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

João Sousa *Alysson Bessani* *Marko Vukolić*
LaSIGE, Faculdade de Cincias, Universidade de Lisboa *IBM Research Zurich*

Abstract

Hyperledger Fabric (HLF) is a flexible permissioned blockchain platform designed for business applications beyond the basic digital coin addressed by Bitcoin and other existing networks. A key property of HLF is its extensibility, and in particular the support for multiple ordering services for building the blockchain. Nonetheless, the version 1.0 was launched in early 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 HLF on top of the BFT-SMaRt state machine replication/consensus library, implementing also optimizations for wide-area deployment. Our results show that HLF with our ordering service can achieve up to ten thousand transactions per second and write a transaction irrevocably in the blockchain in half a second, even with peers spread in different continents.

1 Introduction

The impressive growth of Bitcoin and other blockchain platforms based on the Proof-of-Work (PoW) technique, made evident the performance limitations of this approach. These limitations are mostly related with perfomance: 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 [25]. Several alternative blockchain platforms proposed in the last years try to avoid these limitations by employing more traditional Byzantine Fault-Tolerant (BFT) consensus protocols (e.g., [10]) for establishing consensus on the blocks in a blockchain [8].

Hyperledger Fabric[1] (HLF) is a platform that target business applications. It is built with flexibility and generality as key design concerns, supporting thus a wide variety of non-deterministic smart contracts (here called chaincode) and pluggable services [26]. The support for pluggable components, gives the HLF an unprecedented level of extensibility, and in particular the support for multiple ordering services for writing transactions on the blockchain. Despite of that, the version 1.0 (launched in early 2017) comes without any Byzantine fault-tolerant (BFT) ordering service, supporting only crash tolerance through an ordering service based on Apache Kafka.[2]

In this paper, we describe our efforts in overcoming this limitation, by presenting the design, implementation, and evaluation of a BFT ordering service for HLF 1.0[3] based on the BFT-SMaRt state machine replication/consensus library [4], and its extensions to support low-latency consensus on the internet [23]. Our preliminary evaluation, both on a local cluster and in a geo-distributed setting, show that HLF with 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 consensus nodes spread through different continents. The source code of the our service is freely available on the internet for the HLF community.[4]

As an additional contribution, the paper also discuss the key concerns that need to be addressed to apply existing (BFT or not) state machine replication protocols to blockchain platforms like HLF, and the service model and workload of interest in this kind of systems, which are substantially different from the microbenchmarks [10] and the Zookeeper-like client-server model [16] 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 2) and Hyperdeger Fabric (Section 3). After that, the BFT-SMaRt and WHEAT protocols (Section 4) are briefly described, and we proceed to present the

[1] `https://www.hyperledger.org/projects/fabric`.

[2] `https://kafka.apache.org/`.

[3] PBFT implementation present in HLF v0.6 was deprecated with transition to the new v1 architecture [1].

[4] `https://github.com/jcs47/hyperledger-bftsmart`.

BFT-SMART ordering service (Section 5) and its experimental evaluation (Section 6). We discuss some related work in Section 7 and conclude this paper in Section 8.

2 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 [19]. 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 (2) decide which block to be appended to the ledger.

The blockchain may abide to either the *permissionless* or *permissioned* models [25]. Permissionless ledgers are maintained across peer-to-peer networks in a totally decentralized and anonymous manner[19, 6]. In order to determine which block to append to the ledger next, peers need to execute Proof-of-Work (PoW) consensus [13]. 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 achieve 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 [2, 12]. 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 curated completely anonymously; any peer willing to hold a copy of the ledger and create new blocks to it is able to do so. 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 process still exerts a limit on transaction latency.

By contrast, permissioned blockchains require a set of trusted 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 [17, 18, 26, 8]. 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.

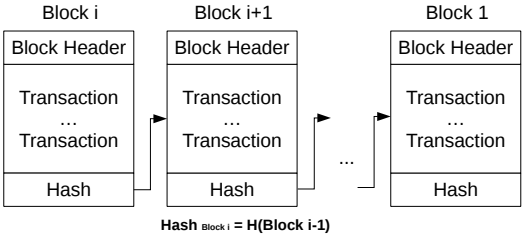


Figure 1: Blockchain structure.

3 Hyperledger Fabric

Hyperledger Fabric (HLF) [26, 9] is an open-source project within the Hyperledger umbrella project.[5] It is a modular permissioned blockchain platform designed to support pluggable implementations of different components, such as the ordering and membership services. HLF enables clients to manage transactions by using chaincodes, endorsing peers and an ordering service.

Chaincode is HLF's counterpart for smart contracts [24]. It consists of code deployed on the HLF'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.

HLF protocol. The HLF general transaction processing protocol [1] – 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.

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

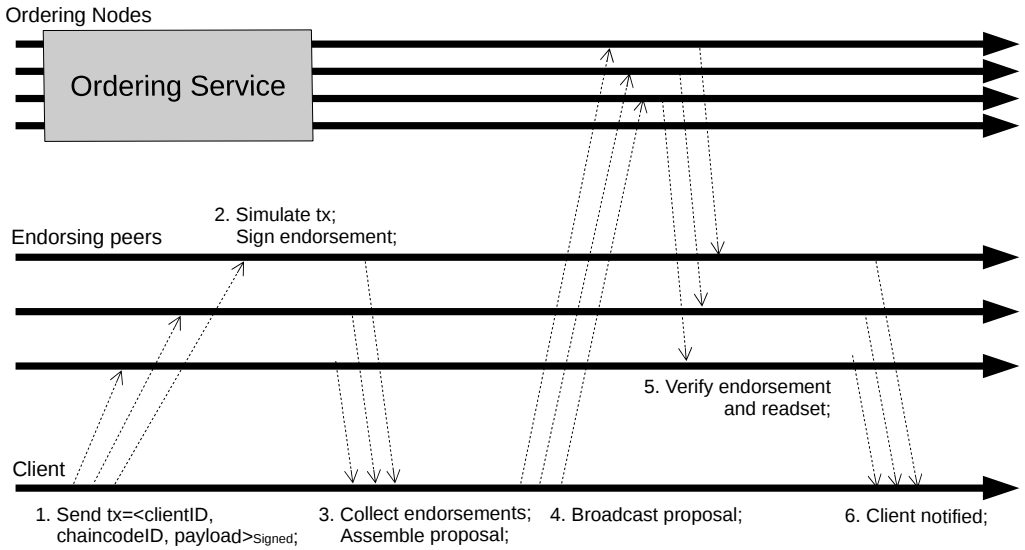


Figure 2: HLF protocol.

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.[6] 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, 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 on the ledger.

An important aspect of the HLF 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 [26].

Pluggable consensus. As mentioned before, HLF is a modular blockchain platform. In particular, one of the components that support plug-and-play capability is the ordering service. Currently, HLF'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 platform; and (2) a replicated ordering service capable of withstanding crash faults, consisting of an Apache Kafka cluster[7] and its respective ZooKeeper ensemble [16]. 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

[6] A channel is a private blockchain on a HLF network, providing data partition. Each peers of the channel share a channel-specific ledger.

[7] `https://kafka.apache.org/`

Kafka-based module is both decentralized and robust, but can only withstand crash faults.

4 BFT-SMaRt & WHEAT

The ordering service presented in this paper was designed on top of existing BFT systems, namely BFT-SMaRt [4] and WHEAT [23]. In this section we present a brief description of these works.

BFT-SMaRt implements a modular SMR protocol on top of a Byzantine consensus algorithm [22]. 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 [10].

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.

This is the message pattern that 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 mechanism is dubbed *synchronization phase* and is described in detail in [22]. We do not describe it in this work because our experiments do not evaluate this part of the protocol.

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 in [10] and uses a vote assignment scheme for efficient quorum usage introduced in [23]. Tentative execution consists of delivering client requests right after finishing the WRITE phase, thus executing the ACCEPT phase asynchronously. This optimization comes at the cost of (1) potentially needing to perform a rollback on the application state if there is a leader change, and (2) forcing clients to wait for $\lceil \frac{n+f+1}{2} \rceil$ messages from replicas (instead of $f+1$) [10]. Moreover, the vote assignment schemes integrate the classical ideas of weighted replication [14, 15, 21] to state machine replication protocols by relying primarily on the fastest replicas present in the system while still preserving its original safety and liveness properties of the 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.

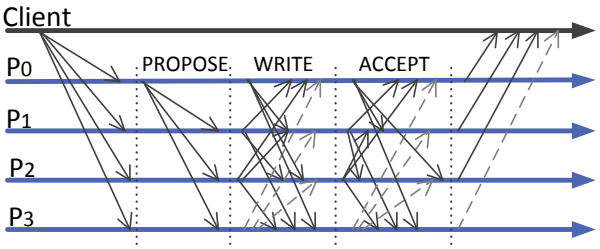


Figure 3: BFT-SMaRt message pattern.

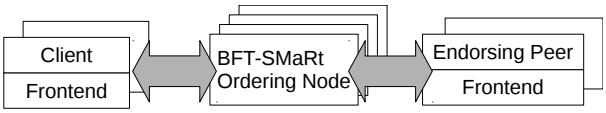


Figure 4: BFT-SMaRt ordering service.

5 BFT-SMaRt Ordering Service

The BFT-SMaRt module for HLF's ordering service consists of an ordering cluster and a set of frontends (Figure 4). 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 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.[8] 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.

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

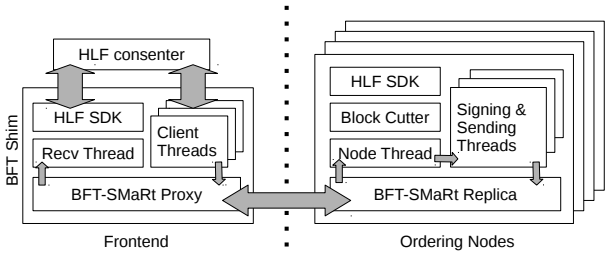


Figure 5: BFT-SMaRt ordering service architecture.

5.1 Architecture

BFT-SMaRt's ordering service architecture is illustrated in Figure 5. The frontend is composed by the HLF consenter and a BFT shim. The HLF consenter is implemented in Go and provides an interface for the HLF codebase to submit envelopes. These envelopes are relayed to the BFT shim using sockets. This shim is implemented in Java and maintains (1) a client thread pool that receive envelopes from the consenter 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 both the consenter and shim perform computations on equivalent data structures, the shim uses the the Hyperledger Fabric SDK to parse and assemble data structures used in HLF.

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 they store the envelopes received from the service replica. Once the blockcutter holds a pre-determined number of envelopes (the block size), it notifies the node thread that it is time to drain its envelopes and create the next block. After the blockcutter is drained, the envelopes are assigned a sequence number associated to their future block and submitted to the signing/sending thread pool alongside with the respective block header (containing the aforementioned sequence number and the cryptographic hashes from the previous header and the hash for the current 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 each node needs to maintain as the application state is the block header from the previous iteration of the node thread. Similarly to the frontend, the HLF SDK is used to correctly handle and create the data structures used by the platform. In addition, the HLF SDK is also used to generate cryptographic hashes and ECDSA (Elliptic Curve DSA) signatures that can be validated by other components of HLF.

Once the block is created and properly signed, they are transmitted to all active frontends. This is done by providing a custom *replier* (supported by the extensible API of BFT-SMART) that instead of sending the execution result (i.e., the generated block) to the invoking client, sends it to the registered BFT-SMART clients (i.e., the frontends).

5.2 Durability and Reconfiguration

Besides the transaction ordering and execution, BFT-SMART also provides additional capabilities that are fundamental for practical state machine replication, such as durability (of state, in case the all ordering nodes fail) and reconfiguration (of the group of ordering nodes). Durability in particular can lead to many inefficiencies on state machine replication systems [3]. Fortunately, our ordering service will not be subject to most of these inefficiencies as the service state is very small: just the sequence number of the next block (a 8-byte integer) and the hash of previous block (a 24-byte byte array).[9] Such small state enables the execution of frequent checkpoints with little performance degradation. This is important for limiting the size of the SMR operation logs, as they are deleted just after a new checkpoint is stored.

A small log and checkpoint allow the addition of new nodes to the ordering cluster, as the most costly operation during a group reconfiguration is the state transfer from one of the up to date nodes to the joining node [4].

6 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 HLF platform, but only the ordering service, which typically is the bottleneck of the system.

6.1 Signature Generation

The throughput of the ordering service (i.e., the rate at which envelopes are added to the blockchain) is bounded by one of three factors: a) the rate at which envelopes are ordered by BFT-SMART for a given envelope size, number of envelopes per block and number of receivers; b) the number of blocks signed per second; or c) the size of the generated blocks. More precisely, given an envelope size *es*, block sizes *bs*, and a number of receivers *r*

[9] It is worth to recall that in HLF the ordering nodes are not responsible for storing the blockchain, just to generate the blocks and disseminate to other peers.

(i.e., the frontends of endorsing and committing peers to which the ordering nodes transmit the generated blocks), peak throughput is bounded as follows:

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

Therefore, we start by presenting a micro-benchmark designed to evaluate the performance associated with the signature of HLF blocks.[10] In particular, this micro-benchmark is aimed at exploring the impact of signature parallelization using up to 16 worker threads with blocks containing 10 envelopes of 0 bytes each. The experiment was conducted in a Dell PowerEdge R410 machine, 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.

Results. The results for the micro-benchmark are depicted in Figure 6. As expected, the rate of signature generation increases with the number of available worker threads, reaching a maximum rate of 8.400 ECDSA signatures/second. This means that if each block contains 10 envelopes, we have a theoretical upper bound of 84.000 transactions/seconds in our servers for this block size.

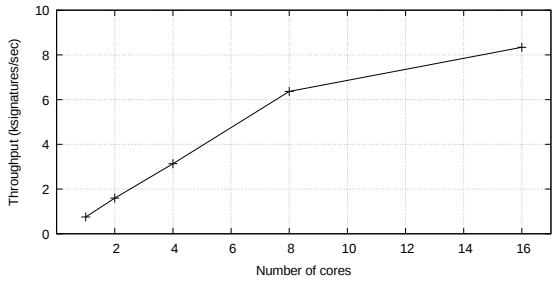


Figure 6: Signature Generation for Fabric blocks.

We also executed this micro-benchmarks with different envelope and block sizes, but we omit these results because they are similar to Figure 6. This is because the signatures are generated against the block header rather than against the whole block. Since the header's size is constant regardless of the data contained in the block, the performance remains constant.

6.2 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 connected through a Gigabit ethernet.

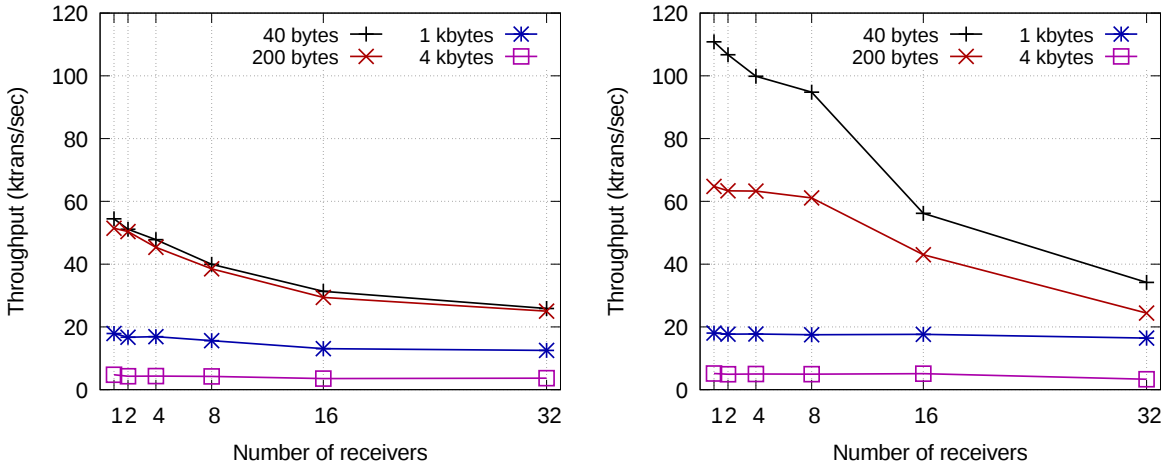
For each micro-benchmark configured to have x nodes and y envelopes/block, we gathered results for (1) envelopes with different sizes, and (2) a variable number of receivers. More precisely, each envelope size is representative of submitting to the ordering cluster: (1) a SHA-256 hash (40 bytes); (2) three ECDSA endorsement signatures (200 bytes); and (3) transaction messages of 1 and 4 kbytes. In practice, and considering the way HLF 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 the platform.

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 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.

Results. The obtained results for local-area are presented in Figure 7. 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, the PROPOSE message of the underlying replication protocol can have up to 0.39/1.6 Mbytes for these envelope sizes.

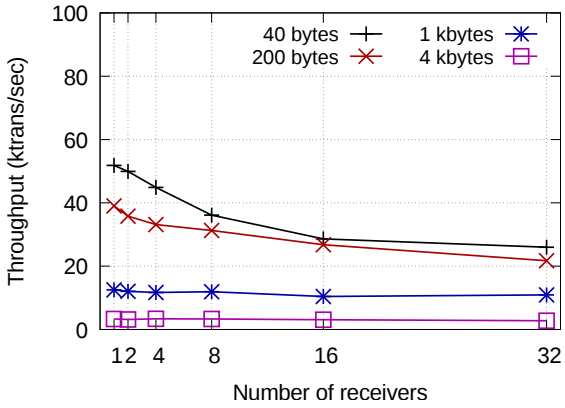
It can be observed that when using 10 envelopes/block (Figures 7a, 7c, and 7e), the maximum throughput observed is approximately 50.000 transactions/second (when there exists only 1 to 2 receivers in the system), which is below the values observed in Section 6.1. Nonetheless, this can be explained by the fact that signa-

[10] The equation consider a block is signed only once by each ordering node, however, in HLF 1.0 sometimes a block need to be signed twice. The second signature is needed to attach the block transaction to an execution context (but details are out of the scope of this paper). If this is the case for the considered application, the TP_{signed} term used in the equation must be exchanged by $\frac{TP_{signed}}{2}$.

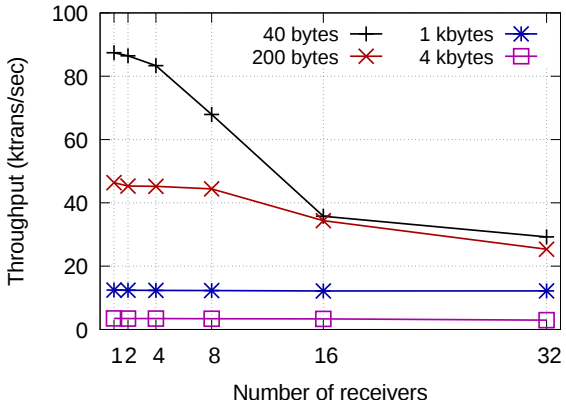


(a) 4 orderers, 10 envelopes/block.

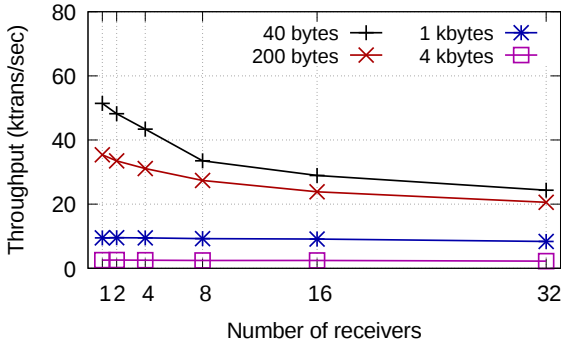
(b) 4 orderers, 100 envelopes/block.



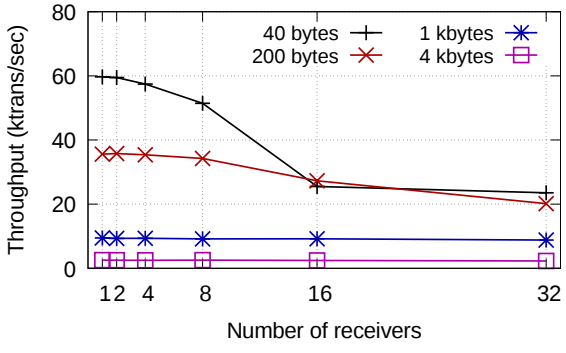
(c) 7 orderers, 10 envelopes/block.



(d) 7 orderers, 100 envelopes/block.



(e) 10 orderers, 10 envelopes/block.



(f) 10 orderers, 100 envelopes/block.

Figure 7: BFT-SMART Ordering Service throughput for different envelope, block and cluster sizes.

ture 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 drop in relation to the micro-benchmark from Section 6.1 – which executed in a single machine, stripped of the overhead associated with BFT-SMART– is to be expected. Moreover, for up to 2 receivers and envelope sizes of 1 and 4 kbytes, the peak throughput becomes similar to the results observed in [4]. 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 7b, 7d, and 7f 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.

6.3 Geo-distributed Ordering Cluster

In addition to the aforementioned micro-benchmarks deployed in a local datacenter, we also conducted a geo-distributed benchmark focused on collecting latency measurements at 4 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 Canada (frontend only), Oregon (collocated with leader node weighting V_{max} in WHEAT), Virginia (collocated with non-leader node, but still weighting V_{max}) and São Paulo.[11] Each frontend was configured to launch enough client threads to keep node throughput always above 1000 transactions/second.

Results. Figure 8 presents the results for the geo-distributed micro-benchmark with a a block size of 10 envelopes. As expected, WHEAT's latency is consistently lower than BFT-SMaRt's across all frontends by almost 50%. It is worth pointing out that envelope size has a relatively minor impact on latency: across all regions, the difference between a 40 and a 4k bytes envelope was never above 29 milliseconds for any percentile or protocol. In fact, it is the placement of the frontends that can exhibit a larger impact on latency: the difference between Virginia (weighted V_{max}) and São Paulo (weighted V_{min}) is above 43 milliseconds for BFT-SMaRt (+6.5%) and above 90 milliseconds (+23%) for WHEAT. In addition, the difference between São Paulo and Oregon/Canada is even larger (58 milliseconds for BFT-SMaRt and 100 miliseconds for WHEAT, corresponding to an increase of +8,5% and +27% respectively).

We also repeated the experiment for blocks of 100 envelopes (Figure 9). The results are similar to the previous experiment, but with increased latency (up to 63 milliseconds higher). This is because with similar workload but a larger block size, the rate of block generation decreases, which has a direct impact on latency.

7 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 [19]. 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 [27].

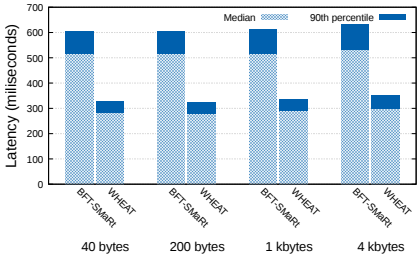
Because of the known performance penalty associated with Proof-of-Work creation and the fact that Blockchain technology is gaining the attention of many industries, the idea of permissioned blockchains are quickly gaining traction. Examples of other permissioned blockchain platforms include Chain,[12] which uses the Federated Consensus algorithm.[13] Tendermint [17] implements the BFT protocol designed by Buchman et. al.[5]. Kadena [18] uses a variant of the Raft consensu protocol [20] adapted to Byzantine faults [11]. Finally, Symbiont Assembly[14] uses a Go implementation of the Mod-SMaRt algorithm [22] and heavily follows the design of BFT-SMaRt. A recent survey compares all these permissioned protocols and points BFT-SMaRt as a prominent candidate for implementing this type of ledgers.

[11] According to the WHEAT binary weight distribution for BFT state machine replication [23], when using five replicas, two of them will have weight $V_{max} = 2$ and the remaining three will have $V_{min} = 1$.

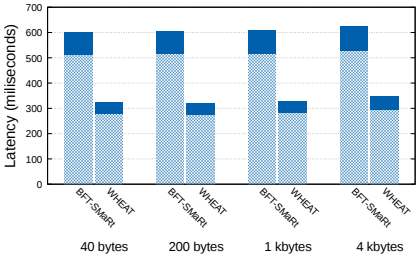
[12] `https://chain.com/`

[13] `https://chain.com/docs/1.2/protocol/papers/federated-consensus`

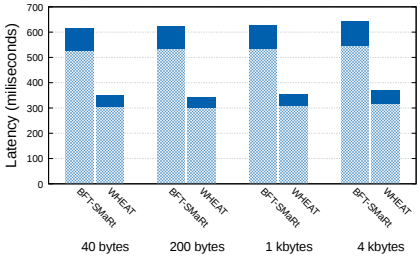
[14] `https://symbiont.io/technology/assembly/`



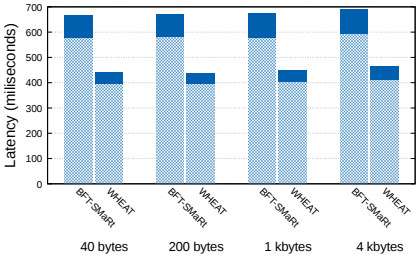
(a) Canada (clients only).



(b) Oregon (weighted V_{max}, leader node).

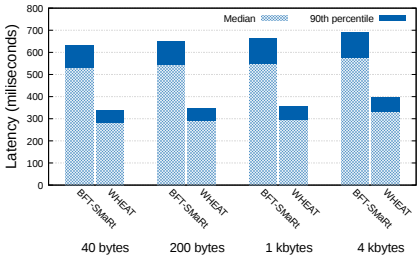


(c) Virginia (weighted V_{max}).

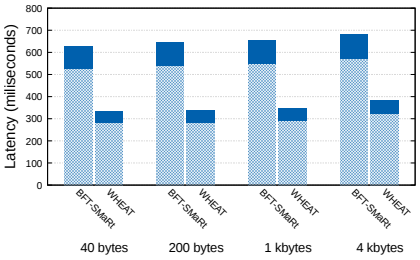


(d) São Paulo (weighted V_{min}).

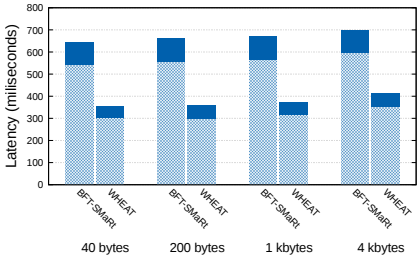
Figure 8: Amazon EC2 latency results (4 receivers, blocks with 10 envelopes).



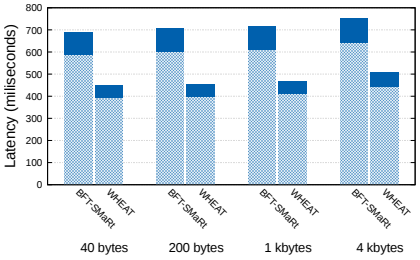
(a) Canada (clients only).



(b) Oregon (weighted V_{max}, leader node).



(c) Virginia (weighted V_{max}).



(d) São Paulo (weighted V_{min}).

Figure 9: Amazon EC2 latency results (4 receivers, blocks with 100 envelopes).

8 Conclusion

The evaluation confirms the initial hypothesis about peak throughput being bounded either by the rate at which signatures can 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 transactional 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 peak throughput of approximately 2200 transactions/second – which

is more 2× of Ethereum's theoretical peak of 1000 transactions/second [7], and vastly superior than Bitcoin's peak of 7 transaction/second [25]. 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.

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

References

[1] Elli Androulaki, Christian Cachin, Konstantinos Christidis, Chet Murthy, Binh Nguyen, and Marko Vukolic. Next consensus architecture proposal, 2016.

[2] Adam Back. Hashcash - a denial of service countermeasure. http://www.hashcash.org/papers/hashcash.pdf, 2002.

[3] Alysson Bessani, Marcel Santos, Joao Felix, Nuno Neves, and Miguel Correia. On the efficiency of durable state machine replication. In *Proc. of the USENIX Annual Technical Conference – USENIX ATC 2013*, June 2013.

[4] Alysson Bessani, Joao Sousa, and Eduardo Alchieri. State machine replication for the masses with BFT-SMART. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

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

[6] Vitalik Buterin. Ethereum white paper. https://github.com/ethereum/wiki/wiki/White-Paper, 2015.

[7] Vitalik Buterin. Ethereum platform review: Opportunities and challenges for private and consortium blockchains. http://r3cev.com, 2016.

[8] C. Cachin and M Vukolic. Blockchain consensus protocols in the wild. Technical Report arXiv:1707.01873, IBM Research - Zurich, July 2017.

[9] Christian Cachin. Architecture of the hyperledger blockchain fabric. https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf, 2016.

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

[11] Christopher Copeland and Hongxia Zhong. Tangaroa: a byzantine fault tolerant raft. http://www.scs.stanford.edu/14au-cs244b/labs/projects/copeland_zhong.pdf, 2014.

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

[13] Juan Garay, Aggelos Kiayias, and Nikos 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.

[14] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, 1985.

[15] David Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM SIGOPS Symposium on Operating Systems Principles*, 1979.

[16] 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.

[17] Jae Kwon. Tendermint: Consensus without mining, http://www.the-blockchain.com/docs/tendermint2016.

[18] Will Martino. Kadena: The first scalable, high performance private blockchain, http://kadena.io/docs/kadena-consensuswhitepaper-aug2016.pdf, 2016.

[19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf, 2009.

[20] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference*, Philadelphia, PA, USA, 2014.

[21] Jehan Pris. Voting with witnesses: A consistency scheme for replicated files. In *In Proceedings of the 6th International Conference on Distributed Computing Systems*, Cambridge, MA, USA, 1986.

[22] Joao Sousa and Alysson Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proceedings of the 9th European Dependable Computing Conference*, 2012.

[23] Joao Sousa and Alysson 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*, Sept 2015.

[24] Nick Szabo. Smart contracts: Building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought*, (16), 1996.

[25] Marko 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, iNetSec 2015, Zurich, Switzerland, October 29, 2015*, pages 112–125, Zurich, Switzerland, 2015.

[26] Marko Vukolić. Rethinking permissioned blockchains. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, Abu Dhabi, United Arab Emirates, 2017.

[27] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, http://gavwood.com/paper.pdf, 2015.