A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform

João Sousa and Alysson Bessani
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

Marko Vukolić
IBM Research Zurich, Switzerland

***Abstract*—Hyperledger Fabric is a flexible operating system for permissioned blockchains designed for business applications beyond the basic digital coin addressed by Bitcoin and other existing networks. A key property of this system is its extensibility, and in particular the support for multiple ordering services for building the blockchain. However, version 1 was launched in 2017 without an implementation of a Byzantine fault-tolerant (BFT) ordering service. To overcome this limitation, we designed, implemented, and evaluated a BFT ordering service for this system on top of the BFT-SMaRt state machine replication/consensus library, with optimizations for wide-area deployment. Our results show that our ordering service can process up to ten thousand transactions per second and write a transaction irrevocably in the blockchain in half a second, even with peers spread across different continents.**

I. Introduction

The impressive growth of Bitcoin and other blockchain platforms based on the Proof-of-Work (PoW) technique made evident the limitations of this approach. These limitations are mostly related to performance: existing systems are capable of processing from 7 (Bitcoin) to 10s-100s transactions per second and present transaction confirmation latencies of up to one hour [1]. Several alternative blockchain platforms proposed in the last years try to avoid these limitations by employing traditional Byzantine Fault-Tolerant (BFT) consensus protocols (e.g., [2]) for establishing consensus on the order of blocks [3].

Hyperledger Fabric (or simply, Fabric) is a system for deploying and operating permissioned blockchains that targets business applications [4]. It is built with flexibility and generality as key design concerns, supporting thus a wide variety of non-deterministic smart contracts (here called chaincodes) and pluggable services. The support for pluggable components gives Fabric an unprecedented level of extensibility and, in particular, enables it to use multiple ordering services for managing the blockchain. Despite this, version 1.0 (launched in June 2017) comes without any Byzantine fault-tolerant (BFT) ordering service implementation, providing only a crash fault-tolerant ordering service.

In this paper, we describe our efforts in overcoming this limitation, by presenting the design, implementation, and evaluation of a new *BFT ordering service* for Fabric v1.[1] This service is based on the well-know BFT-SMaRt state machine replication/consensus library [5], and its extension for WANs [6]. Our evaluation, conducted both on a local cluster and in a geo-distributed setting, shows that BFT-SMaRt ordering service can achieve up to 10k representative transactions per second and write a transaction irrevocably in the blockchain in half a second, even with ordering nodes spread through different continents.

Besides presenting our BFT ordering service, this paper also discusses the key concerns that need to be addressed to apply existing BFT state machine replication protocols to blockchain platforms and systems like Fabric. The huge interest of industry in permissioned blockchains has reinvigorate BFT research (e.g., [7]), and spawned many efforts to integrate (new or existing) BFT protocols in blockchain platforms (see [3] for a survey). Nonetheless, to the best of our knowledge, there are still no other works discussing a *practical* integration of a classical state machine replication library with a blockchain platform. In particular, we detail the service model and workload of interest in this kind of systems, which are substantially different from the microbenchmarks [2] and the Zookeeper-like client-server model [8] still used to evaluate BFT protocols.

The rest of this paper is organized as follows. We start by presenting the fundamentals of blockchain technology (Section II) and Hyperdeger Fabric (Section III). After that, the BFT-SMaRt and WHEAT protocols are briefly described (Section IV), and we proceed to present the BFT-SMaRt ordering service (Section V) and its experimental evaluation (Section VI). We propose some improvements to Fabric in Section VII, discuss some related work in Sections VIII and conclude the paper in Section IX.

II. Blockchain Technology

A blockchain is an open database that maintains a distributed ledger typically deployed within a peer-to-peer network. It is comprised by a continuously growing list of records called *blocks* that contain transactions [9]. Blocks are protected from tampering by cryptographic hashes and a consensus mechanism.

The structure of a blockchain – illustrated in Figure 1 – consists of a sequence of blocks in which each one contains the cryptographic hash of the previous block in the chain. This introduces the property that block j cannot be forged without also forging all subsequent blocks $j + 1...i$. Furthermore, the consensus mechanism is used to (1) prevent the whole chain from being modified; and to (2) decide which block is to be appended to the ledger.

[1] Source code available at https://github.com/jcs47/hyperledger-bftsmart.

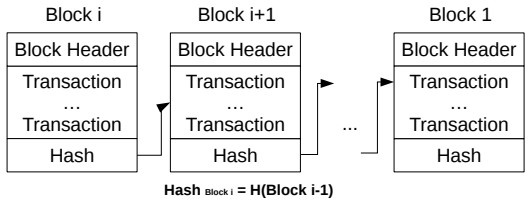


Fig. 1: Blockchain structure.

The blockchain may abide by either the *permissionless* or *permissioned* models [1]. Permissionless ledgers are maintained across peer-to-peer networks in a totally decentralized and anonymous manner [9], [10]. In order to determine which block to append to the ledger next, peers need to execute a Proof-of-Work (PoW) consensus [11]. The key idea behind PoW consensus is to limit the rate of new blocks by solving a cryptographic puzzle, i.e., execute a CPU intensive computation that takes time to solve, but can be verified quickly. This is achieved by forcing peers to find a nonce N such that given their block B and a limit L, the cryptographic hash of $B||N$ is lower than L [12], [13]. The first peer that presents such solution gets its block appended to the ledger. Roughly speaking, as long as the adversary controls less than half of the total computing power present in the network, PoW consensus prevents the adversary from creating new blocks faster than honest participants.

Permissionless blockchains have the benefit of enabling the ledger to be managed in a completely open way, i.e., any peer willing to hold a copy of the ledger can try to create new blocks for it. On the other hand, the computational effort associated to PoW consensus is both energy- and time-consuming; even if specialized hardware is used to find a Proof-of-Work, this mechanism still imposes a limit on transaction latency.

By contrast, permissioned blockchains employ a closed consortium of nodes tasked with creating new blocks and executing a traditional Byzantine consensus protocol to decide the order by which the blocks are inserted to the ledger [3], [14], [15]. Hence, permissioned blockchains do not expend the amount of resources that open blockchains do and are able to reach better transaction latency and throughput. In addition, it makes possible to control the set of participants tasked with maintaining the ledger – rendering this type of blockchain a more attractive solution for larger corporations, since it can be separated from the dark web or illegal activities.

III. Hyperledger Fabric

Hyperledger Fabric (Fabric) [4] is an open-source project within the Hyperledger collaborative effort.[2] It is a modular permissioned blockchain system designed to support pluggable implementations of different components, such as the ordering and membership services. Fabric enables clients to manage transactions by using chaincodes, endorsing peers and an ordering service.

[2]https://www.hyperledger.org/

Chaincode is Fabric's counterpart for smart contracts [16]. It consists of code deployed on the Fabric's network, where it is executed and validated by the endorsing peers, who maintain the ledger, the state of a database (modeled as a versioned key/value store), and abide by endorsement policies. The ordering service is responsible for creating blocks for the distributed ledger, as well as the order by which each blocks is appended to the ledger.

a) Fabric protocol: The Fabric general transaction processing protocol [4] – depicted in Figure 2 – works as follows:

1) *Clients create a transaction and send it to endorsing peers*. This message is a signed request to invoke a chaincode function. It must include the chaincode ID, timestamp and the transaction's payload.
2) *Endorsing peers simulate transactions and produce an endorsement signature*. They must verify if the client is properly authorized to perform the transaction by evaluating access control policies of a chaincode. Transactions are then executed against the current state. Peers transmit to the client the result of this execution (read and write sets associated to their current state) alongside the endorsing peer's signature. No updates are made to the ledger at this point.
3) *Clients collect and assemble endorsements into a transaction.* The client verifies the endorsing peers signatures, determine if the responses have the matching read/write set and checks if the endorsement policies has been fulfilled. If these conditions are met, the client creates a signed *envelope* with the peers' read and write sets, signatures and the Channel ID. A channel is a private blockchain on a Fabric network, providing data partition. Each peers of the channel share a channel-specific ledger. The aforementioned envelope represents a *transaction proposal*.
4) *Clients broadcast the transaction proposal to the ordering service.* The ordering service does not read the contents of the envelope; it only gathers envelopes from all channels in the network, orders them using atomic broadcast, and creates signed chain blocks containing these envelopes.
5) *The blocks of envelopes are delivered to the peers on the channel.* The envelopes within the block are again validated to (1) ensure the endorsement policies were fulfilled, and (2) to check if there were changes to the peers' state for read set variables (since the read set was generated by the transaction execution). To this end, the read set contains a set of versioned keys that endorsing peers read at the time of simulating a transaction (step 2). Depending on the success of these validations, the transaction proposal contained in envelopes are marked as either being valid or invalid.
6) *Peers append the received block to the channel's blockchain.* For each valid transaction, the write sets are committed to the peers' current state. An event is triggered to notify the client that the transaction has been immutably appended to the channel's blockchain,

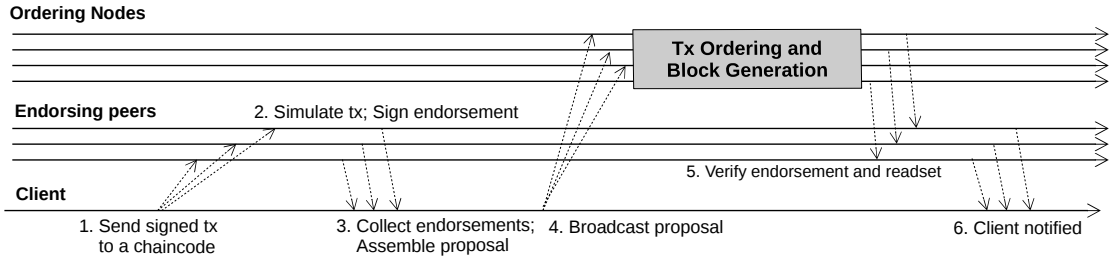


Fig. 2: Hyperledger Fabric transaction processing protocol [4].

as well as notification of whether the transaction were deemed valid or invalid. Notice that invalid transactions are also added to the ledger, but they are not executed at the peers. This also has the added benefit of making it possible to identify malicious clients, since their actions are also recorded.

An important aspect of the Fabric protocol is that endorsement (step 2) and validation (step 5) can be done at different peers. Furthermore, contrary to the chaincode execution during endorsement, the validation code needs to be deterministic, i.e., the same transaction validated by different peers in the same state produces the same output [4].

b) Pluggable consensus: As mentioned before, Fabric is a modular blockchain system. In particular, one of the components that support plug-and-play capability is the ordering service. Currently, Fabric's codebase includes the following ordering service modules: (1) a centralized, non-replicated ordering service that does not execute any distributed protocol that is used mostly for testing the system; and (2) a replicated ordering service capable of withstanding crash faults, consisting of an Apache Kafka cluster[3] and its respective ZooKeeper ensemble [8]. At the time of this writing, both modules have limitations. The non-replicated module requires very few hardware resources, but it is also a single point of failure. The Kafka-based module is both decentralized and robust, but can only withstand crash faults.

IV. BFT-SMaRt & WHEAT

The ordering service presented in this paper was designed on top of existing BFT systems, namely BFT-SMaRt [5] and WHEAT [6]. In this section we present a brief description of these works.

BFT-SMaRt implements a modular state machine replication protocol on top of a Byzantine consensus algorithm [17]. Under favourable network conditions and the absence of faulty replicas, BFT-SMaRt executes the message pattern depicted in Figure 3, which is similar to the PBFT protocol [2].

Clients send their requests to all replicas, triggering the execution of the consensus protocol. Each consensus instance i begins with one replica – the *leader* – proposing a batch of requests to be decided within that consensus. This is done by sending a PROPOSE message containing the aforementioned batch to the other replicas. All replicas that receive the PROPOSE message verify if its sender is the leader and if the batch proposed is valid. If this is the case, they register the batch being proposed and send a WRITE message to all other replicas containing a cryptographic hash of the proposed batch. If a replica receives $\lceil \frac{n+f+1}{2} \rceil$ WRITE messages with the same hash, it sends an ACCEPT message to all other replicas containing this hash. If some replica receives $\lceil \frac{n+f+1}{2} \rceil$ ACCEPT messages for the same hash, it deliver its correspondent batch as the decision for its respective consensus instance.

The message pattern just described is executed if the leader is correct and the system is synchronous. If these conditions do not hold, the protocol needs to elect a new leader and force all replicas to converge to the same consensus execution. This procedure is described in detail in [17].

Our ordering service also employs WHEAT, a variant of BFT-SMaRt optimized for geo-replicated environments. It differs from the aforementioned protocol in the following way: it employs the tentative executions proposed by Castro and Liskov [2] and uses a vote assignment scheme for efficient quorum usage [6]. The vote assignment scheme integrates classical ideas from weighted replication [18] to state machine replication protocols. The idea is to build small quorums with fastest replicas without endangering the safety and liveness of the underlying consensus protocol. This mechanism improves latency by allowing more choice: if there is a spare replica in the system that is faster than the rest, the optimal quorum will contain this replica. It works by being given parameters f (number of assumed faults) and Δ (amount of extra replicas), then based on this input, compute values V_{max} and u. V_{max} is the weight value to be given to the u fastest replicas in the system. All other $n - u$ replicas are given value V_{min}. For instance, when using five replicas ($f = 1$ and $\Delta = 1$), two of them will have weight $V_{max} = 2$ and the remaining three will have $V_{min} = 1$.

V. BFT-SMaRt Ordering Service

The BFT-SMaRt module for Fabric's ordering service consists of an ordering cluster and a set of frontends. The ordering cluster is composed by a set of $3f + 1$ nodes that collect envelopes from the frontends and execute the BFT-SMaRt's replication protocol with the purpose of totally ordering these envelopes among them. Once a node gathers a predetermined number of envelopes, it creates a new block containing these envelopes and a hash of the previously created block, generates a digital signature for the block, and disseminates it to all

[3]https://kafka.apache.org/

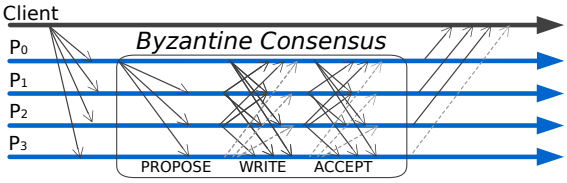


Fig. 3: BFT-SMaRt message pattern.

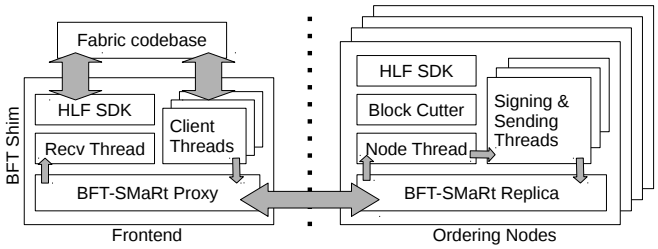


Fig. 4: BFT-SMaRt ordering service architecture.

known frontends, which collect $2f + 1$ matching blocks from ordering nodes. The $2f + 1$ blocks are necessary because frontends do not verify signatures. However, this number guarantees a minimum of $f + 1$ valid signatures to peers and clients.[4] Frontends are part of the peer trust domain and are responsible for (1) relaying the envelope to the ordering cluster on behalf of the client, and (2) receiving the blocks generated by the ordering cluster and relaying them to the peers responsible for maintaining the distributed ledger.

a) Architecture: BFT-SMaRt's ordering service architecture is illustrated in Figure 4. The frontend is composed by the Fabric codebase and a BFT shim. The Fabric codebase (implemented in Go) provides an interface for Fabric clients to submit envelopes. These envelopes are relayed to the BFT shim using UNIX sockets. This shim is implemented in Java and maintains (1) a client thread pool that receive envelopes and relays them to the ordering cluster, and (2) a receiver thread that collects blocks from the cluster. Envelopes (resp. blocks) are sent to (resp. received from) the cluster through the BFT-SMaRt proxy. The proxy does that by issuing an asynchronous invocation request to the BFT-SMaRt client-side library, ensuring it does not block waiting for replies. To ensure that the shim performs computations on equivalent data structures to the Fabric codebase, the ordering service uses the Hyperledger Fabric Java SDK to parse and assemble data structures used in Fabric.

b) Batching: The ordering nodes are implemented on top of the BFT-SMaRt service replica, thus receiving a stream of totally ordered envelopes. Each node maintains an object named *blockcutter*, where the envelopes received from the service replica are stored before being assembled into a block. The blockcutter is responsible for managing the envelopes associated to each Fabric channel and creating a batch of envelopes to be included in a block for the ledger associated to that channel. We implement this batching mechanism instead of relying on BFT-SMaRt's native batching because (1) each BFT-SMaRt's batch may contain envelopes that are not associated to the same channel, which means the envelopes cannot be all assembled into the same block; (2) Fabric supports *configuration envelopes*, which are supposed to remain isolated from regular envelopes; and (3) Fabric's native batching policies are not equivalent to BFT-SMaRt's (for instance, Fabric imposes a batching limit based on its size in terms of bytes, whereas BFT-SMaRt limit is based on number of requests per batch). Once the blockcutter holds a pre-determined number of envelopes for a channel (the block size), it notifies the node thread that it is time to drain its envelopes and create the next block.

c) Parallelization: After the blockcutter is drained, a sequence number is assigned to the future block and submitted to the signing/sending thread pool alongside with the respective block header. This header contains the aforementioned sequence number and the cryptographic hashes from the previous header and the hash for the block's envelopes. Notice that this thread pool does not cause non-determinism across the nodes because (1) the block header and envelopes to be assigned to new blocks are generated sequentially within the node thread, and (2) the only structures that each node needs to maintain as part of the application state is the block header from the previous iteration of the node thread. Similarly to the frontend, the Fabric Java SDK is used to correctly handle and create the data structures used by the system. In addition, this SDK is also used to generate cryptographic hashes and ECDSA (Elliptic Curve DSA) signatures [19] that can be validated by other components of Fabric. Once the block is created and signed, it is transmitted to all active frontends. This is done through a custom *replier* (supported by the extensible API of BFT-SMaRt) that, instead of sending the operation result (i.e., the generated block) to the invoking client, sends it to a set of registered BFT-SMaRt clients (i.e., the frontends).

d) Durability and Node Membership: Besides the transaction ordering and execution, the BFT-SMaRt replica also provides additional capabilities that are fundamental for practical state machine replication, such as durability (of state, in case all ordering nodes fail) and reconfiguration of the group of ordering nodes. The state is comprised by the headers for the last block associated to each channel, information about the current configuration of channels, and the envelopes currently stored at the blockcutter. Since the headers have a constant size and the envelopes are periodically drained from the blockcutter, the state maintained at the ordering nodes will always be bounded and remain smaller than the size of the ledger maintained by Fabric peers.

e) Validation and Reconfiguration: One last aspect of this service relates to channel reconfiguration and transaction validation. Fabric's architecture is resilient to blocks contained junk transactions, hence ordering services can avoid performing transaction validation. In the particular case of our ordering service, transactions can be validated by the signing/sending threads prior to generating block signatures. Transactions can

[4] If the frontends are programmed to perform signature verification, only $f + 1$ matching blocks suffice.

then be removed from the block if the validation fails. The exception to this is a special category of transactions that are used to perform channel reconfiguration. These transactions need to be validated and executed prior to submitting them to a blockcutter.

VI. Evaluation

In this section we describe the experiments conducted to evaluate BFT-SMaRt's ordering service and discuss the observed results. Our aim here is not to evaluate the whole Fabric system, but only the ordering service, which may typically be the bottleneck of the system.

A. Parameters affecting the Ordering Performance

The throughput of the ordering service (i.e., the rate at which envelopes are added to the blockchain TP_{os}) is bounded by one of three factors: a) the rate at which envelopes are ordered by BFT-SMaRt ($TP_{bftsmart}$) for a given envelope size, number of envelopes per block and number of receivers; b) the number of blocks signed per second (TP_{sign}); or c) the size of the generated blocks. These parameters are illustrated in Figure 5.

Given an envelope size es, block sizes bs, and a number of receivers r (i.e., the peer frontends to which the ordering nodes transmit the generated blocks), the peak throughput of the ordering service is bounded as follows:

$$TP_{os}^{bs,es,r} \leq min(TP_{sign} \times bs, TP_{bftsmart}^{bs,es,r}) \quad (1)$$

An important remark is that this equation considers that a block is signed only once by each ordering node, however, in Fabric 1.0 a block need to be signed twice. The second signature is needed to attach the block transaction to an execution context (details are out of the scope of this paper). If this is the case for the considered application, the TP_{sign} term used in the equation must be replaced by $\frac{TP_{sign}}{2}$.

B. Signature Generation

In order to estimate TP_{sign}, we run a very simple signature benchmark program written in Java in a Dell PowerEdge R410 server, which possesses two quad-core 2.27 GHz Intel Xeon E5520 processor with hyper-threading (thus having 16 hardware threads) and 32 GB of memory. The server runs Ubuntu 14.04 with JVM 1.8.0. Our program spawns a number of threads to create ECDSA signatures for blocks of fixed size and calculates how many of such signatures are generated per second.

Our results show that our server can generate up to 8.4k signatures/sec, when running with 16 threads. Furthermore, the effect of the block size is mostly negligible as the ECDSA signature is computed over the hash of the block. These results, together with the fact that a blocks are expected to contain 10+ envelopes in Fabric, lead us to conclude that signature generation is not expected to be a bottleneck in our setup.[5]

[5]For example, by using blocks with $bs = 100$ envelopes, we can sign up to $TP_{sign} \times bs = 840k$ envelopes/sec.

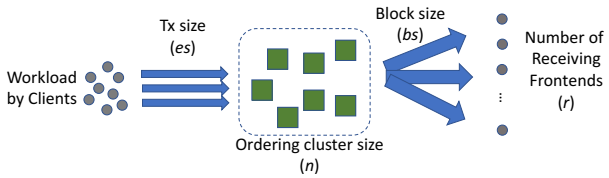


Fig. 5: Ordering service performance model.

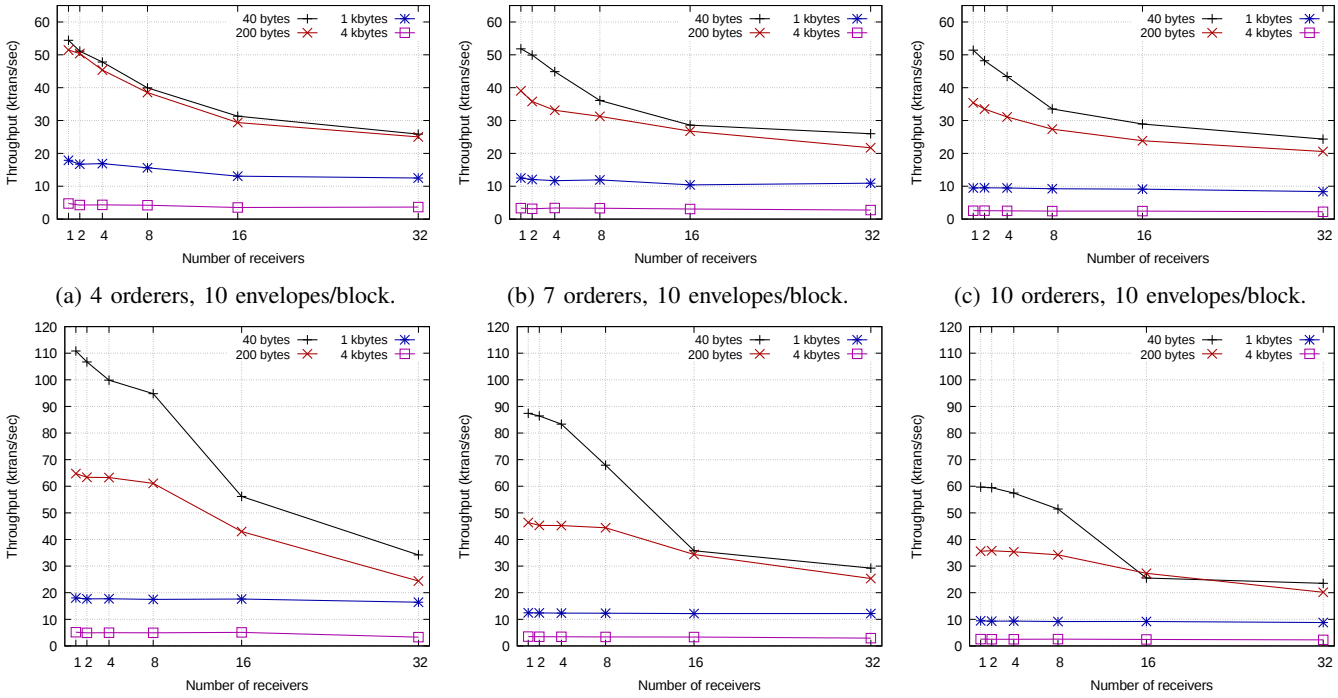
C. Ordering Cluster in a LAN

The experiments aims to evaluate the BFT-SMaRt ordering service by using clients that emulate the behavior of multiple ordering service frontends. They were executed with clusters of 4, 7, and 10 nodes, withstanding 1, 2, and 3 Byzantine faults, respectively. Furthermore, we also fiddled with the block size, by configuring each cluster configuration to assemble blocks containing either 10 or 100 envelopes (i.e., transactions). This is meant to observe the behaviour of each cluster when throughput is bound by either the rate of signature generation or by the rate of envelope reception. The environment is comprised by Dell PowerEdge R410 servers, like the one described before, connected through a Gigabit ethernet.

For each micro-benchmark configured to have x nodes and y envelopes/block, we gathered results for (1) envelopes with different sizes, and (2) a variable number of receivers. More precisely, each envelope size is representative of submitting to the ordering cluster: (1) a SHA-256 hash (40 bytes); (2) three ECDSA endorsement signatures (200 bytes); and (3) transaction messages of 1 and 4 kbytes. In practice, and considering the way Fabric 1.0 operates, the values related with (3) are more representative of the size of a transaction. In particular, our limited experience shows that transactions compressed with gzip tend to be usually close to 1 kbyte. Nonetheless, measurements for (1) and (2) are important to show the potential of the ordering service if different design choices were taken in future versions of Fabric.

Measurements for the throughput associated to block generation were gathered at ordering node 0 (the leader replica of BFT-SMaRt's replication protocol). To reach the system's peak throughput, each execution was performed using 16 to 32 clients distributed across 2 additional machines. We also repeated the micro-benchmark with 4 nodes with blocks of 100 envelopes. All experiments used 16 signing threads (to match the number of available cores) and were repeated 3 times taking 5 minutes each.

The obtained results for local-area are presented in Figure 6. Even though throughput drops when increasing the number of receivers, the impact of the number of receivers is considerably smaller for larger transactions (1k and 4 kbytes). This is because for these envelope sizes, the overhead of the replication protocol is greater than the overhead of transmitting blocks of 10 and 40 kbytes. In particular, since the batch limit of the BFT-SMaRt is set to 400 requests (default value), the PROPOSE message of the underlying replication protocol can have up to 0.4-1.6MBs with these envelope sizes.



(a) 4 orderers, 10 envelopes/block. (b) 7 orderers, 10 envelopes/block. (c) 10 orderers, 10 envelopes/block.

(d) 4 orderers, 100 envelopes/block. (e) 7 orderers, 100 envelopes/block. (f) 10 orderers, 100 envelopes/block.

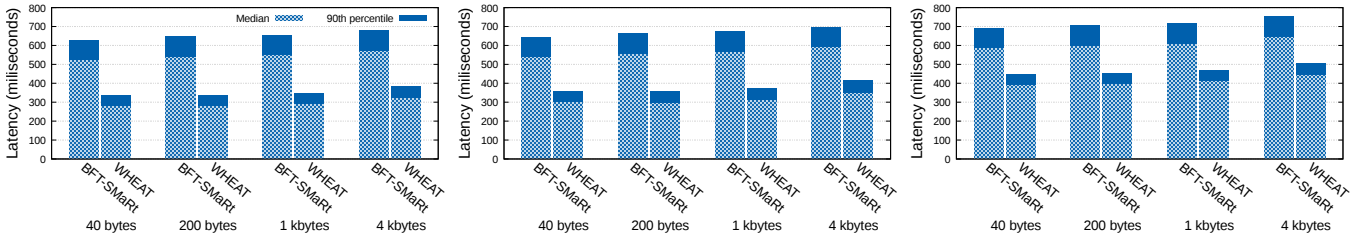
Fig. 6: BFT-SMART Ordering Service throughput for different envelope, block and cluster sizes.

It can be observed that when using 10 envelopes/block (Figures 6a, 6b, and 6c), the maximum throughput observed is approximately 50k transactions/second (when there exists only 1 to 2 receivers in the system), which is way below the $8.4k \times 10 = 84k$ envelopes/sec capacity of only signatures are considered (Section VI-B). This can be explained by the fact that signature generation needs to share CPU power with the replication protocol, hence creating a thug-of-war between the application's worker threads and BFT-SMART's I/O threads and queues – in particular, BFT-SMART alone can take up to 60% of CPU usage when executing a void service with asynchronous clients. Hence, the performance drops when compared to the micro-benchmark from Section VI-B, which was executed in a single machine, stripped of the overhead associated with BFT-SMART. Moreover, for up to 2 receivers and envelope sizes of 1 and 4 kbytes, the peak throughput becomes similar to the results observed in [5]. This is because for these request sizes BFT-SMART is unable to order envelopes at a rate equal to the rate at which the system is able to produce signatures.

Figures 6d, 6e, and 6f show the results obtained for 100 envelopes/block, when each node is not subject to CPU exhaustion. It can be observed that, across all cluster sizes, throughput is significantly higher for smaller envelope sizes and up to 8 receivers. This happens because even though each node creates blocks at a lower rate – approximately 1100 blocks per seconds – each block contains 100 envelopes instead of only 10. Moreover, this configuration makes the rate at which envelopes are ordered to become similar to the rate at which blocks are created. This means that for smaller envelope sizes, it is better to adjust the nodes' configuration to avoid consuming all the CPU time and rely on the rate of envelope arrival. However, for envelopes of 1 and 4 kbytes the behavior is similar to using 10 envelopes/block, specially from 7 nodes onward. This is because for larger envelope sizes – as discussed previously – the predominant overhead becomes the replication protocol. Interestingly, for a larger number of receivers (16 and 32), throughput converges to similar values across all combinations of envelope/cluster/block sizes. Whereas for larger envelope sizes this is due to the overhead of the replication protocol, for smaller envelope sizes this happens because the transmission of blocks to the receivers becomes the predominant overhead.

D. Geo-distributed Ordering Cluster

In addition to the aforementioned micro-benchmarks deployed in a local datacenter, we also conducted a geo-distributed experiment focused on collecting latency measurements at 3 frontends scattered across the Americas, with the nodes of the ordering service distributed all around the world: Oregon, Ireland, Sydney, and São Paulo (four BFT-SMART replicas), with Virginia standing as WHEAT's additional replica (five replicas). Since signatures generation requires considerable CPU power, we used instances of the type *m4.4xlarge*, with 16 virtual CPUs each. The frontends were deployed in Oregon (collocated with leader node weighting V_{max} in WHEAT), Virginia (collocated with non-leader node, but still weighting V_{max}) and São Paulo. Each frontend was



(a) Oregon (weighted V_{max}, leader node). (b) Virginia (weighted V_{max}). (c) São Paulo (weighted V_{min}).

Fig. 7: Amazon EC2 latency results (4 receivers, blocks with 100 envelopes).

configured to launch enough client threads to keep node throughput always above 1000 transactions/second.

Figure 7 presents the results for the geo-distributed micro-benchmark with a a block size of 100 envelopes. As expected, WHEAT's latency is consistently lower than BFT-SMaRt's across all frontends – always below 500 miliseconds and up to 45% less than BFT-SMaRt. It is worth pointing out that envelope size has a relatively modest impact on latency: across all regions, the difference between a 4k and a 400k bytes block was never above 61 milliseconds for any percentile or protocol. By contrast, the placement of the frontends when using WHEAT exhibited a larger impact on latency: the difference between Virginia (weighted V_{max}) and São Paulo (weighted V_{min}) is above 90 milliseconds. In addition, the difference between São Paulo and Oregon is even larger, in the order of 120 miliseconds.

VII. Possible Improvements on Fabric

In the following we list some improvements that could be made on the Fabric codebase to facilitate the implementation and improve the performance of ordering services.

The implementation of our ordering service produces a single signature per block, aimed at protecting its integrity. By contrast, as pointed out in Section VI-A, Fabric expects each block to contain two signatures associated with it. While one signature is meant to protect the integrity of its associated block, the other is intended to bind the block transactions to an execution context. However, this can lead to a significant performance penalty when the bounding factor is the rate of signature generation, i.e., small transactions with few frontends. By inspecting the code and talking to Fabric developers, we found no good reason to have this second signature, as it basically covers the regular payload plus the id of a block storing the last reconfiguration envelope. It appears that producing only this second signature suffices to protect the integrity of the blockchain.

To withstand malicious behavior from ordering nodes, each one locally assembles blocks and produces their respective signatures. This results in a stream of blocks that are appended to the local copy of the ledger that is maintained at the frontends. However, the Fabric codebase is better suited for crash-only ordering services such as Kafka, which generates a *stream of envelopes* rather than a *stream of blocks*. For instance, upon receiving a stream of envelopes, Fabric uses methods to both generate blocks and append them to the chain. Moreover, the methods that append the blocks to the chain also produce the signatures discussed previously. This is not only unnecessary in the case of our ordering service, but also does not provide any additional protection to the block in a scenario in which Byzantine faults are considered. This forced us to augment the Fabric with support for receiving pre-signed blocks and strictly appending them to the chain.

As mentioned in Section V, we use UNIX sockets to communicate between the Fabric process that receives transactions from clients (Go) and BFT-SMaRt's process that relays envelopes to ordering nodes (Java). This adds an overhead that could be avoided if we had a single Java process receiving envelopes directly from clients and relaying them to ordering nodes. While this overhead could potentially be mitigated with a proper Go wrapper for BFT-SMaRt,[6] we believe it would be worth to augment the Fabric Java SDK (that we also use) with support for reception and parsing of client requests.

Finally, due to BFT-SMaRt native support of view reconfiguration, our ordering service can fully support reconfiguration of the set of ordering nodes. However, this does not extend to the set of frontends, specifically at Fabric's Go process. In order to support reconfiguration on this set of nodes, Fabric needs to be augmented with the capacity to transfer the ledger between these Fabric processes.

VIII. Related Work

The concept of blockchain was originally introduced by Bitcoin to solve the double spending problem associated with crypto-currency in permissionless peer-to-peer networks [9]. Since Bitcoin's inception and widespread adoption, other platforms based on Proof-of-Work blockchain have emerged. Within these new platforms, Ethereum is particularly relevant for its support of smart contracts [10].

Because of the known performance penalty associated with Proof-of-Work creation and the fact that Blockchain technology is gaining the attention of many industries, the idea of permissioned blockchains are quickly gaining traction. Examples of other permissioned blockchain platforms include Chain, which uses the Federated Consensus algorithm [21]. Tendermint implements the BFT protocol designed by

[6]Such wrapper is already available for C++ and Python [20].

Buchman et. al. [14]. Kadena [15] uses a variant of the Raft consensus protocol [22] adapted to Byzantine faults [23]. Finally, Symbiont Assembly[7] uses a Go implementation of the Mod-SMaRt algorithm [17] and heavily follows the design of BFT-SMaRt. A recent survey [3] compares all these permissioned protocols and points BFT-SMaRt as a prominent candidate for implementing this type of ledgers.

Many services have been implemented on top of BFT-SMaRt over the years.[8] The one that most closely resembles the architecture of the ordering service presented here is SieveQ [24]. This system is a hybrid between a publish-subscribe service and an application-level firewall that also orders messages before sending them to targetted receivers. Among many differences, SieveQ focus on the robustness against DoS attacks and recovery of faulty replicas, while our service focuses on the specifics of block generation for Fabric.

IX. Conclusion

In this paper we described the design, implementation, and evaluation of a BFT ordering service for Hyperledger Fabric using the BFT-SMaRt replication library. Our experimental evaluation shows that peak throughput is bound either by the rate at which block signatures are generated by a replica, or the rate of envelopes ordered by the total order protocol. Moreover, the results also suggest that, for smaller envelope sizes, increasing the block size while decreasing the rate of signature generation can yield higher throughput than to simply rely on the maximum possible rate of signature generation. Nonetheless, for a higher number of repliers, throughput tends to converge to similar values across all micro-benchmarks. Even when transmitting blocks of 400 kbytes to 32 receivers in a cluster of 10 nodes, the ordering service still reaches a sustained throughput of approximately 2200 transactions/second – which is more than twice of Ethereum's theoretical peak of 1000 transactions/second [25], and vastly superior than Bitcoin's peak of 7 transaction/second [1]. Finally, latency measurements taken from a geo-replicated setting are also shown attractive, with values within half a second under moderate workload using WHEAT, even when accounting for large block sizes.

Acknowledgment

This work was supported by an IBM Faculty Award, by FCT through the LASIGE Research Unit (UID/CEC/00408/2013) and the IRCoC project (PTDC/EEI-SCR/6970/2014), and by the European Commission through the H2020 SUPERCLOUD project (643964).

References

[1] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication," in *Open Problems in Network Security - IFIP WG 11.4 International Workshop*, Zurich, Switzerland, 2015.

[2] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.

[3] C. Cachin and M. Vukolic, "Blockchain consensus protocol in the wild (invited paper)," in *Proceedings of 31th International Symposium on Distributed Computing*, Vienna, Austria, 2017.

[4] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the 13th ACM SIGOPS European Conference on Computer Systems*, Porto, Portugal, 2018.

[5] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with BFT-SMART," in *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks*, Atlanta, GA, USA, 2014.

[6] J. Sousa and A. Bessani, "Separating the WHEAT from the chaff: An empirical design for geo-replicated state machines," in *Proceedings of the IEEE 34th Symposium on Reliable Distributed Systems*, Montreal, Quebec, Canada, 2015.

[7] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, Vienna, Austria, 2016.

[8] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale services," in *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, USA, 2010.

[9] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://bitcoin.org/bitcoin.pdf

[10] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2015. [Online]. Available: http://gavwood.com/Paper.pdf

[11] J. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Sofia, Bulgaria, 2015.

[12] A. Back, "Hashcash - a denial of service counter-measure," 2002. [Online]. Available: http://www.hashcash.org/papers/hashcash.pdf

[13] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, London, UK, 1993.

[14] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Master's thesis, University of Guelph, 2016.

[15] W. Martino, "Kadena: The first scalable, high performance private blockchain," 2016. [Online]. Available: http://kadena.io/docs/Kadena-ConsensusWhitePaper-Aug2016.pdf

[16] N. Szabo, "Smart contracts: Building blocks for digital markets," *EXTROPY: The Journal of Transhumanist Thought*, no. 16, 1996.

[17] J. Sousa and A. Bessani, "From Byzantine consensus to BFT state machine replication: A latency-optimal transformation," in *Proceedings of the 9th European Dependable Computing Conference*, Sibiu, Romania, 2012.

[18] D. Gifford, "Weighted voting for replicated data," in *Proceedings of the 7th ACM SIGOPS Symposium on Operating Systems Principles*, Pacific Grove, CA, USA, 1979.

[19] D. B. Johnson and A. J. Menezes, "Elliptic curve DSA (ECSDA): An enhanced DSA," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, Berkeley, CA, USA, 1998.

[20] C. Y. da Silva Costa and E. A. P. Alchier, "Diversity on state machine replication," in *Proceedings of the 32nd IEEE International Conference on Advanced Information Networking and Applications*, Kraków, Poland, 2018.

[21] "Chain protocol whitepaper," 2014. [Online]. Available: https://chain.com/docs/1.2/protocol/papers/whitepaper

[22] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference*, Philadelphia, PA, USA, 2014.

[23] C. Copeland and H. Zhong, "Tangaroa: a Byzantine fault tolerant raft," 2014. [Online]. Available: http://www.scs.stanford.edu/14au-cs244b/labs/projects/copeland_zhong.pdf

[24] M. Garcia, N. Neves, and A. Bessani, "SieveQ: A layered BFT protection system for critical services," *IEEE Transactions on Dependable and Secure Computing (accepted for publication)*, 2016.

[25] V. Buterin, "Ethereum platform review: Opportunities and challenges for private and consortium blockchains," 2016. [Online]. Available: http://r3cev.com

[7] https://symbiont.io/technology/assembly/

[8] https://github.com/bft-smart/library/wiki/Used-in-and-by