A Byzantine Fault-Tolerant Consensus Library for Hyperledger Fabric

Artem Barger
IBM Research, Haifa Lab
Haifa, Israel
bartem@il.ibm.com

Yacov Manevich
IBM Research, Haifa Lab
Haifa, Israel
yacovm@il.ibm.com

Hagar Meir
IBM Research, Haifa Lab
Haifa, Israel
hagar.meir@ibm.com

Yoav Tock
IBM Research, Haifa Lab
Haifa, Israel
tock@il.ibm.com

ABSTRACT

Hyperledger Fabric is an enterprise grade permissioned distributed ledger platform that offers modularity for a broad set of industry use cases. One modular component is a pluggable ordering service that establishes consensus on the order of transactions and batches them into blocks. However, as of the time of this writing, there is no production grade Byzantine Fault-Tolerant (BFT) ordering service for Fabric, with the latest version (v2.1) supporting only Crash Fault-Tolerance (CFT). In our work, we address crucial aspects of BFT integration into Fabric that were left unsolved in all prior works, making them unfit for production use.

In this work we describe the design and implementation of a BFT ordering service for Fabric, employing a new BFT consensus library. The new library, based on the BFT-Smart protocol and written in Go, is tailored to the blockchain use-case, yet is general enough to cater to a wide variety of other uses. We evaluate the new BFT ordering service by comparing it with the currently supported Raft-based CFT ordering service in Hyperledger Fabric.

KEYWORDS

Blockchain, Distributed Ledger

ACM Reference Format:
Artem Barger, Yacov Manevich, Hagar Meir, and Yoav Tock. 2020. A Byzantine Fault-Tolerant Consensus Library for Hyperledger Fabric. In *Proceedings of* . ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Blockchain technology became popular with the advent of Bitcoin, a cryptocurrency platform that employs "nakamoto consensus" – an algorithm based on Proof of Work (PoW) – to order transactions. Subsequent blockchain platforms like Ethereum popularized the notion of the "smart contract" as a means for executing sophisticated transactions. These pioneering platforms contain a great technological potential for transforming the way we do business. However, for enterprise applications they contain two major flaws that sparked additional innovation.

Consensus mechanisms based on Proof of Work suffer from low transaction rates, long settlement times, and are widely criticised for their exorbitant power consumption. As a consequence, blockchain platforms started using Byzantine Fault Tolerant (BFT) consensus mechanisms (e.g. PBFT [21]) as a replacement for PoW (see Tendermint [17]). Large-scale blockchain applications present tough requirements in terms of scalability, fairness, and robustness, which yielded a wave of innovation in BFT research (e.g. HoneyBadgerBFT [48]), and systems that extend the traditional BFT model (e.g. Stellar [46]). For a review see "Blockchain Consensus Protocols in the Wild" [20].

Bitcoin and Ethereum are pseudo-anonymous open membership platforms. This conflicts with the stringent regulations on financial systems (e.g. KYC & AML), and the privacy needs of business networks. Enterprise oriented blockchain platforms address these concerns by creating *"permissioned"* networks, where participants are identified by their real-world identity and organizational affiliations. Privacy concerns are handled by providing *privacy preserving* collaboration patterns such as: (1) the dissemination of information on a need to know basis (e.g. Corda [15]), (2) the definition of segregated communication channels (e.g. Hyperledger Fabric [9]), (3) the use of advanced cryptography (e.g. ZK [53, 60]), as well as other techniques, and combinations of those.

Hyperledger Fabric (or just Fabric) is an open source project dedicated to the development of an enterprise grade permissioned blockchain platforms [9]. Fabric employs the *execute-order-validate* paradigm for distributed execution of smart contracts. In Fabric, transactions are first tentatively *executed*, or endorsed, by a subset of peers. Transactions with tentative results are then grouped into blocks and *ordered*. Finally, a *validation* phase makes sure that transactions were properly endorsed and are not in conflict with other transactions. All transactions are stored in the ledger, but valid transactions are then committed to the state database, whereas invalid transactions are omitted from the state.

In the heart of Fabric is the ordering service, which receives endorsed transaction proposals from the clients, and emits a stream of blocks. At the time of writing, the latest Fabric release (v2.0)

, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

uses the Raft [5, 49] protocol which is Crash Fault Tolerant (CFT). Despite previous efforts to do so, Fabric still does not have a BFT ordering service. In this paper we describe our efforts to remedy this predicament.

The latest attempt to provide Fabric with a BFT ordering service was by Sousa et al. in 2018 [58], which adapted the Kafka-based ordering service of Fabric v1.1 and replaced Kafka with a cluster of BFT-Smart servers. That attempt was not adopted by the community [4] because of various reasons (elaborated in Section 7). The four top reasons were: (1) it was a two process two language solution, and (2) it did not address some of the more difficult, configuration related work flows of Fabric. Moreover, since it was built on the Kafka architecture and used an external monolith BFT cluster, (3) followers were not able to validate the transactions proposed by the leader during the consensus phase, and (4) it missed out an opportunity to perform several much needed optimizations that exploit the unique blockchain use-case.

In the time that passed since then, Fabric incorporated the Raft protocol as the core of the ordering service, significantly changing the Ordering Service Node (OSN) in the process. Our goal was to implement a BFT library in the Go programming language, that would be fit to use as an upgrade for Raft. Our prime candidates were BFTSmart and PBFT. We soon realized that simply re-writing the Java (or C) implementation in Go will not make the cut. The blockchain use case offers many opportunities for optimizations that are absent from a general-purpose transaction-ordering reference scenario. In addition, Fabric itself presents some unique requirements that are not addressed by traditional protocol implementations. We therefore set out to design and implement a BFT library that on the one hand addresses the special needs and opportunities of a blockchain platform, but on the other hand is general and customizable enough to be useful for other use cases.

One of our goals was to provide an end-to-end Fabric system that addresses all the concerns that a BFT system must face. This forced us to tackle issues that span the entire Fabric transaction flow – from the client, the ordering service, and the peers, to the structure and validation of blocks. The result is the first fully functional BFT-enabled Fabric platform. Our key contributions are:

- **Consensus library** A stand-alone Byzantine fault-tolerant consensus library, based on BFT-Smart. The code is open sourced [44] and is written in the Go programming language.
- **Interface** An easy to integrate interface of a consensus library, suited for blockchains. This interface captures the special needs of a blockchain application, but is fully customizable for other use cases as well.
- **Byzantine fault-tolerant Hyperledger Fabric** A full integration of the library with Hyperledger Fabric [9]. This code is also publicly available and open sourced [43].
- **Evaluation** An evaluation of our BFT version of Fabric versus Fabric based on Raft. The evaluation demonstrates that our implementation is comparable in performance to the earlier BFT-Smart based implementation [58], but slower than Raft, mainly due to the lack of pipelining.

The rest of the paper is organized as follows. Section 2 introduces some background on blockchain, consensus protocols and Hyperledger Fabric. In Section 3 we provide a high level view of Fabric's new BFT Ordering Service Node, describing the main internal components and their relation to the library. Section 4 provides a more detailed exposition of the BFT consensus library we developed, whereas Section 5 describes the additions and modifications we had to perform to Fabric's orderer, peer, and client, in order to turn it into an end-to-end BFT system. In Section 6 we evaluate the fully integrated BFT Fabric system, and compare it to the current Raft-based implementation. Finally, Sections 7 and 8 discuss related work and summarize our conclusions, respectively.

2 BACKGROUND

2.1 Blockchain technology

Blockchain could be viewed as a distributed append-only immutable data structure - a distributed ledger which maintains ordered transaction records between distrusting parties. Transactions are usually grouped into blocks, and then, parties involved in the blockchain network take part in a consensus protocol to validate transactions and agree on an order between blocks. Blocks are linked with a hash chain, which forms an immutable ledger of ordered transactions (see Figure 1). Each party is responsible of maintaining its own copy of the distributed ledger, not assuming trust on anyone else.

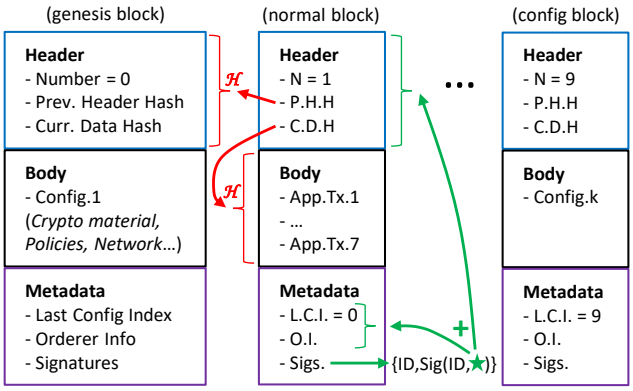


Figure 1: Fabric block structure and hash chain (simplified from [31]). A hash chain is established by including in each header the current data hash and the previous block header hash (red). Orderers sign a concatenation of the header and two fields from the metadata: the last config index and orderer info (green). The metadata signatures field contains an array of signing identities (orderers) and their signatures. Config blocks carry a single configuration transaction, whereas normal blocks carry a batch of application transactions. The first (genesis) block is always a config block. The metadata last config index field references the last config block, against which the block is validated.

The consensus protocols used in blockchain platforms vary greatly. Most enterprise blockchain platforms abandoned the inefficient and wasteful PoW in favor of some variant of a quorum-based consensus protocol. These protocols reach a decision on a sequence of values – which in the blockchain case is the identity and order of

transactions – when a quorum Q of nodes out of a predetermined assembly N are in agreement (see also State Machine Replication [54]). Loosely speaking, when the number of faults to protect against is F, crash fault tolerant (CFT) protocols (e.g. Paxos [35, 36], Raft [49], ZooKeeper [2]) need a simple majority ($Q = F+1$ out of $N = 2F+1$), whereas Byzantine fault tolerant (BFT) protocols require a $\frac{2}{3}$ majority ($Q = 2F + 1$ out of $N = 3F + 1$). BFT is more expensive than CFT: it requires more replicas to project against the same number of faults, executes an additional communication round (3 vs. 2), and requires the use of cryptographic primitives (digital signatures or message authentication codes). The benefit is protection against arbitrary faults – including malicious behavior and collusion between failed nodes – which is essential in blockchain platforms with distributed trust at their core. Despite their cost, BFT protocols are orders of magnitude faster then PoW based protocols, with a negligible energy consumption.

2.2 Hyperledger Fabric

Hyperledger Fabric is an open source project, released by the Linux Foundation[1]. It introduces a new architecture for enterprise grade permissioned blockchain platforms following the novel paradigm of *execute-order-validate* for distributed execution of smart contracts (*chaincode* in Fabric).

A distributed application in Fabric is basically comprised of two main parts: 1) *Chaincode* is business logic implemented in a general-purpose programming language (Java, Go, JavaScript) and invoked during the execution phase. 2) *Endorsement policies* are rules which specify what is the correct set of peers responsible for the execution and approval of a given chaincode invocation [10, 45].

The Fabric blockchain network is formed by nodes which could be classified into three categories based on their roles: 1) *Clients* are network nodes running the application code, which coordinate transaction execution. Client application code typically uses the Fabric SDK in order to communicate with the platform. 2) *Peers* are platform nodes that maintain a record of transactions using an append-only ledger and are responsible for the execution of the chaincode and its life-cycle. These nodes also maintain a "state" in the form of a versioned key-value store. Not all peers are responsible for execution of the chaincode, but only a subset of peers called *endorsing peers*. 3) *Ordering nodes* are platform nodes that form a cluster that exposes an abstraction of atomic broadcast in order to establish total order between all transactions and to batch them into blocks.

In order to address the privacy concerns of business partners, fabric introduces the concept of channels. A channel in Fabric allows a well defined group of organizations that form a consortium to privately transact with each other. Each channel is essentially an independent private blockchain, with its own ledger, smart contacts, and a well defined set of participants. An additional privacy tool in Fabric is private data, which allows channel participants to expose data to select members of the channel, whereas other members only see a digest of said data.

2.2.1 Transaction flow. The following summarizes the execution flow of a transaction submitted by a client into Fabric (see Figure 2):

1. The client uses an SDK to form and sign a transaction proposal. Next, the client sends the transaction proposal to a set of endorsing peers.
2. Endorsing peers simulate the transaction by invoking the chaincode, recording state updates, and producing an output in the form of a versioned read-write set. Next, each endorsing peer signs the read-write set and returns the result back to the client.
3. The client collects responses from all endorsing peers, and validates that all endorsing peers have signed the same payload. It then concatenates all the signatures of the endorsing peers along with the read-write sets, creating a transaction.
4. The client then submits the transaction to the ordering service by invoking an "atomic broadcast" API. In the Raft-based ordering service the client submits to a single orderer.
5. The ordering service collects all incoming transactions, packs transactions into blocks, and then orders the blocks to impose total order of transactions within a channel context.
6. Blocks are delivered to all the peers using a combination of some peers pulling blocks directly from the ordering service and a gossip-based dissemination mechanism between the peers.
7. Upon receiving a new block, each peer iterates over the transactions in it and validates: a) the endorsement policy, and b) performs multi-version concurrency control checks against the state.
8. Once the transaction validation has finished, the peer appends the block to the ledger and updates its state based on valid transactions. After the block is committed the peer emits events to notify clients connected to it.

2.2.2 The ordering service. Fabric is a modular blockchain system supporting multiple types of ordering services. In Fabric's first release (v1.0) the ordering service was base on Kafka [1], a replicated, (crash) fault tolerant messaging platform. The ordering service nodes (OSNs) sent transactions to a Kafka topic (one for each channel), and consumed from it an ordered transaction stream. Then the OSNs employed a deterministic function that cut identical blocks across all nodes. In this architecture the ordering nodes did not communicate between them directly; they only acted as producers and consumers of a Kafka service. Moreover, every node was servicing all the channels.

On release v1.4.1 Fabric introduced an ordering service based on a Go implementation of the Raft [49] consensus algorithm from *etcd* [3]. This significantly changed the architecture of the OSN. Each channel now operates an independent cluster of Raft nodes. An OSN can service multiple channels, but is not required to service all of them. This permits linear scalability in the number of channels by spreading channels across OSNs (see Figure 3). Raft is a leader-based protocol. The leader of each cluster (channel) batches incoming transactions into a block, and then proposes that block to the consensus protocol. The results is a totaly ordered stream of blocks replicated across all OSNs that service the channel. Clients are required to submit transactions to a single node, preferably the leader. However, transactions may be submitted to non-leader nodes, which then forward them to the leader. At the time of this

[1] www.linuxfoundation.org

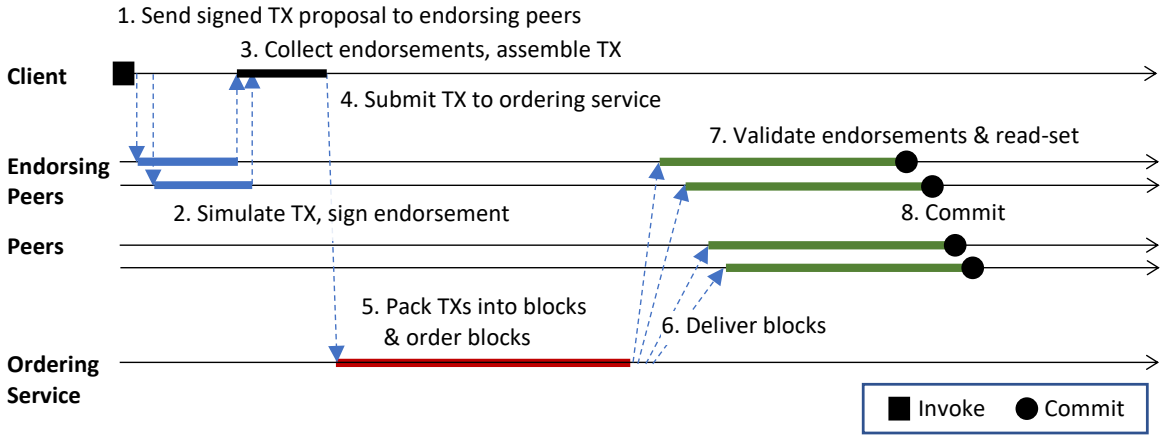


Figure 2: Transaction flow in Fabric.

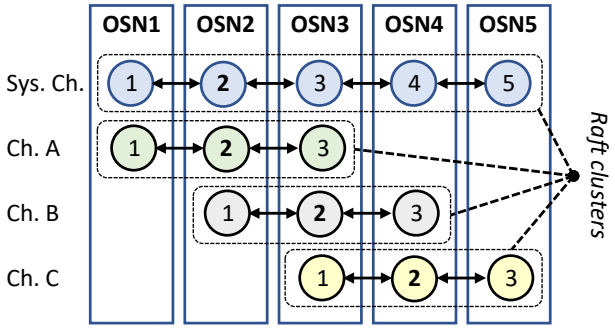


Figure 3: Fabric Ordering Service Nodes (OSNs) host multiple Raft clusters, one for each channel. The participation of OSNs in channels is configurable. The system channel is an optional mechanism for coordinating the creation of application channels [28]. If used, it must be deployed on all OSNs.

writing Fabric's latest version (v2.1) offers the Raft-base ordering service as default, and deprecated the Kafka-base option.

2.3 BFT-SMaRt

Parts of the library presented in this paper were designed based on the BFT-SMaRt consensus library [13]. BFT-SMaRt implements a modular state machine replication protocol on top of a leader driven Byzantine consensus algorithm [57]. The message pattern in the normal case is similar to the PBFT protocol [21], i.e. PRE-PREPARE, PREPARE, and COMMIT phases/messages.

If the current leader is faulty, a new leader is elected using a *view change* protocol, which we implement with the *synchronization* phase of BFT-SMaRt [57] in mind. The view change process uses three types of messages: VIEW CHANGE, VIEW DATA, and NEW VIEW. If there is a reason to suspect the leader, such as a timeout on a client's request, followers send a VIEW CHANGE message to all nodes. This message is very short, carrying only the desired next *view* number, and is required to ignite the view change process. Next, if a quorum of VIEW CHANGE messages is received, then nodes send a signed VIEW DATA message to the potentially new leader, containing all needed information, such as the latest *checkpoint*. Once the new leader receives at least a quorum of VIEW DATA messages and collects enough data about the current state, then it sends a NEW VIEW message, comprised of the collected VIEW DATA messages, to inform all nodes of the elected new view and leader and the current state.

In addition, we follow the BFT-SMaRt suggested usage of clients' requests timeouts. If a timeout is triggered on a client's request for the first time, the request is forwarded to the leader. This is done because a faulty or malicious client may have sent its request only to some of the nodes, therefore triggering a view change. If there is a second timeout for the same request, then the node starts the view change process.

3 ARCHITECTURE

In this section we describe the architecture of the BFT OSN, depicted in Figure 4. We'll briefly describe the different components composing an OSN and the interactions between them, deferring the full details to sections 4 & 5, below.

After a client collects endorsements and assembles a transaction (see Figure 2), it tries to submit it to all the OSNs (elaborated in Section 5.5). An OSN will first *filter* incoming transactions using several rules encoded in the *validator*; for example, verifying the client's signature against the consortium's definitions, rejecting transactions that arrive from un-authorized clients. Valid transactions are then submitted to the BFT *consensus library* for ordering. Submitted transactions are queued in a request pool within the library.

The consensus library is leader based, and at any given moment the node will operate either as a leader or a follower. A leader OSN will batch incoming transactions, and then call the *assembler* to assemble these transactions into a block. The *assembler* returns

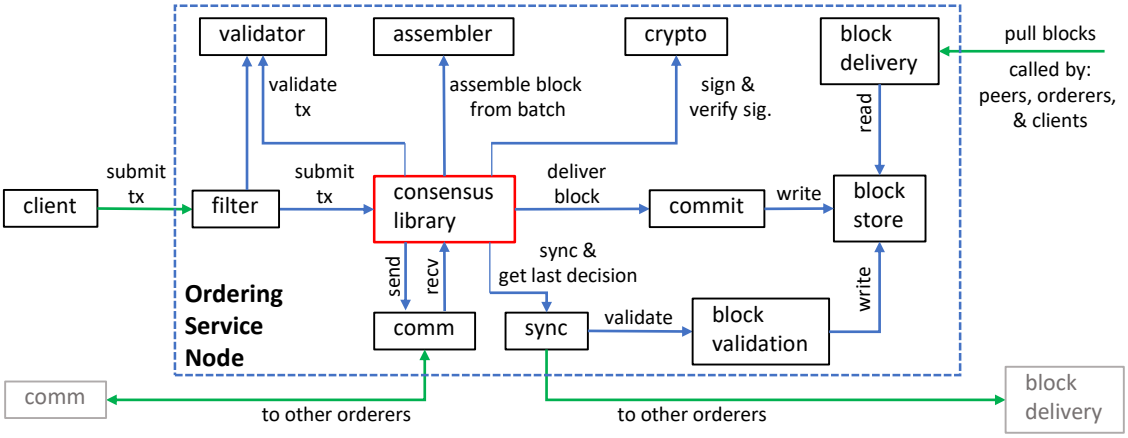


Figure 4: The architecture of a Fabric BFT-based Ordering Service Node.

a block that may contain some or all of the transactions in the batch, and may even reorder them. The assembler is in charge of ensuring that block composition adheres to the rules and requirements of Fabric. Transactions that were not included in the block will be included in a later batch. The leader will then propose the block, starting the consensus protocol. A follower node will receive this block proposal from the *communication* component and will revalidate every transaction in it using the same *validator* used by the *filter*. This is done because in a BFT environment the follower cannot trust the leader – it must therefore validate the proposal against the application running on top of it – the Fabric blockchain. After a block proposal passes through the first two phases of consensus the *commit* message is signed using the *crypto* component and sent to all the nodes. Every node that receives a commit uses the crypto component to verify the signature on it. When enough valid commit messages accumulate, the block along with all the Q commit signatures is delivered to the *commit* component that saves it into the *block store*. We call the block and commit signatures the "decision".

Each OSN has a *block delivery* service that allows peers, orderers and clients to request blocks from it. This is how peers get their blocks, and this is how an OSN that was left behind the quorum catches-up. If the consensus library suspects that it is behind the frontier of delivered decisions, it will call the *sync* component, which in turn uses the *block delivery* service of other OSNs in order to pull the missing blocks. Every block that it receives goes through a *block validation* component that makes sure it is properly signed by Q OSNs, as required by the BFT protocol. When it finishes, it will provide the library with the most recent decision.

This architecture resembles the Raft-based OSN architecture in the sense that the consensus library is embedded within the OSN process. However, there are important differences:

- In a Raft-OSN the block is cut before ordering and is then submitted to the raft library for ordering. Here transactions are submitted to the BFT library, because the library monitors against transaction censorship.
- In a Raft-OSN the followers trust the leader's block proposal (as it can only crash fail), whereas here the followers must revalidate the transactions within the block proposal.
- In a Raft-OSN the delivered block is signed after consensus by the node, right before it is saved to the block store. Here the block is signed during the consensus protocol by Q nodes.

The requirements of integrating a BFT consensus library into the Fabric OSN guided us into the definition of a library API that is on the one hand perfectly suited to the blockchain domain, but on the other hand is general enough to be useful in a variety of other domains. In the next section we shed more light on the consensus library and its API.

4 THE CONSENSUS LIBRARY AND API

We implemented a BFT consensus algorithm in a stand-alone library. The code is open sourced [44] and is written in Go. The consensus algorithm is mostly based on the well known PBFT algorithm [21, 22] and the BFT-SMaRt state machine replication protocol [13, 57]. In order to make the library production ready we employ some well known techniques, such as heartbeats.

The library is not aware of what kind of application is using it. However, it is still suited for integration with a permissioned blockchain. This is possible by designing an interface that allows the use of blockchain like applications. This API includes abstractions such as communication, crypto primitives, and replication. The application is expected to provide implementations for these abstractions. When implementing these interfaces the Byzantine fault-tolerant nature should be taken into account, as elaborated next.

4.1 An interface suited for a blockchain application

In this section parts of the library's interface is described and how it is suited for a blockchain application. Figure 5 illustrates the API calls made from the library to the application during the normal case of consensus, and Figure 6 provides a code example of the API.

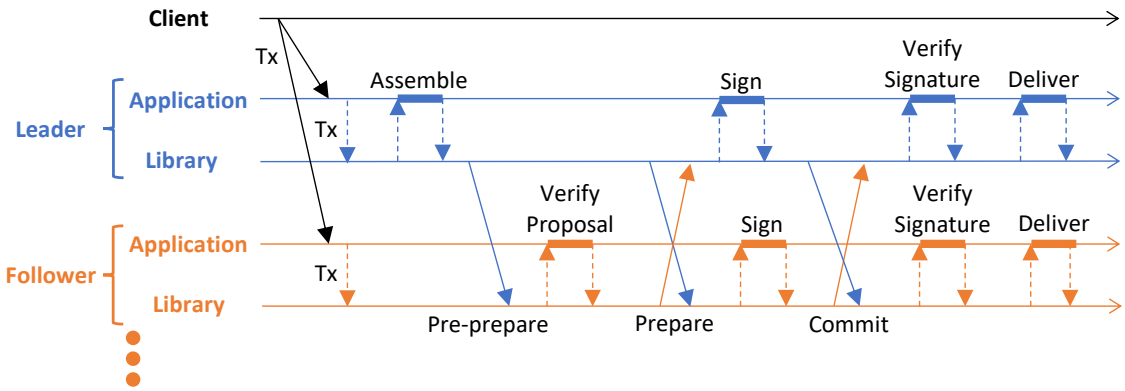


Figure 5: Normal case library-application flow.

```
Assemble(metadata []byte,
  requests [][]byte) Proposal
VerifyProposal(Proposal) error
SignProposal(Proposal) Signature
VerifySignature(Signature) error
Deliver(proposal Proposal,
  signatures []Signature) Reconfig
```

Figure 6: API code example.

4.1.1 Assembler. In order to build a new proposal for the next round of consensus, the current leader turns to the application by calling the assembler. The leader gathers some requests/transactions into a batch and gives this batch along with some metadata, such as the sequence and view number, to the assembler. The application, which implements the assembler, can now construct the proposal out of the given batch and metadata in any form it wishes.

In the case of a blockchain application, this enables adding block specific information, such as the hash of the previous block. Moreover, in Fabric, a configuration transaction must reside in a block by itself. This rule is enforced by the assembler we implemented within Fabric, as elaborated in Section 5.2.

4.1.2 Signing and verification. Signing occurs in two places in the library: 1) during the COMMIT phase nodes sign the current proposal and send the signature as part of the COMMIT message 2) each node signs its VIEW DATA message during the view change process.

As part of the COMMIT phase each node collects a quorum of COMMIT messages, including a quorum of signatures over the proposal. For a permissioned blockchain application this means that blocks will be signed by a quorum of nodes as part of the consensus process, and these signatures act as a proof that the block has successfully undergone all the phases of consensus, as desired by these types of applications. Using the consensus process for signature collection is done also by other blockchain specific consensus libraries, such as Helix [11].

The actual signing and verification processes take place outside of the library, by the application. Exposing an API for signing and verification enables a plugable process and does not limit the application to any kind of specific implementation.

4.1.3 Delivery. Once a node receives a quorum of commit messages then it delivers the committed proposal along with signatures to the application. It is up to the application to store the delivered proposal and signatures, as the library may dispose of this data once the deliver call returns.

4.1.4 Verify proposal. A follower that receives a proposal in the pre-prepare message from the leader must validate it, as the leader may be malicious and try to harm the liveness of the entire cluster by proposing harmful proposals. On the library side the metadata, such as view number and sequence number, are verified. Then, the library calls VERIFYPROPOSAL so that the application may run any kind of checks it wishes. For example, in Fabric, we make sure that the proposal is constructed as a valid and correctly numbered block, its header contains the hash of the previous block, and all transactions in the block are valid Fabric transactions.

4.1.5 Synchronization. There are some cases where a node suspects it is behind by several commits and it initiates a replication protocol. The library calls SYNC and the application is responsible for running a replication protocol. The application must make sure the replication is suited for a Byzantine environment, meaning it needs to validate signatures of a quorum of nodes on the proposals fetched, otherwise malicious nodes would be able to forge proposals. In Section 5.2 we specify our implementation in Fabric, which serves as an example for a Byzantine aware implementation.

Even though the library delegates data synchronization to the application layer, it still needs to take care of a special corner case: If the cluster has gone through a view change without a new proposal being committed afterwards, the latest proposal that a late node will synchronize doesn't contain information about the view change. To that end, after the synchronization ends, the control is returned from the application to the library, and it performs remote queries to all nodes and seeks to obtain answers from $F + 1$ different nodes

that agree on a view higher than known to the synchronized node. This ensures the node can catch up with the cluster and improves fault tolerance.

4.1.6 Communication. The library assumes the existence of a communication infrastructure that implements point to point authenticated channels. The infrastructure notifies the library whenever a new message arrives as well as the identifier of the node that sent it, and sends messages to requested nodes on behalf of the library. The library distinguishes between two types of messages:

- **Forwarded clients' requests**: Recall, as per the consensus algorithm, clients send requests to all nodes and not just to the leader. Whenever a client's request is not included in a batch in a timely manner, every non-faulty follower sends the request to the leader. This is done to prevent malicious clients from sending messages only to followers, forcing a view change and thus impairing liveness.
- **Consensus protocol messages**: This includes all other messages sent between nodes, namely pre-prepares, prepares, commits, view change protocol messages, and heartbeats. Unlike request forwarding, here it is vital that the message sending would be a wait free operation, otherwise the execution of the protocol can be indefinitely stalled.

4.1.7 WAL. The WAL (write-ahead log) is a plugable component, and we provide a simple implementation (the implementation is inspired by the WAL in Raft [5]). The WAL merely maintains the latest state, and does not store the entire history. The main part of the storage is delegated to the application, by assuming that DELIVER returns only after safely storing the committed proposal and signatures. This is suitable for a blockchain application.

4.1.8 Verify client's request. The library assumes the application submits only valid clients' requests, however, there are cases where a client's request must be reverified: 1) if a follower forwards a request to the leader after a timeout, the leader calls VERIFYREQUEST 2) if the configuration changed, as elaborated next, then all of the clients' requests must be reverified.

4.2 Reconfiguration

The library supports dynamic (and non-dynamic) reconfiguration. Unlike other common consensus libraries such as etcd/raft, reconfiguration is done implicitly and not explicitly: Instead of the application sending an explicit reconfiguration transaction to the library, the library infers its configuration status after each commit. This enables us to reconfigure the system in an atomic manner.

To that end, the library exposes an API for delivery and synchronization that returns a RECONFIG object that states whether there was a reconfiguration and what is the new configuration. In the case of synchronization, the reconfiguration could have been in any of the replicated proposals, and not necessarily in the last one. If the returned RECONFIG object states that there was a reconfiguration, the library closes all of its components and reopens them with the new configuration. Since there is no pipeline implemented in the library, there is no need for any kind of flush after a reconfiguration.

Moreover, since reconfiguration in the application layer is derived from transactions, we state that it is up to the application to validate whether a proposal contains a valid reconfiguration transaction. Specifically, it is expected that proposal verification (Sec 4.1.4) for proposals carrying reconfiguration transactions would involve further checks than standard transactions, including but not limited to whether the client that signed the transaction has administrative privileges.

In addition, the configuration may affect the validity of the clients' requests. And so the application maintains a VERIFICATIONSEQUENCE as an identifier of the current configuration. This sequence is checked by the library after each DELIVER and SYNC, and if the sequence changed then the library reverifies all of the pending clients' requests.

5 BYZANTINE FAULT-TOLERANT HYPERLEDGER FABRIC

In order to turn Fabric into an end-to-end BFT system we had to make changes that go beyond replacing the Raft consensus library with our BFT library. In this section we describe the important changes that had to be made.

5.1 Block structure and signatures

Fabric's block is composed of a header, a payload, and metadata. The header contains the block number, the hash of the current payload, and the hash of the previous block. The payload contains transactions. The metadata contains three important fields: (a) the index of the last config block, (b) a consenter specific information, and (c) orderer signatures (see Figure 1).

The signature is over the block header, fields (a) and (b) of the block metadata, and the signer's identity.

Our implementation does not change the Fabric block structure. However, we change the type of consenter dependent information in it to the ViewID and Sequence number from the consensus library. Moreover, unlike in a Raft-based orderer, orderers sign the block during the consensus process, such that when a block is delivered from the library, its respective metadata signatures field already contains Q signatures. In contrast, in a Raft-based orderer each orderer independently signs the block just before commit.

5.2 The orderer node

Changing the orderer node to work with our library amounts to implementing the logic that initializes the library, submits transactions to the library, and handles invocations from the library. The invocations from the library are defined by the interfaces described in the previous section, and require the implementation of a few simple components that reuse existing Fabric capabilities.

Synchronization When an orderer node falls behind the cluster, it needs to catch-up and synchronize. This is done by requesting blocks from other cluster members. First, the node polls all members of the cluster for their block height. It then chooses a neighbor to pull blocks from. In the Raft implementation, the node with the highest block number is selected, and is requested for blocks up until that number. However, in a Byzantine environment we have to take into account that nodes may lie about their height. A malicious node may declare a large height and stall the synchronization. We therefore sort the heights and take the $(f + 1)^{th}$

largest height as the height-target. The node then starts pulling blocks from all neighbors (instead of just one) up until the height-target.

Block assembly The block assembler is in charge of inspecting a batch of incoming transactions offered to it by the leader of the cluster, and returning a block of transactions that is build in accordance with the semantics of the blockchain implemented on top of the library. In Fabric, these rules indicate that a configuration transaction must be packed in a block by itself. The assembler is the place for incorporating transaction reordering optimizations such as the one described in [56].

Signer and Verifier The signer and verifier implementations simply wrap the already existing capabilities of Fabric to sign and verify data.

5.3 Block validation

Every time a fabric node receives a block from another node outside of the consensus process, it must validate that the block is properly signed. This happens in three different scenarios: 1) when a peer receives a block from the ordering service, 2) when a peer receives a block from another peer during the gossip protocol, and 3) when an ordering node receives a block from another ordering node during the synchronization protocol. For that end, Fabric defines a block validation policy, which specifies the criteria that the signatures on the block must match. The default policy in Fabric is to require a single signature from a node that belongs to any of the organizations that compose the ordering service consortium. For a BFT service, this is obviously not enough.

We implemented a new validation policy that requires that the block be sighed by Q out-of N nodes. Moreover, we require that these signatures are from specific nodes – the consenter set defined in the last channel configuration transaction. This means that the new validation policy we defined must be dynamically updated every time the consenter set is updated with a configuration transaction, which was previously not supported.

5.4 The peer

If Fabric, blocks are disseminated to the peers using a combination of techniques. First, the peers that belong to the same organization elect a leader. Then, that leader receives blocks from the ordering service by requesting a stream of new blocks from a randomly selected ordering node – this is called the "delivery service". Then, all the peers including the leader engage in a gossip protocol in order to further disseminate the blocks to all the peers. If non-leader peers suspect the leader, they re-elect a new one.

In our BFT implementation, the blocks that are received from the ordering service cannot be forged, because they are signed by at least $2F + 1$ ordering nodes. However, receiving the stream of blocks from a single orderer exposes the peer to a censorship attack. In fact, up to F orderers may collude and mount this attack. Note that this behavior may also result from bugs or mal-functions that cause the orderer to stop sending new block, and not necessarily from malicious behavior. To protect against this scenario we implemented a block delivery service that is resilient to this sort of attack.

A naive solution is for the peer to ask for the block stream from $F + 1$ ordering nodes. This is expensive because blocks are large. In contrast, our approach takes advantage of the Fabric block structure, in which the orderer signs the hash of the block content, not the content itself (see Figure 1). This allows a peer to verify the signatures on a received block even if the block's content is omitted – using only the block header and metadata.

We *augment the delivery service* and allow a peer to request a new kind of delivery – a stream that contains only the header and metadata of each new block – a *Header-Sig* stream . The peer requests a stream of full blocks from a randomly selected orderer, and a *Header-Sig* stream from all the rest. The peer then monitors the front of delivered blocks: if F or more *Header-Sig* streams are ahead of the full block stream for more than a threshold period, then the peer suspects the orderer that delivers full blocks. When that happens, the peer demotes that orderer to deliver a *Header-Sig* stream, and selects another orderer to deliver a full block stream.

5.5 The client

In the Raft CFT setting a client is only required to submit a transaction proposal to a single orderer. If the orderer is not the leader, it simply forwards that transaction to the leader. In a BFT setting that does not work, because any F orderers may be malicious and simply drop the transaction. We therefore modified the client in the Fabric Java SDK [32] to submit every transaction to all the orderers.

6 EVALUATION

We evaluate the performance of our Byzantine Fault-Tolerant ordering service for Hyperledger Fabric that integrates our consensus library.

6.1 Setup

We consider both LAN and WAN setups, and evaluate the performance with different cluster sizes (4, 7, 10) and with various configurations of batch sizes or number of transactions per block (100, 250, 500, 1000). We use a homogeneous virtual machine specification for all nodes with a 16 core Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz with 32GB RAM and SSD hard drives.

To ascertain our system can be deployed in real life scenarios, in our WAN setup we spread the servers across the entire globe to maximize the latency: We deploy servers in Dallas, London, Washington, San Jose, Toronto, Sydney , Milan, Chennai, Hong Kong, and Tokyo. In all of our test runs, the leader is located in London and the average latency from it to the followers is 133ms with a median of 112ms, and ranges starting from a few dozens milliseconds in the case of followers in the same continent, to 250ms in the worst case (to Tokyo). We repeat each test 10 times and present the average result.

In order to generate traffic, we pre-allocate an in-memory heap of 1.4 million Fabric transactions each sized 3,993 bytes (fit for a transaction which contains 3 certificates of endorsing peers) and send it to the leader from 700 concurrent workers. Sending transactions at a high rate from 700 different threads ensures the leader always saturates the blocks as much as the configured batch size allows it. Then, we pick a node and analyze its logs to determine

the time it took to commit a specified number of blocks and compute the throughput. Additionally, we measure the time it takes to perform CPU computations and I/O operations to further analyze the bottlenecks and argue about possible optimizations.

6.2 Performance: BFT versus Raft

We compare our BFT ordering service with the existing state of the art Raft ordering service (Fabric v2.1) and benchmark it in the same setup with similar cluster sizes (5 to 11).

LAN evaluation: The first experiment is conducted with a LAN setup. Figure 7 depicts the throughput (in transactions per second) with various block sizes (derived from the number of transactions in a block, as all transactions are identically sized).

The throughput of our BFT type ordering service is dominated by the block size, as the more transactions that are fit into a block, the more transactions are transferred over the network in a consensus round which involves the 3 phases.

For BFT, A rate of 2,500 transactions per second means sending 80Mb of traffic per second per node, and even in a cluster of 10 nodes, it means a total fan-out of 720 Mb per second. In a local area network such a throughput is not enough to be a bottleneck, and this is aligned with our results where we get similar throughput regardless of cluster size. In contrast, for Raft the situation is different, and at a throughput of 12,000 transactions per second, the leader sends 375Mb to each of the 10 followers which starts hindering the throughput, and indeed we see that the throughput of an 11-sized cluster is smaller than the counterparts. Oddly enough, the throughput of the 7-sized cluster is slightly higher.

Since the latencies in block propagation in our LAN setup are negligible and range between less than a millisecond to a few milliseconds, we attribute the differences in throughput mainly to the high CPU processing overhead that is mandatory in BFT, but doesn't exist in Raft. We measured the time it took to perform local CPU computations for block verification, such as signature checks on transactions, and hash chain validation on blocks. Additionally, we measure the time to perform various I/O operations that are part of the consensus stages such as writing to the Write-Ahead-Log and we present them in (6.3).

WAN evaluation: Next we present an experiment with a WAN setup. Figure 8 shows the performance of a setup identical in terms of cluster size and block size, however the servers are deployed across 10 different data centers across the globe.

Here, throughput is significantly reduced both in BFT and in Raft. In the case of 1,000 transactions per block, the BFT throughput is reduced to 40% of the LAN setup, and to 20% in for Raft.

Recall that in our BFT, the leader starts consensus on a block only when the previous block has been committed. In contrast, Raft can consent on blocks that extend the blockchain despite the previous blocks haven't been consented yet. An interesting phenomenon is that such a pipelining mechanism should in theory make up for the high latency, however our measurements of Raft reach a throughput about 3,000 TPS even in the case of 5 nodes, which hint that the WAN bandwidth is saturated very early.

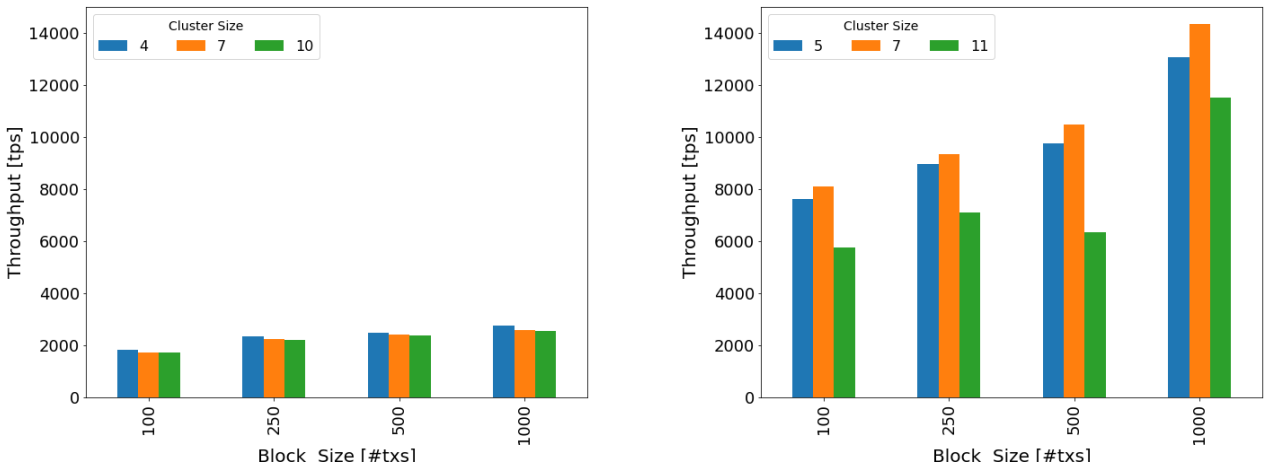
Unlike in the LAN case, in the WAN setup for BFT, the latency plays the biggest role and not the CPU computation, and as a result the drop in performance is more severe as the block size decreases.

6.3 Latency of BFT consensus

Our consensus algorithm is non-pipelined, meaning that the leader sends a pre-prepare for block i only after it has committed block $i-1$. Hence, when deploying our system in a low latency and high bandwidth environment such as a local area network, the performance is greatly effected by local operations such as CPU computation and disk I/O.

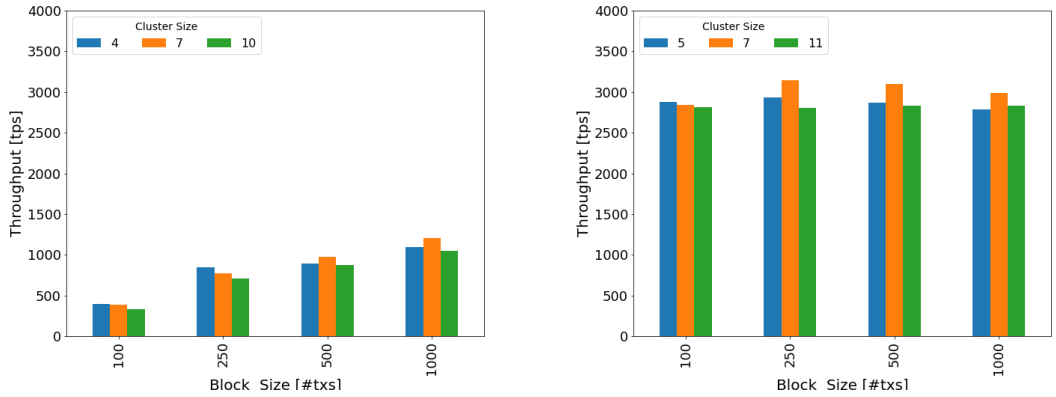
We measure the time it takes to perform both CPU computations and disk I/O, analyze the results and present them in Figure 9-a. Additionally, we provide a normalized analysis of latency per transaction based on the number of transactions in a block, which shows that increasing block size leads to efficiency gains. Next, we elaborate on what each of the operations entails:

- **TotalProposalVerification**: The total time it takes for a follower to verify a pre-prepare message sent from the leader which entails transforming the proposal to Fabric representation, then performing transaction verification, and verifying the hash chain is built properly.
- **TxnVerification**: This involves verifying the signature of the client on the transaction. This is needed in order to prevent the leader including forged transactions in the Blockchain which even if will be invalidated by peers, needlessly extend the blockchain, take up storage space and pose a performance overhead. In our implementation, we allocate a single goroutine per transaction, and they are executed in parallel based on the Go runtime's scheduling.
- **ProposalToBlock**: Being a general purpose consensus library, we delegate all validation to the application layer. Therefore, Fabric needs to transform the proposal to its own representation and this mainly involves protocol buffer serialization.
- **PrePrepareWAL**: The time it takes to persist the proposal to the WAL. This is needed for crash fault tolerance, and is done in CFT consensus as well. The time it takes to write the prepare and commit messages is not shown because it is negligible.
- **HashChainVerification**: Given a proposal which comprises the candidate to be the next block of the chain, we check that the header indeed points to the hash of the previous block, and that the data hash of the block derived from the proposal matches the computed data hash. As seen in Figure 9, the hash chain verification is linear in the size of the block. That is because the block hash in Fabric is a cumulative hash, and not a Merkle tree one, therefore it cannot be parallelized.
- **Deliver**: When a a quorum of signatures are collected for a proposal, our library hands over the proposal and its corresponding signatures to Fabric, where the block is appended to the ledger, and the indices that locate the block inside the block storage are updated. Interestingly, this takes much more time than writing the pre-prepare to the WAL, which is probably because it involves random writes and not sequential ones as in the WAL case.
- **CommitCollectVerify**: This involves the time to collect a quorum of signed commit messages as well as to validate the signatures.



(a) BFT (b) Raft

Figure 7: Cluster throughput as a function of block size, in a LAN, for BFT and Raft. Transaction size is 3993B.



(a) BFT (b) Raft

Figure 8: Cluster throughput as a function of block size, in a WAN, for BFT and Raft. Transaction size is 3993B.

Conclusions and possible optimizations: In local area networks, our throughput is effected mostly by time spent in local operations such as CPU computation and disk I/O. Several improvements might be considered:

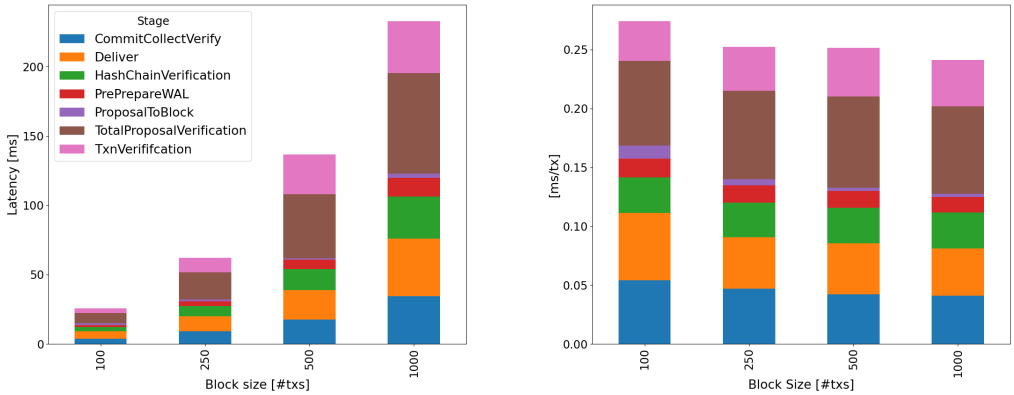
(1) **Parallel data hash computation**: If we replace the Fabric block data hash to a Merkle tree, one can split the Merkle tree to a k parts equal in size and compute each part in parallel, after which $log2\,(k)$ layers would be left which can be computed serially. Given n transactions, such a mechanism would compute $2^{log2(n)-log2(k)}$ of the Merkle tree in $\frac{1}{k}$ of the time.

(2) **Asynchronous commit**: As can be seen in Figure 9, the time required for the application to commit the block (*Deliver*) is much larger than the time it takes to persist it to the Write Ahead Log (WAL). Therefore, we can instead just write the commit messages to the WAL and let the application perform the commit asynchronously.

7 RELATED WORK

7.1 Byzantine fault tolerance

The ability to maintain and keep a consistent and immutable sequence of transactions shared between mutually distrustful parties is one of the keystones of blockchain platforms. In this work, this is achieved by utilizing the principals of the Byzantine fault tolerant



(a) Latency of local operations

(b) Normalized latency of local operations

Figure 9: The cost of each stage in the BFT consensus protocol, in LAN: (a) The latency of each stage, as a function of block size and cluster size (left). (b) The normalised latency of each stage (latency / number of TXs in block) for a 10 node cluster.

State Machine Replication (SMR) [54], which is equivalent to reaching agreement among distributed replica sets in light of possible Byzantine failures [52]. SMR relies on the ability to reach consensus on the total order of replicated commands [37, 55]. Therefore, consensus is usually used as a building block while implementing SMR protocols. The first formal definition of BFT consensus was stated by Lamport et al. [39]. The first protocol to attend the problem assuming synchronous network communication was suggested by Pease et al. [50], improved shortly after by Dolev and Strong [26] providing an optimal message complexity pattern [25].

While trying to provide a solution for Byzantine consensus in an asynchronous setting, it was proven to be impossible to devise deterministic algorithms in presence of even a single failure [29]. Several approaches were suggested to overcome this impossibility result: a) introducing partial synchrony assumptions [24, 27], b) relying on randomization [12, 51, 59], and c) failure detectors [8, 18].

Dwork et al. was the first to introduce the partial synchrony network model, where the system maintains safety during asynchronous periods, and resumes liveness after a point in time in which the system reassumes synchrony. The Practical Byzantine Fault Tolerant (PBFT) algorithm developed by Castro and Liskov [21] was a pioneering practical solution under the partial synchrony model. PBFT was the first consensus protocol to solve SMR in presence of Byzantine failures, laying the foundation to extensive research work [6, 23, 34]. To a large extent, PBFT was inspired by the Paxos [38] protocol, consisting of three sub-protocols: 1) normal operation, 2) checkpoint, and 3) view change. The normal operation requires $O(n^2)$ messages to reach consent on certain value, whereas view change requires $O(n^3)$ messages. Many optimizations were suggested to improve PBFT, in particular, BFT-SMaRt [13] was designed to optimize communication complexity of replacing a faulty leader. Zyzzyva [33] introduced the idea of optimistic speculative execution; however, severe safety violations were found in it by Abraham et al. [7], later leading to the development of the SBFT [30] protocol, fixing these safety violations.

7.2 Blockchain and BFT

In general, the increasing interest in blockchain lead to the development of algorithms such as Tendermint [16], HotStuff [61] and SBFT [30], where the key focus is on improving the view change sub-protocol or replacement of a faulty leader.

Many decades of research lead to BFT protocols exploiting randomization to provide consensus solution in the asynchronous model, with well known classical results such as [12, 14, 19, 59]. However, these protocols are far from being practical due to their poor performance. Only recently Miller et al. [48] suggested a leaderless randomized algorithm with reasonable and promising performance results.

Despite the renaissance of research around BFT consensus algorithms, triggered primarily by increased interest in blockchain, there are only a few openly available implementations suitable for production-grade exploitation [40–42]. Unfortunately, existing libraries lack a practical reusable interface which is flexible and generic enough to implement a BFT-enabled ordering service for Hyperledger Fabric. Any implementation based on a programming language other than Go would have exhibited the same drawbacks as the Kafka based ordering service architecture, where the consensus component will have been deployed as an external service. In the quest to develop a clustered ordering service based on an embedded consensus library, we had to write our own library.

7.3 BFT-Smart and Hyperledger Fabric

In its first release, the Fabric ordering service was based on the Kafka messaging service. In this implementation, the OSNs receive transactions from clients, and submit them to a Kafka topic, one topic for each Fabric channel. All OSNs would consume from the topics – again, one topic per channel – and therefore receive a totally ordered stream of transactions per channel. Each OSN would then deterministically cut the transaction stream into blocks.

In 2018 Sousa et al. [58] made an attempt to convert this implementation into a BFT ordering service. They replaced the Kafka service with a cluster of BFT-Smart based servers, where each server consisted of a BFT-Smart core wrapped with a thin layer that allowed it to somewhat "understand" Fabric transactions. The proof of concept presented in the paper exhibited promising performance, but was eventually not adopted by the community. The community discussions reveal a mix of fundamental and technical reasons for that [4].

The solution presented was composed of two processes (orderer front-end and BFT server), written in two languages (Go & Java, resp.). That was not well received as it complicates the development, maintenance, and deployment of the ordering service. The experience gained with the Kafka-based service motivated the community to move towards a single process that embeds a consensus library, as eventually happened with the introduction of the Raft-based ordering service in Fabric v1.4.1.

There were, however, more fundamental reasons. The code that wrapped the BFT-Smart core did not include Fabric's membership service provider (the MSP), which defines identities, organizations, and certificate authorities (CAs), and includes the cryptographic tools for the validation of signatures. Therefore, the BFT cluster signatures where not compliant with Fabric's, and the identities of the BFT cluster servers were not part of Fabric's configuration. In fabric, configuration is part of the blockchain (see Figure 1) and must be agreed upon. Incorporating the Java BFT cluster endpoints and identities (certificates & CAs) into the configuration flow would have meant providing a Java implementation to an already highly sensitive component. This shortcoming also meant that the front-end orderers had to collect $2F + 1$ signed messages from the BFT cluster servers, increasing the number of communication rounds to four.

The blocks produced by the front-end servers included only the signature of a single front-end orderer. This does not allow an observer of the blockchain (or a peer) to be convinced that the block was properly generated by a BFT service. Moreover, even if the $2F + 1$ BFT cluster signatures were included in the block metadata, that does not help an observer, as the identities of said servers are not included in Fabric's configuration. Moreover, peers and clients did not have the policies that correspond to the BFT service they consumed from.

Another subtle problem with a monolithic BFT cluster is that it does not allow a follower to properly validate the transactions proposed by the leader against the semantics of Fabric – again – without pulling in a significant amount of Fabric's code.

BFT-Smart owns much of its performance to the internal batching of requests. However, those batches are not consistent with the blocks of Fabric, so had to be un-packed, multiplexed into different channels, and then re-packed into Fabric blocks.

These problems were in front of us when we designed the library and its integration with Fabric. The interface of the library allowed us to seamlessly integrate with Fabric's MSP and configuration flow. Our implementation allows the leader to assemble blocks according to fabric's rules, so transactions are batched once. It allows followers to validate the transactions against Fabric's semantics during the consensus protocol, and it collects the quorum signatures during the commit phase. This reduces the number of communication rounds to three, and allows the observer of a single block to validate the correctness of the BFT protocol. The corresponding block validation policy was added to the peer and orderer for that affect, and the SDK client was augmented to interact properly with a BFT service.

7.4 Tendermint and Hyperledger Fabric

Another option we considered was to re-use the Tendermint Core [42] as an embedded consensus library. There is an Application Blockchain Interface (ABCI) which defines an API between the BFT protocol and SMR (application). However, the consensus protocol itself actually implements the blockchain, with batches of transactions chained together forming the ledger. The ABCI only provides an interface for the application to validate and execute transaction. That implies that using Tendermint as a consensus library in the Fabric ordering service node would have resulted in a ledger with Tendermint blocks in addition to the ledger maintained by Fabric.

Tendermint implements a peer-to-peer communication protocol inspired by the station-to-station protocol, where each network peer uses ED25519 key pairs as a persistent long term node ID. When peers establish a peer-to-peer connection they exchange generated X25519 ephemeral session keys, which are used to produced a shared secret similar to Diffie Hellman key exchange [47]. After the shared secret is generated it is used for symmetric encryption to secure the communication medium. However, there are two major caveats:

(1) This system is still vulnerable to a Man-In-The-Middle attack if the persistent public key of the remote node is not known in advance.
(2) This significantly deviates from the communication model of the Fabric and current implementation is too coupled into the Tendermint Core, hence would have required substantial refactoring.

Overall, the lack in configuration flexibility and the inability to replace the communication layer lead us towards the decision to implement our own BFT library instead.

8 CONCLUSION

In this paper we described the design, implementation, and evaluation of a BFT ordering service for Hyperledger Fabric. In the heart of this implementation lies a new consensus library, based on the BFT-Smart protocol, and written in Go. The library presents a new API suited for permissioned blockchain applications, such as Fabric. It delegates many of the core functions that any such library must use to the application employing it, allowing for maximal flexibility and generality. For example, cryptographic functions, identity management, as well as point to point communication are not embedded but are exposed through proper interfaces, to be implemented by the application using it. This allowed us to re-use some of the sophisticated mechanisms that Fabric already possessed. In the quest to make Fabric a truly end-to-end BFT system, it is not enough to augment the ordering service alone. We took special care to ensure that the peer and the client SDK interact properly with the BFT ordering service.

We chose to implement the BFT-Smart protocol because of its simplicity and elegance. This protocol is significantly simpler than

PBFT, because it does not allow for a transaction pipeline. In BFT-Smart there is only a single proposed transaction by a given leader at any point in time, which dramatically simplifies the view change sub-protocol. This simplicity greatly increases our confidence in the correctness of the implementation and reduced the effort it took to implement the library. However, these advantages come with a cost – reduced performance. This is especially salient when comparing against the highly mature and optimized etcd/raft library, which uses pipelining extensively. Despite the cost of reduced performance relative to etcd/raft, our implementation exhibits levels of performance that are sufficient for most permissioned blockchain applications: a 7 node BFT ordering service ($F = 2$) can support over 2500 TPS in a LAN, and over 1000 TPS in a WAN. These numbers are for a single channel; a Fabric network can scale horizontally by adding channels.

REFERENCES

[1] [n.d.]. Apache Kafka. https://kafka.apache.org.

[2] [n.d.]. Apache ZooKeeper. https://zookeeper.apache.org.

[3] [n.d.]. *etcd*: A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io and https://github.com/etcd-io/etcd.

[4] 2018. Regarding byzantine fault tolerance in Hyperleder Fabric. https://lists.hyperledger.org/g/fabric/topic/17549966#3135. A thread on the Hyperledger mailing list.

[5] 2020. The Raft Consensus Algorithm. https://raft.github.io.

[6] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. 2005. Fault-scalable Byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review* 39, 5 (2005), 59–74.

[7] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. 2017. Revisiting fast practical byzantine fault tolerance. *arXiv preprint arXiv:1712.01367* (2017).

[8] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. 2000. Failure detection and consensus in the crash-recovery model. *Distributed computing* 13, 2 (2000), 99–125.

[9] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. ACM, New York, NY, USA, Article 30, 15 pages. https://doi.org/10.1145/3190508.3190538

[10] E. Androulaki, A. De Caro, M. Neugschwandtner, and A. Sorniotti. 2019. Endorsement in Hyperledger Fabric. In *2019 IEEE International Conference on Blockchain (Blockchain)*. 510–519.

[11] A. Asayag, G. Cohen, I. Grayevsky, M. Leshkowitz, O. Rottenstreich, R. Tamari, and D. Yakira. 2018. A Fair Consensus Protocol for Transaction Ordering. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. 55–65.

[12] Michael Ben-Or. 1983. Another advantage of free choice (Extended Abstract) Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*. 27–30.

[13] A. Bessani, J. Sousa, and E. E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMaRt. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 355–362. https://doi.org/10.1109/DSN.2014.43

[14] Gabriel Bracha. 1984. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*. 154–162.

[15] Richard Gendal Brown. 2018. The Corda Platform: An Introduction. (2018).

[16] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains.* Ph.D. Dissertation.

[17] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. *CoRR* abs/1807.04938 (2018). arXiv:1807.04938 http://arxiv.org/abs/1807.04938

[18] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*. Springer, 524–541.

[19] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology* 18, 3 (2005), 219–246.

[20] Christian Cachin and Marko Vukolic. 2017. Blockchain Consensus Protocols in the Wild. *CoRR* abs/1707.01873 (2017). arXiv:1707.01873 http://arxiv.org/abs/1707.01873

[21] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, USA, 173–186.

[22] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.* 20, 4 (Nov. 2002), 398–461. https://doi.org/10.1145/571637.571640

[23] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. 2006. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 177–190.

[24] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. 1987. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)* 34, 1 (1987), 77–97.

[25] Danny Dolev and Rüdiger Reischuk. 1985. Bounds on information exchange for Byzantine agreement. *Journal of the ACM (JACM)* 32, 1 (1985), 191–204.

[26] Danny Dolev and H Raymond Strong. 1982. Polynomial algorithms for multiple processor agreement. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*. 401–407.

[27] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[28] Fabric RFC Repository 2020. Channel participation API without a system channel. https://github.com/hyperledger/fabric-rfcs/blob/master/text/0000-channel-participation-api-without-system-channel.md.

[29] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.

[30] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2018. SBFT: a scalable decentralized trust infrastructure for blockchains. *arXiv preprint arXiv:1804.01626* (2018).

[31] Hyperledger Fabric Documentation 2020. The blockchain data structure. https://hyperledger-fabric.readthedocs.io/en/release-2.0/ledger/ledger.html#blockchain.

[32] Java SDK Library Repository 2019. The Java SDK Library Open-Source Repository (anonymized for blind review). https://github.com/xxx/fabric-sdk-java.

[33] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 45–58.

[34] Ramakrishna Kotla and Michael Dahlin. 2004. High throughput Byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004.* IEEE, 575–584.

[35] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[36] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[37] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.

[38] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

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

[40] Library Repository 1999. Source code, Practical Byzantine Fault Tolerance. http://www.pmg.csail.mit.edu/bft//.

[41] Library Repository 2016. The Honey Badger of BFT Protocols. https://github.com/initc3/HoneyBadgerBFT-Python/.

[42] Library Repository 2016. Tendermint Core: Byzantine-Fault Tolerant State Machines. https://github.com/tendermint/tendermint.

[43] Library Repository 2019. Hyperledger Fabric BFT Open-Source Repository (anonymized for blind review). https://github.com/xxx/fabric.

[44] Library Repository 2019. The Library Open-Source Repository (anonymized for blind review). https://github.com/xxx/consensus.

[45] Y Manevich, A Barger, and Y Tock. 2019. Endorsement in Hyperledger Fabric via service discovery. *IBM Journal of Research and Development* 63, 2/3 (2019), 2:1–2:9.

[46] David Mazières. 2015. The Stellar Consensus Protocol : A Federated Model for Internet-level Consensus. (2015).

[47] Ralph C Merkle. 1978. Secure communications over insecure channels. *Commun. ACM* 21, 4 (1978), 294–299.

[48] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/

2976749.2978399

[49] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[50] Marshall Pease, Robert Shostak, and Leslie Lamport. 1980. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)* 27, 2 (1980), 228–234.

[51] Michael O Rabin. 1983. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. IEEE, 403–409.

[52] Michael K Reiter. 1995. The Rampart toolkit for building high-integrity services. In *Theory and practice in distributed systems*. Springer, 99–110.

[53] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. 459–474.

[54] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[55] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[56] 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 *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 105–122. https://doi.org/10.1145/3299869.3319883

[57] J. Sousa and A. Bessani. 2012. From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation. In *2012 Ninth European Dependable Computing Conference*. 37–48. https://doi.org/10.1109/EDCC.2012.32

[58] J. Sousa, A. Bessani, and M. Vukolic. 2018. A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 51–58. https://doi.org/10.1109/DSN.2018.00018

[59] Sam Toueg. 1984. Randomized byzantine agreements. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*. 163–178.

[60] Nicolas Van Saberhagen. 2013. CryptoNote v 2.0.

[61] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.