate paradigm, given that at least one of any two transactions that operate on the same state variables will be marked as invalid by the concurrency control module. Although most of these applications can be coded in a different way, that is much less or not at all prone to concurrency concerns, such coding skills seem not to be widely available.

Reverting to the standard order-execute paradigm of most DLT platforms [12] would solve the issue of concurrency given the fact that smart contracts are executed sequentially. However this approach is undesirable because of the loss of performance, scalability and the additional complexity required to manage non-determinism. A viable approach is to offer chaincode developers a richer set of primitives to interact with the world state in the validation phase, compared to the simple key-value store that is available in Fabric today, e.g., introducing commutative modify operations (such as increment/decrements, additions/substractions) to complement reads and writes. This approach is potentially very powerful and is explored in the context of Fabric, yet it is non-trivial since it may lead to the proliferation of many operations, resembling to a re-introduction of a domain specific language.

It is important to point out that if a zero end-to-end latency would be achievable, concurrency issues in the EOV model would not exist. Since zero latency is not possible, our goal with minimizing latency in EOV blockchain such as Fabric, aims at reducing as much as possible the number of invalid transactions due to concurrency collisions produced by legacy coding approaches (i.e., those targeting centralized databases, not blockchains). While this does not eliminate the problem of invalid transactions, it can tremendously help with the "feel" of programming an EOV blockchain, since collisions on data objects in the millisecond windows are arguably going to occur much less frequently then those in time windows at the orders of a second or more. This is precisely the functional motivation behind reducing latency with StreamChain.

3 STREAMCHAIN

3.1 Idea Overview

Blocks in distributed ledgers provide a means of batching to amortize the cost of solving a puzzle for a proof-of-work system and to amortize the cost of performing cryptographic operations in general. Additionally, handling large blocks will make disk-based operations cheaper. In the case of permissioned ledgers, the cryptographic operations (such as signature checks on transactions) are parallelizable, and the only expensive operation is disk writes. Therefore we argue to remove blocks and process transactions as they arrive, both between ordering nodes and inside the peers. The only use of batching should be to amortize the cost of performing persistent operations to the ledger.

In practice, this means that instead of waiting for a complete block to accumulate, the orderer sends transactions to peers as soon as their order is determined and a signature afterwards, at regular intervals (we can think of these as "virtual block boundaries"). This means that the peers can validate transactions and stage their changes to the ledger as they arrive, in a streaming fashion. As a result, the end-to-end latency will be reduced to the latency of the ordering operation and the validation for a single transaction. The staleness of data that is used to execute new transaction is also reduced. Persistence is still ensured at the block granularity and the failure model of the system remains unchanged.

The operations required for batched (block-based) and streaming processing are very similar. Signing a block at the end and not at the beginning does not change the contents of the block, nor the amount of work required by the orderer because producing the signature requires anyhow a pass over all the data. The size of the blocks can be configured freely without affecting the perceived latency of the system.

3.2 Integration with a CFT Ordering Sevice

When using a crash fault tolerant (CFT) ordering service, StreamChain peers can connect directly to a single trusted ordering node, whereas in the case of a BFT ordering service (e.g., [10]) they connect to the majority of ordering nodes[1]. The notion of a block is only necessary to amortize the cost of writing to persistent storage and to ensure that blocks can be identified and validated offline (this also maintains backwards compatibility).

3.3 Integration with a BFT Ordering Service

For practical reasons, we further explored a variant of StreamChain in which peers would connect to a single ordering node even in the BFT case. In this case, it is necessary to verify the authenticity of transactions before returning success to the clients. For this, the peer relies on the periodic signatures that validate previously received transactions, but it can nevertheless use the staged data to execute and validate transactions. It is only that a peer cannot commit the staged data until it receives a correct (multi)signature. If a wrong signature is received, a peer would have to roll back staged data. This never results in corrupted state, because previous simulation/execution that depends on the rolled back data would also fail in the validation phase.

For this behavior, we rely on the notion of a "materialized version" of the ledger state (we will call this "state DB" in Fabric) that most systems already implement. This state DB is typically a key-value store that also stores the version number of each key-value pair and is used concurrently by both the execution, validation and commit operations. In a batch-based validation scheme both the state DB

[1] When connecting to $t + 1$ ordering nodes out of $2t$, the peers can stream transactions from one, and the hashes from the rest.

and the ledger on disk are updated at the granularity of a block. In StreamChain, we envision updating the state DB at transaction granularity (but rolling back an entire block) and updating the ledger at block granularity.

To ensure that no transaction that has been executed against a later rolled-back state passes validation, the version numbers associated with key-value pairs in the state DB have to encode their "provenance". Since it is not enough to use a [block-number, transaction-number] tuple currently used in Fabric, we propose including the running hash of transactions in the version number.

4 PROOF OF CONCEPT IMPLEMENTATION

We have built a fairly simple yet realistic prototype version of StreamChain by modifying the behavior of nodes in Fabric (with Kafka ordering service), as shown in Figure 3.
In a nutshell, the changes to the ordering service were as follows:

(1) we changed the configuration of the ordering nodes such that they send out transactions one-by-one
(2) we use ram-disk as a backing storage instead of regular disk drives

We have also changed the implementation of the peers in the following way:

(1) we implemented parallel validation of transaction signatures on streaming transactions
(2) we restructured the commit code-path to allow software pipelining
(3) we added fine-grained locks to the state DB
(4) we used a ram-disk as backing storage instead of regular disk drives

In the following subsections we explain the above changes in more detail.

4.1 Ordering service

In a streaming setup, transactions are transmitted without signatures from the ordering service, with periodic signatures sent at "virtual block" boundaries. We simulate this by removing the code paths creating and verifying signatures from the ordering service (communication is still over TLS and all endorsers sign transactions as usual). Since the cost of verifying a block signature is in the order of 0.5ms on server-grade CPU, it can be neglected for large block sizes. In a real-world implementation it would still be required, however, to include a signature periodically from the orderer. To remove batching delays, we configured the ordering nodes of Fabric to produce blocks with a single transaction inside. These act as envelopes around the transaction.

As we later show in the Evaluation (Section 5.2), throughput suffers with the default ordering behavior that stores each block to disk before sending them to peers. In our proof of concept, this would mean writing each transaction to disk immediately. We simulate the behavior of writing only entire blocks (which is not latency-critical anymore), by replacing the SSD backing the orderer with a ram-disk.

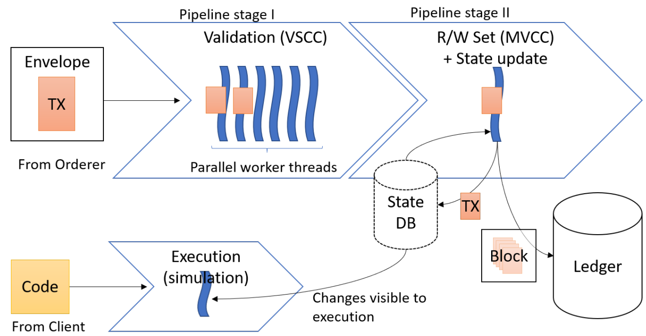

Figure 3: Our StreamChain prototype relies on multiple forms of parallelism (pipeline, threadpool) to reach high throughput while offering low latency processing.

4.2 Validation and Committing

Handling transactions in a streaming fashion inside Fabric peers is straightforward and, as Figure 3 shows, the operations performed in StreamChain are similar to the ones in Fabric. They have been reorganized, however, in a way that allows pipelined validation: we have, on the one hand, parallelized transaction signature checking across blocks[2] and, on the other hand, implemented a two-stage software pipeline. Figure 3 shows the steps performed in each pipeline stage, highlighting the additional parallelism inside the first stage.

Since the state DB can be accessed concurrently at transaction granularity by both the committing logic and the execution of transactions (endorsement), we replaced the single global read-write lock of Fabric with multiple locks, each covering a subset of the key-space. This allows more concurrency in the system and, as a result, lowers the performance impact of committing more often into the state DB. In our prototype, locks are only held as long as a key-value pair is being written or read, which means that the concurrent readers will not see a serializable state. While this opens up a venue for discussion, our reasoning was that if data that an execution (simulation) relies on has been changed by a committing transaction, the validation of the simulated transaction will fail anyhow. The outcome of committed transactions, however, is guaranteed to be linearizable.

As for storage access, to simulate the amortized cost of disk accesses over hundreds of transactions, we have set up the state DB and the ledger such that they are backed by main memory (a ram-disk). This simulates the expected behavior of a full-fledged implementation that would only periodically synchronize state to disk.

5 EVALUATION

We ran our experiments on 9 peers in the IBM Cloud. Each peer has 16 virtual cores (equivalent to 2GHz Intel Xeon E5-2683 cores), 16GBs of memory, SSDs, and a 1Gbps network connection. The workload we execute is composed of MINT transactions of the Fabcoin application as described in [1], with performance reported

[2]Since Fabric 1.1 transaction signatures are verified in parallel inside a block. We use a similar mechanism but allow it to parallelize across blocks.

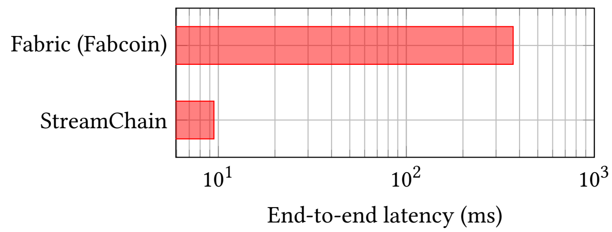

Figure 4: Per-transactions latencies are 34x lower in StreamChain than for Fabcoin running on Fabric.

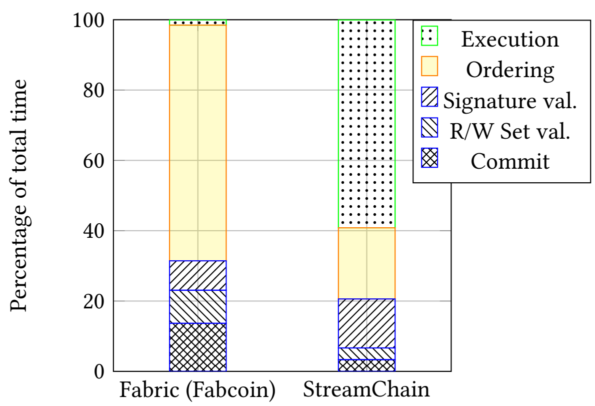

Figure 5: As expected, using blocks increases latencies of ordering step, to the level that it dominates all other operations.

in that work serving as a baseline. Furthermore, we based our ordering service configuration on the setup in [1], namely, four nodes were used for running Kafka and one for the Fabric orderer.

5.1 Latency

Figure 4 shows the absolute gains in end-to-end latency of StreamChain over our baseline (that uses batches of 473 MINT transactions on average), reducing it from 370ms to 10ms. Reflecting back to the discussion in Section 2, this turnaround time is fast enough to allow clients to run in a closed loop and still execute 100 transactions per second for a pure read-modify-write workload.

The end-to-end latency in Fabric comes from four main sources: endorsement (execution), ordering, validation, and network communication time. Since we focus on local-area networks, the latter is negligible. StreamChain does not change the cost of endorsements, so the difference between the two systems is to be found in ordering and validation. Figure 5 shows the importance of the above factors in both systems (100% refers to the total end-to-end latency of each variant). In Fabric, as expected, most of the latency comes from the batching inside the orderer (67%). Furthermore, since every transaction in a block incurs the full validation delay of the block, these account to more than 30% of the latency and endorsement cost per transaction is negligible. In the case of StreamChain, however, ordering and validation contribute almost equally to the latency, and endorsement has a more accentuated effect (for Fabcoin, the

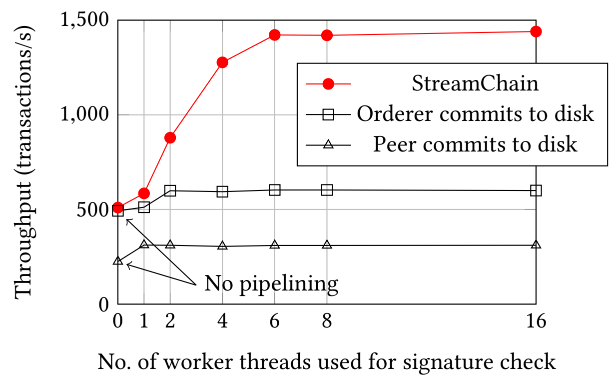

Figure 6: Throughput of a node (running only the validation logic) scales well with increasing parallelism for TX signature checking up to 6 cores. After that point the second pipeline stage (MVCC+Commit) becomes the bottleneck.

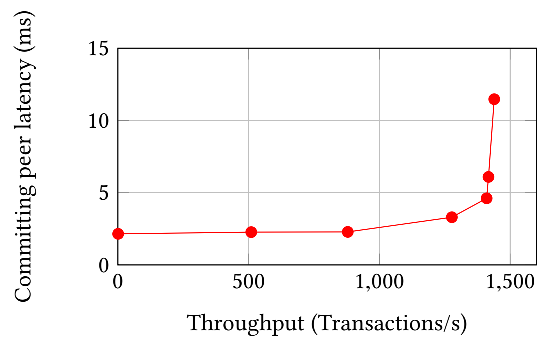

Figure 7: The cost of validation increases very little with throughput. Even close to saturation it is under 12ms.

code performing endorsements runs inside a docker container, as a result there are significant communication and data marshaling overheads in that operation).

5.2 Throughput

Changing the way blocks are handled and constructed in the system solves the latency problem, but when all steps of the validation logic are executed sequentially inside the peer, throughput levels will suffer. This is illustrated by the leftmost points in Figure 6, that represent execution on a single thread with no pipelining inside the peers. If we enable pipelining in StreamChain, throughput increases by a factor of three, then it reaches a plateau around 150 TX/s. The reason for this is that in our current implementation, the time spent in the second pipeline stage is around 0.65ms on average per transaction, which limits the maximum throughput (unless this stage would also be parallelized internally).

Figure 6 shows two additional lines, to demonstrate the throughput of the system when the orderer, or both the orderer and the peers persist changes to an SSD, synchronously, for each transaction. Not surprisingly, throughput is quickly capped by turnaround times to the disk.

Figure 7 shows validation latency at a peer in function of throughput. The minimum time spent in this step is 2.15ms, but even close to saturation StreamChain delivers reliably low latencies (as opposed to the results in [1], where even at low throughput levels this step takes more than 100ms).

It is important to note that the latency values are an upper bound of a full-fledged implementation because in our prototype the state DB has not been optimized for main-memory based operation and it is, in essence, the same disk-oriented key-value store as in Fabric but running on top of a ram-disk.

6 DISCUSSION

6.1 From Proof-of-concept to Production

Here, we discuss additional traits that a future efficient production-grade implementation of StreamChain should possess.

Handling frequent state updates. First, the data structure backing the state DB should be designed for main-memory and allow concurrent readers and writers. Specifically, for Fabric, this described behavior could be achieved by introducing a main-memory-based caching layer on top of persistent LevelDB (currently used to implement the state DB). Execution of transactions and validation should happen exclusively using the cache, and its changes should be only occasionally written back to LevelDB (e.g., at block boundaries).

State updates on disk are also relevant to the ordering nodes, that, as shown in our experiments, will become a bottleneck if each transaction is synchronously flushed to disk (unless using higher IOPS flash-based storage). Thanks to the removal of explicit blocks, the ordering service is free to implement mini-batches if that helps it amortize disk access cost, without any assumption on peer behavior.

Parallel crypto operations. Second, to achieve high throughput, the impact of crypto operations has to be reduced through parallelism and pipelining. Fabric's design is naturally amenable to such pipelining but other, similar, systems should be able to support this behavior as well. While it is, in principle, possible to parallelize other validation operations as well across transactions (e.g., parallelize checking whether transactions conflict on the keys they accessed for reading/writing by exploiting disjoint read-write sets), assuming a main-memory-optimized state DB, the cost of these operations will decrease significantly in the future.

6.2 Other Solutions without Blocks

There is an emerging class of distributed ledgers that do not incorporate the notion of blocks at all and operate on a per-transaction basis. Two well known examples are Corda [3] and IOTA [9], which have been designed with a very specific financial services use-case in mind. The data structure they store data in is not a single chain, but rather a directed acyclic graph (DAG). This allows them to execute transactions between various subsets of the peers in the network and removes the burden of global ordering, but also makes apples to apples comparisons with more traditional distributed ledgers, such as Hyperledger Fabric, difficult.

The decision of batching as a performance optimization, as well as streaming execution is, however, orthogonal to the underlying data structure. StreamChain provides several design advisories that could be applied to Corda or similar systems as well and increase their end-to-end throughput without affecting latencies.

7 CONCLUDING REMARKS

StreamChain demonstrates that it is possible to reduce latencies by an order of magnitude in today's permissioned distributed ledgers without significantly affecting throughput. We achieved this through a simple modification to how blocks are constructed and treated. The technique of switching Fabric over to a stream-based processing should be applicable to other similar systems as well.

Beyond the non-functional improvements, the lower latencies alleviate the staleness problem for EOV systems, enabling a wide range of previously unfeasible use-cases. In the financial sector, for instance, distributed ledgers are only considered for settlement purposes, but with low enough latencies, trading could also become a possibility.

Our experiments also show that in order to push end-to-end latencies to sub-millisecond levels, the ordering service will have to be chosen carefully. It is likely that advanced networking features such as RDMA [8], or implementations using hardware accelerators [5, 6] will have to be used. The most challenging question in this space is how to combine BFT guarantees with such specialized hardware solutions.

REFERENCES

[1] Elli Androulaki et al. 2018. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *Eurosys 2018.*

[2] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quéma, and Marko Vukolić. 2015. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.* 32, 4 (2015), 12:1–12:45.

[3] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. 2016. Corda: An introduction. *R3 CEV, August* (2016).

[4] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)* 20, 4 (2002), 398–461.

[5] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review* 46, 2 (2016), 18–24.

[6] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolić. 2016. Consensus in a Box: Inexpensive Coordination in Hardware.. In *NSDI'16.* 425–438.

[7] Manos Kapritsos et al. 2012. All about Eve: Execute-Verify Replication for Multi-Core Servers.. In *OSDI*, Vol. 12. 237–250.

[8] Marius Poke and Torsten Hoefler. 2015. Dare: High-performance state machine replication on rdma networks. In *HPDC'15.* ACM, 107–118.

[9] Serguei Popov. [n. d.]. The tangle (2017). *URL https://iota. org/IOTA_Whitepaper. pdf* ([n. d.]).

[10] João Sousa, Alysson Bessani, and Marko Vukolić. 2018. A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In *DSN 2018.* 51–58.

[11] Marko Vukolić. 2015. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In *Open Problems in Network Security Workshop (iNetSec).* 112–125.

[12] Marko Vukolić. 2017. Rethinking Permissioned Blockchains. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts (BCC '17).* 3–7.