Dissecting the Performance of Chained-BFT

Fangyu Gai*, Ali Farahbakhsh*, Jianyu Niu*, Chen Feng*, Ivan Beschastnikh†, Hao Duan‡
University of British Columbia (*Okanagan Campus, †Vancouver Campus), Canada,
Email: *{fangyu.gai, mralifar, jianyu.niu, chen.feng}@ubc.ca, †bestchai@cs.ubc.ca
‡Hangzhou Qulian Technology Co., Ltd., China, Email: duan.hao@hyperchain.cn

***Abstract*—Permissioned blockchains employ Byzantine fault-tolerant (BFT) state machine replication (SMR) to reach agreement on an ever-growing, linearly ordered log of transactions. A new paradigm, combined with decades of research in BFT SMR and blockchain (namely chained-BFT, or cBFT), has emerged for directly constructing blockchain protocols. Chained-BFT protocols have a unifying propose-vote scheme instead of multiple different voting phases with a set of voting and commit rules to guarantee safety and liveness. However, distinct voting and commit rules impose varying impacts on performance under different workloads, network conditions, and Byzantine attacks. Therefore, a fair comparison of the proposed protocols poses a challenge that has not yet been addressed by existing work.**

We fill this gap by studying a family of cBFT protocols with a two-pronged systematic approach. First, we present an evaluation and benchmarking framework, called Bamboo, for quick prototyping of cBFT protocols. To validate Bamboo, we introduce an analytic model using queuing theory which also offers a back-of-the-envelope guide for dissecting these protocols. We build multiple cBFT protocols using Bamboo and we are the first to fairly compare three cBFT representatives (i.e., HotStuff, two-chain HotStuff, and Streamlet). We evaluated these protocols under various parameters and scenarios, including two Byzantine attacks that have not been widely discussed in the literature. Our findings reveal interesting trade-offs (e.g., responsiveness vs. forking-resilience) between different cBFT protocols and their design choices, which provide developers and researchers with insights into the design and implementation of this protocol family.

***Index Terms*—Byzantine fault-tolerant, evaluation, performance, framework**

I. Introduction

Classic Byzantine fault-tolerant (BFT) state machine replication (SMR) protocols like PBFT [1] rely on a stable leader to drive the protocol until a view change occurs. This limits their scalability and deployability in a context that involves thousands of nodes and requires highly distributed trust. Recent work has been exploring an alternative, called *chained-BFT* or cBFT, combining decades of research in BFT and state-of-the-art blockchain work, which is considered to be the next generation BFT for blockchains [2]–[5].

The run-time of a cBFT protocol is divided into views. Each view has a designated leader, which is elected via some leader election protocol. Unlike classical BFT protocols, such as PBFT [1], in which each proposed block has to go through three different phases of voting, cBFT protocols use the chained structure that enables a single-phase *Propose-Vote* scheme. Since blocks are cryptographically linked together, a vote cast on a block is also cast on the preceding blocks of the same branch. This allows for streamlining of decisions and fewer messages types, and also eases the burden of reasoning about the correctness of the protocols. See an illustration of chained HotStuff in Figure 1. Votes for the same block from distinct replicas are accumulated as a *Quorum Certificate* (short for QC) if a threshold is reached (i.e., over two thirds of all the nodes) and the QC is also recorded on the blockchain along with the relevant block for bookkeeping.

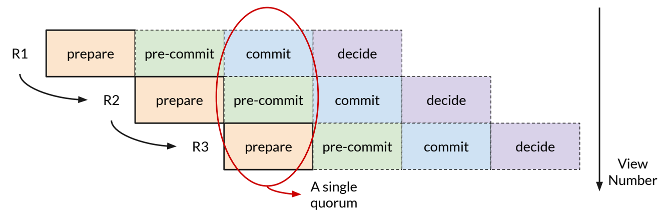

Fig. 1: Run-time of chained-HotStuff [2]. Replicas R_1, R_2, and R_3 are rotating (e.g., round-robin) to propose blocks. A QC accumulated at a phase for a single block can serve in different phases of the preceding blocks.

Liveness and safety properties are guaranteed by voting and commit rules, both of which characterize a cBFT protocol. Informally, a *voting rule* requires that an incoming block is voted on only if it extends a valid branch, while a *commit rule* requires that a chain of blocks is committed by removing a certain number of blocks at the tail of the blockchain. However, the choice of rules changes performance under different conditions: network environment (latency and bandwidth), workload, deployment parameters, and failures.

The idea of connecting modern blockchains with classic BFT originates implicitly in Casper-FFG [6], followed by a series of recent proposals [2]–[4], [6]–[9]. Many companies have been using these protocols. For example, Diem from Facebook (previously known as Libra), Flow from Dapper Labs, and Hyperchain from Qulian have deployed cBFT protocols as their core infrastructure for payment, gaming, and supply chain systems [10]–[12]. Understanding the performance of these protocols is therefore important, yet non-trivial because they differ in their voting and commit rules and design choices. As a result, to our knowledge, there has been no study that empirically provides a fair and comprehensive comparison of cBFT protocols.

To dissect and explain the performance of chained-BFT variants, we propose a novel framework called *Bamboo*. Bamboo provides well-defined interfaces so that developers can quickly prototype their own cBFT protocols by defining voting/commit rules. We used Bamboo to build multiple cBFT protocols, including HotStuff [2], two-chain HotStuff, Streamlet [3], Fast-HotStuff [4], and LBFT [5]. Due to space we only study the first three protocols in this paper. We comprehensively evaluated the performance of these protocols when subjected to different consensus and network parameters, including two Byzantine attacks that could lead to blockchain forking. Since our protocol implementations share most components, Bamboo provided a fair comparison, especially on the impact of different voting/commit rule choices.

A challenge we face when engineering Bamboo is finding the right modularity boundary to express a variety of BFT protocol designs. In Bamboo, we divide a protocol into three modules: *data*, *pacemaker*, and *safety*. The data module maintains a block forest so that blocks can be added and pruned efficiently. The pacemaker module ensures liveness: it allows slow nodes to catch up to the latest view so that the protocol can always make progress. The safety module is for making decisions when new messages arrive. In particular, the safety module defines the **Proposing** rule (i.e., how to make a block proposal), **Voting** rule (i.e., whether to vote for an incoming block), **State Updating** rule (i.e., how the state should be updated), and the **Commit** rule (i.e., whether to commit a set of blocks). The data and pacemaker modules are shared by many protocols, leaving the safety module to be specified by developers.

To cross-validate our Bamboo BFT implementations, we devised an analytical model using queuing theory [13]. This model estimates the latency and the throughput of a cBFT protocol implementation. We show that the results from experiments executed with Bamboo align with our mathematical model, which further indicates that our model can be used to dissect the performance of a cBFT protocol and provides a back-of-the-envelope performance estimate. In summary, we make the following contributions:

- We present the first prototyping and evaluation framework, called Bamboo, for building, understanding, and comparing chained-BFT protocols. We have released the framework for public use[1].
- We introduce an analytical model using queuing theory, which verifies our Bamboo-based implementations and provides a back-of-the-envelope performance estimation for cBFT protocols.
- We conduct a comprehensive evaluation of three representative cBFT protocols: HotStuff, two-chain HotStuff, and Streamlet. Our empirical results capture the performance differences among these protocols under various scenarios including two Byzantine cases that have not been widely discussed in the literature.

[1]https://github.com/gitferry/bamboo

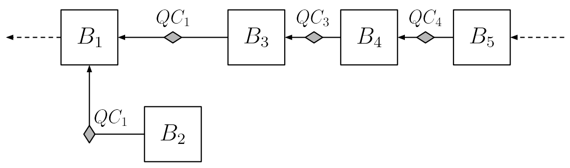

Fig. 2: The tree structure of cBFT-based blockchains. The subscript of a block B refers to the view of the block being proposed while the subscript of a QC refers to the block that the QC certifies. Note that forks can happen because of network delay and malicious behavior.

II. Chained-BFT Protocols

In this section, we present the family of chained-BFT (cBFT) SMR protocols. We first provide an overview of cBFT to outline the basic machinery and then introduce protocol details. Our descriptions of the protocols are informal. Xiang et al. [9] present a more formal description.

A. Overview

At a high level, cBFT protocols share a unifying propose-vote paradigm in which they assign transactions coming from the clients a unique order in the global ledger. This ledger, also known as a blockchain, is a sequence of blocks cryptographically linked together by hashes. Each block in a blockchain contains a hash of its parent block along with a batch of transactions and other metadata.

Similar to classic BFT protocols, cBFT protocols are driven by leader nodes and operate in a view-by-view manner. Each participant takes actions on receipt of messages according to four protocol-specific rules: **Proposing**, **Voting**, **State Updating**, and **Commit**. Each view has a designated leader chosen at random, which proposes a block according to the **Proposing** rule and distributes the block through the blockchain network. On receiving a block, replicas take actions according to the **Voting** rule and update their local state according to the **State Updating** rule. For each view, replicas should certify the validity of the proposed block by forming a *Quorum Certificate* (or QC) for the block. A block with a valid QC is considered certified. The basic structure of a blockchain is depicted in Figure 2. Forks can happen because of conflicting blocks, which is a scenario in which two blocks do not extend each other. Conflicting blocks might arise because of network delays or proposers deliberately ignoring the tail of the blockchain.

Replicas finalize a block whenever the block satisfies the **Commit** rule based on their local state. Once a block is finalized, the entire prefix of the chain is also finalized. Rules dictate that all finalized blocks remain in a single chain. Finalized blocks can be removed from memory to persistent storage for garbage collection.

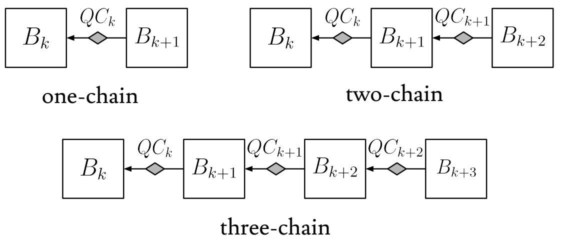

Fig. 3: Illustration of HotStuff's one-chain, two-chain, and three-chain.

B. HotStuff

The HotStuff protocol [2] is considered as the first BFT SMR protocol that achieves linear view change and optimistic responsiveness[2]. These two properties combined make frequent leader rotation practical, which is crucial for decentralization (replicas are equal) and liveness. Before describing the four rules behind HotStuff, we first introduce some definitions. HotStuff defines a **one-chain** as a block certified by a QC, a **two-chain** as a one-chain that has a direct descendent one-chain, and a **three-chain** as a two-chain whose tail block is certified, illustrated in Figure 3.

State Updating rule. We first define some important state variables. The locked block ($lBlock$) is the head of the highest two-chain. The last voted view ($lvView$) is the highest view in which the voting rule is met. The highest QC (hQC) is the highest QC collected from received messages. The variables $lBlock$ and hQC are updated when a new QC is received, while $lvView$ is updated right after a vote is sent.

Proposing rule. When a replica becomes the leader for a certain view, it builds a block based on hQC and broadcasts the proposal.

Voting rule. When a replica sees an incoming block b^*, it checks to see (1) whether $b^*.view$ is larger than $lvView$, and (2) whether it extends $lBlock$ ($lBlock$ is b^*'s grandparent block) or its parent block has a higher view than that of $lBlock$. The vote is sent to the leader of the next view when the two conditions are met. We ignore semantic checks for brevity.

Commit rule. HotStuff follows a three-chain commit rule to make decisions on blocks to be committed. The head of the three-chain along with all the preceding blocks in the same branch are committed as long as a three-chain emerges. As illustrated in Figure 2, when b_{v4} becomes certified (on receiving QC_{v4}), b_{v1} is not committed since b_{v3} is not its directed descendent one-chain. As soon as b_{v5} is certified (on receiving QC_{v5}), b_{v3}, b_{v1}, and all their preceding blocks become committed.

HotStuff decouples liveness guarantee into a module called *Pacemaker* which is used to synchronize sufficiently many honest nodes into the same view if they happen to be out of sync. We leave details of this process to Section III-B.

[2]Optimistic responsiveness means that a correct leader is guaranteed to make progress at network speed without waiting for some apriori upper bound on network delay.

C. Two-chain HotStuff

Two-chain HotStuff (2CHS) is a two-phase variant of HotStuff which could commit a block after two rounds of voting, similar to Tendermint [14] and Casper [6]. The state variables are the same as in HotStuff except that in the **State Updating** rule, $lBlock$ is the head of the one-chain. The **Voting** rule and **Proposing** rule remain the same with HotStuff. The **Commit** rule of 2CHS requires a two-chain. Although the 2CHS saves one round of voting which leads to lower latency as compared to HotStuff, it is not responsive. Since the lock is on one-chain, the proposal has to be built on top of the highest one-chain to make progress. To achieve progress, leaders must wait for the maximal network delay to collect messages from a quorum of replicas after a view change.

D. Streamlet

Streamlet is proposed for pedagogical reasons due to its simplicity. The four rules of Streamlet are based on a similar principle of the longest chain rule from Bitcoin. The state of Streamlet is a *notarized* chain which is a chain of certified blocks. The **State Updating** rule is to maintain such a *notarized* chain.

Proposing rule. The leader proposes a block built on top of the longest *notarized* chain.

Voting rule. A replica will vote for the first proposal, only if the proposed block is built on top of the longest *notarized* chain it has seen. Note that the vote is broadcast.

Commit rule. Whenever three blocks proposed in three consecutive views get certified, the first two blocks out of the three along with the ancestor blocks are committed.

The original Streamlet protocol requires a synchronized clock and each view has a duration of 2Δ, which is considered to be the maximum network delay. To fairly compare protocols, we modify Streamlet in Bamboo by replacing the synchronized clock with the same Pacemaker component described in Section III-B. The validity of this replacement has been previously discussed in [15]. Note that in Streamlet, all the messages are echoed and the votes are broadcast, incurring $O(n^3)$ communication complexity. Additionally, although Streamlet also has a three-chain commit rule, it does not enjoy optimistic responsiveness as it still requires timeouts to ensure liveness.

III. Bamboo Design

In this section, we present the Bamboo's design. We start with the observation that cBFT protocols share common components, such as the blockchain data structure, message handlers, and network infrastructures. Bamboo offers implementations for these shared parts. Each component resides in its own loosely-coupled module and exposes a well-defined API. This design enables modules to be extended and replaced to accommodate different design choices. Figure 4 overviews Bamboo's architecture. Developers can easily prototype a

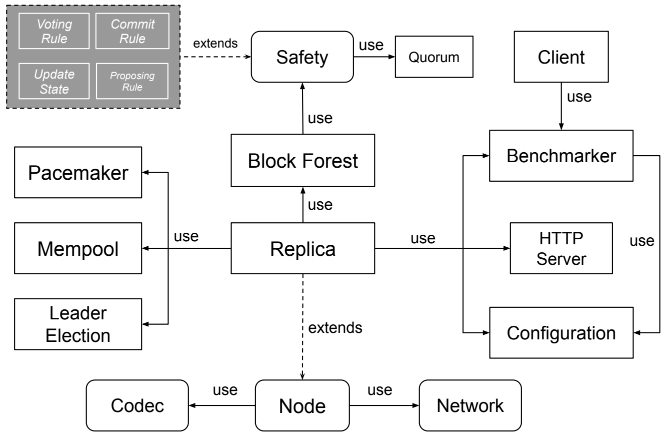

Fig. 4: Modular design of the Bamboo framework.

cBFT-based blockchain protocol by filling in the **voting**, **commit**, **State Updating**, and **Proposing** rules (shaded blocks) that extend the *Safety* API. Using the benchmark facilities, users can conduct comprehensive apple-to-apple comparisons across different cBFT protocols and their design choices. Next, we describe the major components of Bamboo.

A. Block Forest

We use the block forest module to keep track of blocks. Block forest contains multiple block trees, which is a potentially disconnected planar graph. Each vertex in the graph is assigned a height, which increases monotonically. A vertex can only have one parent with a strictly smaller height but it can have multiple children, all with strictly larger height. A block forest provides the ability to prune all vertices up to a specific height. A tree whose root is below the pruning threshold might decompose into multiple disconnected sub-trees as a result of pruning. The Block Forest component guarantees that there is always a main branch, or *main chain*, which contains all the committed blocks cryptographically linked in the proposed order. As a result, a consistency check can be easily conducted by checking the hash of the block at the same height across nodes.

B. Pacemaker

The pacemaker module is in charge of advancing views. It encapsulates the view synchronization logic that ensures liveness of cBFT protocols. Specifically, view synchronization aims to keep a sufficient number of honest replicas in the same view for a sufficiently long period of time so that collaboration will continue and progress can always be made. This module is first defined in HotStuff [2] without detailed specification. LibraBFT [16] provides an implementation of this module which we adopt in our framework. The general idea is essentially the same as a Byzantine fault detector [17]. In this module, whenever a replica times out at its current view, say v, it broadcasts a timeout message $\langle \text{TIMEOUT}, v\rangle$ and advances to view $v+1$ as soon as a quorum $(2f+1)$ of timeout messages of v, called *TimeoutCertificate* (or TC), is received. In addition, the TC will be sent to the leader of view $v+1$ which will drive the protocol forward if it is honest. Note that the Pacemaker module causes quadratic message complexity even though HotStuff has linear message complexity in the happy path. Other proposals such as Cogsworth [18] present optimized solutions that have reduced message complexity and constant latency, which will be considered in future work.

C. Safety Module

The safety module defines all the interfaces needed to implement the consensus core. It consists of the voting rule, commit rule, state updating rule, and the proposing rule. Developers implement these rules according to protocol specifications. The voting and commit rules are key to guaranteeing safety and liveness, as described in Section II. On receipt of incoming messages, the state variables are updated to ensure that the voting and commit rules can be eventually met. The proposing rule defines how a block is proposed in the hope that it can make progress. We implement the two Byzantine strategies (see Section IV-A) by modifying the proposing rule.

TABLE I: Main configuration parameters.

Parameter	Default Value	Description
address	nil	List of peers in which key is the ID and value is the internal IP
master	0	ID of the static leader; rotating if it is set to 0
strategy	silence	Byzantine strategy
byzNo	0	Number of Byzantine nodes
bsize	400	Number of transactions included in a block
memsize	1000	Number of transactions held in the memory pool
psize	0	Payload size of a transaction (bytes)
delay	0	Additional delay of messages sent
timeout	100	Waiting time for entering the next view (ms)
runtime	30	The period clients run for
concurrency	10	Number of concurrent clients

D. Benchmark Facilities

The configuration, client library, and benchmarker components are the core of Bamboo's capability for evaluating cBFT protocols. Bamboo is a configurable framework for both nodes and clients. The configuration component provides configuration parameters about the environment; these are listed in Table I. A configuration is fixed for each run and managed via a JSON file distributed to every node.

The Bamboo client library uses a RESTful API to interact with server nodes, which enables the system to run any benchmark, such as YCSB [19] and Hyperledger Caliper [20]. The benchmarker generates tunable workloads and measures the performance of protocols in terms of throughput, latency, and the two other metrics, which we discuss in the following section. Bamboo also supports simulating network fluctuation and partitioning by tuning the "delay" parameter, which can also be tuned during run-time by sending a "slow" command to a certain node. Since Bamboo currently focuses on protocol-level performance, we adopt an in-memory key-value data store for simplicity. We will consider a VM-based execution layer in our future work.

E. Other components

Mempool. This component serves as a memory pool which is a bidirectional queue in which new transactions are inserted from the back while old transactions (from forked blocks) are inserted from the front. Each node maintains a local memory pool to avoid duplication check.

Network. Bamboo reuses network module from Paxi [21] which is implemented as a simple message-passing model. The transport layer supports TCP and Go channel for communication, which allows both large-scale deployment and single-machine simulation.

Quorum. A quorum system is a key component for ensuring consistency in any distributed fault-tolerant systems. This component supports two simple interfaces to collects votes (via the interface *voted()*) and generate QCs (via *certified()*).

IV. Byzantine Attacks and Metrics

A. Attack Strategies

Since it is impossible to enumerate all the Byzantine attacks [22], we consider only two simple Byzantine attacks that are specific to cBFT protocols and target **only** at their performance, **not** safety nor liveness. These attacks are challenging to detect as the attackers are not violating the protocol from an outsider's view. Although they do not cause safety violation, they could still have significant impact on performance.

In cBFT protocols, a block is not committed until the **Commit** rule is satisfied even if a block is certified (with a QC pointing to it). This is in contrast to non-chained BFT protocols, in which a proposal is either committed or aborted. Therefore, blocks that are conflicting with one another (referring to the same parent block) may appear to be valid according to the **Voting** rule. We call this situation *forking* in the context of blockchain and this can happen due to Byzantine behaviors and network asynchrony. When the blocks become committed, their conflicting counterparts will be garbage collected, thus wasting the efforts of building them.

The two simple attack strategies we show in the following is to show the spirit of maliciously causing forks. We emphasize that both of the strategies are not optimal, and finding an optimal strategy is beyond the scope of this paper. Developers can easily implement these attack strategies in less than 50 LoC of Go code in Bamboo by modifying the **Proposing** rule.

1) Forking Attack: The forking attack aims to overwrite[3] previous blocks that have not been committed even though they are certified. Attackers perform forking attack by proposing conflicting blocks. Take an example from HotStuff, shown in Figure 5. The leader for view 4 has launched a forking attack by building B_4 on top of B_1 instead of B_3. Although B_4 branches out from the main chain, this is a valid proposal

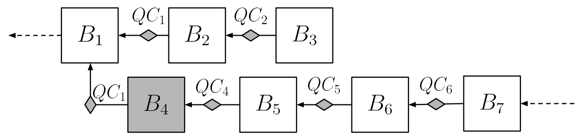

Fig. 5: An example of a forking attack in HotStuff.

from the view of other replicas according to the voting rule. Therefore, replicas will cast votes on B_4 and honest leaders afterward will continue to build blocks following this branch. Eventually, B_4 becomes committed and B_2 and B_3 are overwritten. Similarly, the two-chain HotStuff is also subject to the forking attack, except that the attacker can only create a fork up to B_2 because of the constraint of the voting rule. On the contrary, Streamlet is resilient to this attack because votes are broadcast and replicas only vote for blocks that are built following the longest chain.

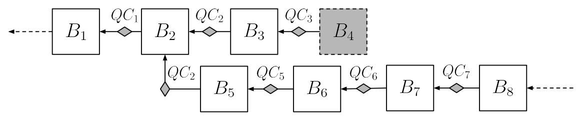

Fig. 6: An example of a silence attack in HotStuff.

2) Silence Attack: The purpose of the silence attack is to break the commit rule, thus extending the time before a block becomes committed. One way to launch this attack is for the attacker to simply remain silent when it is selected as the leader, until the end of the view. An example of this attack is shown in Figure 6, in which B_4 is withheld in view 4 and the (honest) leader for view 5 has to build B_5 following B_2 because of the loss of QC_3. Recall that a block is committed if and only if three consecutive blocks extend it. Thus B_1, which could have been committed in view 4, becomes committed in view 8 (QC_7 is attached to B_8). At the same time, B_3 is overwritten. Note that the silence attack will trigger timeouts, which could lead to long delay, depending on timeout settings.

B. Metrics

Bamboo measures performance in terms of latency, throughput, and two other micro-metrics, *chain growth rate* and *block intervals*. We assume that metrics are measured after the GST (Global Stabilization Time), which means we do not consider network-level attacks from an outsider, such as eclipse attack [23] and Denial-of-Service attack, which may cause a network partition (i.e., asynchronous network) and block progress. Throughput is the number of transactions that are processed per second, or Tx/s, while latency is measured at the client end, and refers to the time between the client sending the transaction and receiving a confirmation. Since it

[3]Overwritten block will be simply ignored and garbage-collected.

is important to show the performance of a protocol in terms of tail latency under stress, these two metrics are measured by increasing the benchmark throughput (by increasing the concurrency level of the clients) until the system is saturated.

To have a deeper understanding of the impact of the two Byzantine attacks mentioned earlier, we introduce two micro metrics, *chain growth rate* and *block interval*.

1) **Chain growth rate (CGR):** From a replica's view, the chain growth rate is defined as the rate of committed blocks appended onto the blockchain over the long run. Let $C(v)$ denote the number of committed blocks during v views. Thus, we have:
$$CGR = \lim_{v\to\infty} \frac{C(v)}{v} \tag{1}$$
2) **Block interval (BI):** The block interval BI measures the average number of views a block takes from being added to the blockchain until being committed over the long run. Let L_i denote the number of views that the i-th block takes from the view b_i is produced to the view that the block is committed during v views. Thus, we have:
$$BI = \lim_{v\to\infty} \frac{\sum_{i=1}^{C(v)} L_i}{C(v)} \tag{2}$$

V. Performance Model

In this section, we introduce a mathematical model that allows us to estimate the delay performance of cBFT protocols. This model is similar in spirit to the queuing model proposed in [21] except that we have different leader-election and commit rules (which complicate the analysis). This model will be used later to validate the Bamboo's implementation.

A. Assumptions

We use the same set of assumptions as used in [21] about the machines, network, and transactions.

1) Machines: We assume that all the machines (being used to host replicas or clients) have the same network bandwidth and identical CPUs. For simplicity, we focus on the case that each machine has a *single* CPU and NIC. We also assume that each replica has a unique local memory pool.

2) Network: We consider a system of N nodes and assume that the Round-Trip Time (RTT) in the network between any two nodes follows a normal distribution with mean μ and standard deviation σ. (In practice, μ and σ can be determined via measurement.) This assumption is justified for a LAN setting in [21].

3) Transactions: For simplicity, we assume that transactions are of equal size so that each block contains the same (maximum) number of transactions denoted by n. Also, we assume that the transactions arrive into the system according to a Poisson process with rate λ.

B. Building Blocks

Before we conduct our delay analysis, we introduce some building blocks. There are two types of delays in the system: machine-related delay and network-related delay.

1) Machine-related delay: Similar to [21], we treat each machine as a single queue consisting of CPU and NIC. The CPU delay, denoted by t_{CPU}, captures the delay of operations such as signing and verifying signatures. Since the machines have identical CPUs, t_{CPU} is a constant parameter.

The NIC delay, denoted by t_{NIC}, is the total amount of time a block spends in the NIC of the sender and the receiver. Let us denote the bandwidth of a machine as b and the size of a block as m. Then, we have $t_{NIC} = \frac{2m}{b}$. The factor of 2 comes from the fact that a block first goes through the sender's NIC and then the receiver's NIC.

2) Network-related delay: The network-related delay can be divided into two parts. First, we have the round trip delay between the client and its associated replica from which it will receive the response to its request. This delay is a normal random variable with mean μ as explained before. Second, we have another delay that captures the amount of time needed *in the network* for a leader to collect a quorum of votes from the replicas. Since a quorum consists of $\frac{2N}{3}$ votes, this delay is equal to the $(\frac{2N}{3}-1)$-th order statistics of $N-1$ i.i.d. normal random variables with mean μ and standard deviation σ. Here, the term of -1 is necessary because the leader already has its own vote, and we just need to analyze the rest of the votes needed for a quorum. The expected value of this delay, denoted by t_Q, can be easily obtained through numerical methods (e.g., [24]). Alternatively, it can be obtained through Monte Carlo simulation as suggested in [21].

C. Delay Analysis

We are now ready to present our delay analysis for cBFT protocols. We will start with the HotStuff protocol [2]. Our analysis will focus on the lifetime of a block in the system, which can be used to approximate the lifetime of its transactions.This approximation is valid since a collection of transactions waiting to be included into a block can be viewed as a "virtual" block waiting to be proposed. We will not consider the effect of Byzantine attacks in our delay analysis, as it has already been discussed in our previous work [25]. Instead, we will focus on the happy-path (assuming a synchronous network and no Byzantine behavior) performance of cBFT protocols (especially HotStuff), because such analysis is important for us to validate our implementation and is missing in [25].

Generally, the latency of a transaction can be divided into the following parts as illustrated in Figure 7:
$$latency = t_L + t_s + t_{Commit} + w_Q, \tag{3}$$
where t_L is the average RTT (Round Trip Time) between the client (who issues this transaction) and its associated replica, t_s is the average service time required to serve the block containing this transaction, t_{Commit} is the average amount of time it takes for the block to get committed once it has been certified, and w_Q is the average waiting time for the block to get processed. We will now derive these terms one by one.

1) Derivation of t_L: As explained in Section V-B2, t_L is equal to μ, a pre-defined parameter from measurement.

2) Derivation of t_s: First, the block containing this transaction has to go through the CPU of the leader, and then through the NICs of the leader and the NIC of each replica the leader will send the proposal. This takes $t_{CPU}+t_{NIC}$ units of time. Second, the block has to wait for a quorum of votes to be gathered by the next leader. Such delay is explained in Section V-B2 with expected value t_Q. In addition, we have the CPU delay of the replicas, and the NIC delay between the next leader and the replicas, given by $t_{CPU}+t_{NIC}$. Finally, once a quorum of votes is received by the next leader, it has to process this quorum, which takes t_{CPU} units of time. Therefore, we have the following expression for t_s:

$$\begin{aligned} t_s &= t_{CPU} + t_{NIC} + t_Q + t_{CPU} + t_{NIC} + t_{CPU} \\ &= 3t_{CPU} + 2t_{NIC} + t_Q. \end{aligned} \tag{4}$$

3) Derivation of t_{commit}: Recall that a block in HotStuff is committed once a three-chain is established. Therefore, in a happy-path condition, a block would have to wait for two other blocks to get certified (see Section II-B for details) so that the three of them would form a three-chain. Thus, we have $t_{commit} = 2t_s$.

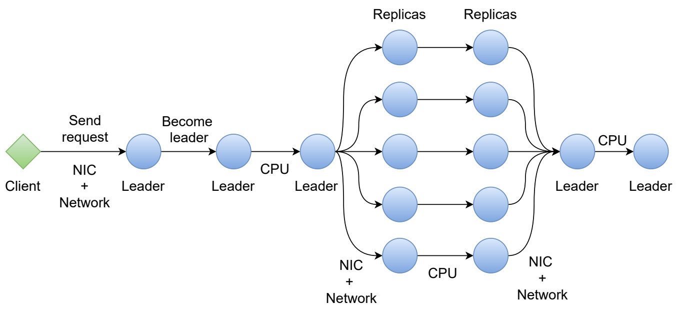

Fig. 7: Transaction lifetime in HotStuff, excluding t_{commit}

4) Derivation of w_Q: Recall that transactions arrive into the system according to a Poisson process with rate λ. Also recall that we assume that each transaction goes to a replica chosen uniformly at random. Thus, transactions arrive at each replica according to a Poisson process with rate $\frac{\lambda}{N}$. Approximately, we can think that blocks arrive at each replica according to a Poisson process with rate $\frac{\lambda}{nN}$, where n is the maximum number of transactions contained in a block.

As explained before, the average service time at each replica is given by t_s. Recall that a replica becomes a leader every N rounds (views) on average and only a leader can propose a block. Hence, the effective average service time (of proposing and processing a block) at each replica is Nt_s. Therefore, the queuing process at a replica can be approximated by an M/D/1 model with the average waiting time given by

$$w_Q = \frac{\rho}{2u(1-\rho)}, \tag{5}$$

where $u = \frac{1}{Nt_s}$ is the effective service rate, $\rho = \frac{\gamma}{u}$, and $\gamma = \frac{\lambda}{nN}$ is the block arrival rate.

D. Analysis of Streamlet and Two-Chain HotStuff

It is straightforward to adapt the above analysis to Streamlet and Two-Chain HotStuff. Equation (3) stays the same for these two protocols since it captures the essence of a block's lifetime. The delay t_L is a network-related parameter and is thus the same across different protocols. The waiting time w_Q also has the same formula, because we can still use an M/D/1 model to approximate the queuing process. Therefore, we shall only focus on t_s and t_{commit}.

1) Two-Chain HotStuff: Since the views of HotStuff and Two-chain HotStuff are identical, t_s in Two-chain HotStuff can be calculated by using Equation (4). The only difference between these two protocols is the fact that Two-chain HotStuff requires a two-chain to commit a block, whereas HotStuff requires a three-chain. Therefore, we have $t_{commit} = t_s$ in Two-chain HotStuff (instead of $t_{commit} = 2t_s$ in HotStuff).

2) Streamlet: The commit rule of Streamlet says that a block has to wait for one more block in the happy path so that it can be committed. Thus, we have $t_{commit} = t_s$ in Streamlet. Next, let us turn to t_s. Recall that the replicas broadcast their votes in Streamlet and they also echo every vote they have received. So, in order to calculate t_s, we need to find out how long it takes for the next leader to receive a quorum of votes, including the processing and NIC delays. This calculation can be done with different levels of complexity based on the underlying assumptions. In the simplest case, we can assume that the next leader receives the quorum of votes from the voters themselves. In this case, the delay analysis for t_s will be the same as that in Section V-C2. However, there is a chance that the next leader receives an echo of a vote even earlier than the vote itself. Intuitively, such an event makes the actual t_s smaller than what is calculated using Equation (4). In other words, our previous analysis in Section V-C2 is still valid as an upper bound. This upper bound turns out to be fairly tight according to the experimental results we will present.

E. Discussion

The protocols we analyze have certain differences that are not necessarily apparent in the model. Streamlet has a cubic communication complexity, and this increases the network and NIC delays (e.g., because of congestion). These differences are captured by the measurements of system parameters such as μ and t_{cpu}.

Additionally, our analysis can be generalized for studying other design choices. Two examples are given below.

- The clients may choose to broadcast transactions instead of sending them to a single replica.
- The leader election procedure can be done differently (e.g., based on hash functions).

This allows us to explore the effect of these design choices.

VI. Experimental Results

In this section, we first present a comparison between our model and Bamboo's implementation. Next, we compare the basic performance of the three protocols with different block sizes, payload sizes, and network delays, followed by a scalability test. Then, we present protocols' tolerance to the two Byzantine attacks discussed in Section IV-A. We finish the section by showing how Bamboo can be used to evaluate protocol responsiveness.

We use Bamboo to conduct experiments to explore the performance of three representative chained-BFT protocols: HotStuff (HS), two-chain HotStuff (2CHS), and Streamlet (SL) because they have clear trade-offs in different scenarios. For instance, HS trades off latency (one more round of voting than 2CHS) for responsiveness while Streamlet compromises message complexity in exchange for simplicity and resilience against forking. We aim to empirically show these trade-offs using Bamboo.

Bamboo is implemented in Golang with around 4,600 LoC and each protocol is around 300 LoC including two Byzantine strategies. Bamboo uses secp256k1 for all digital signatures in both votes and quorum certificates. We carry out our experiments on Tencent Cloud S5.2XLARGE16 instances with 8 vCPUs and 16 GB of RAM. Each replica is initiated on a single VM. We use 2 VMs as clients to send requests to randomly chosen replicas. The VMs are located in the same data center with inter-VM latency below 1ms. Clients issue requests to a randomly selected replica, which replicates the requests when it becomes the leader. Requests are processed using in-memory storage. We do not throttle the bandwidth in any run.

TABLE II: Comparison of transaction arrival rate and transaction throughput in HotStuff, for block size of 400 and 4 replicas.

Arrival rate (Tx/s)	Throughput (Tx/s)
19,992	20,115
39,976	39,895
59,956	59,934
79,880	79,774
99,772	100,159
119,236	119,159
131,232	131,275

A. Model vs. Implementation

To validate our Bamboo-based protocol implementations, we compare the results from our model with results from the protocol implementations. Our experiments use different configurations for the number of nodes (network size) and the block size. We evaluate the following network size/block size pairs: 4/100, 8/100, 4/400, 8/400. The other settings are set according to the values listed in Table I. In each run of an experiment the clients' concurrency level is increased until the network is saturated.

Figure 8 shows four plots of throughput (thousands of transactions per second) versus latency (milliseconds), one plot per network size/block size configuration. Each plot shows a line for each of the three Bamboo-based protocol implementations and another line for corresponding models for each protocol. By comparing the model and Bamboo lines for a protocol we can see how well the model and our implementation match.

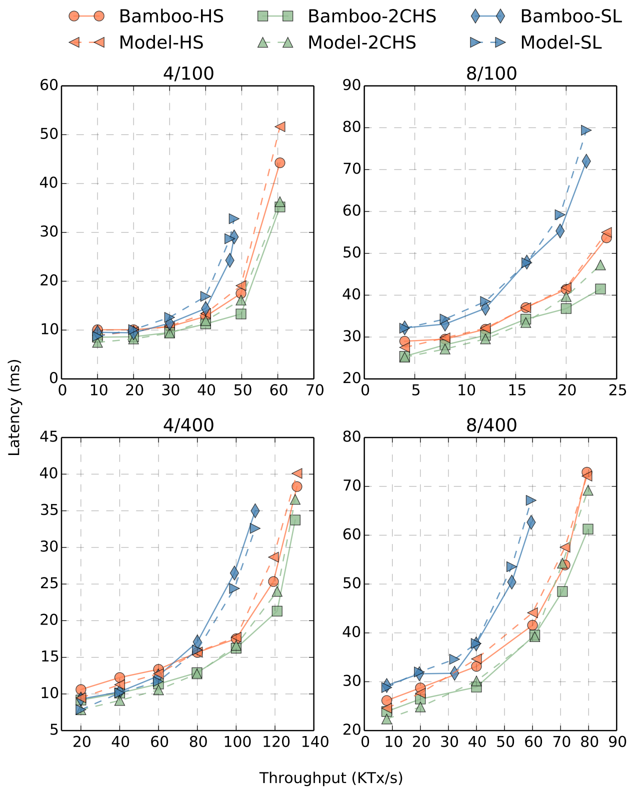

Fig. 8: Comparison of the model and implementation for HotStuff, two-chain HotStuff (2CHS), and Streamlet.

The four plots in Figure 8 demonstrate that our analytical model successfully validates our Bamboo implementations. Furthermore, it indicates that our model can be used to perform back-of-the-envelope performance forecasting of the protocols with given workloads. It is important to note that our model estimates latency based on the transaction arrival rate. However, our experimental measurements indicate that the transaction throughput (throughput of transactions as observed on the blockchain) is roughly equal to the transaction arrival rate (Table II). This is because in most of the workloads the delay is dominated by the queue waiting time, and is (on average) roughly the same for each block, in each workload. Additionally, in low workloads the transactions go through a somewhat idle network, causing them to face similar t_s (recall that t_s is a random variable).

B. Happy-Path Performance

We measure happy-path performance in terms of throughput and latency. In the following experiments, unless specified, we use 4 replicas.

Block sizes. In our first experiment, we vary the block size and compare our implementation by using the original HotStuff (OHS) implementation as the baseline. The original HotStuff library[4] is open-source and written in C++, allowing

[4]https://github.com/hot-stuff/libhotstuff.

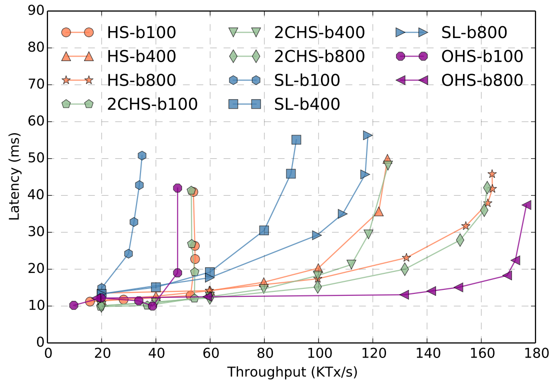

Fig. 9: Throughput vs. latency with block sizes of 100, 400, and 800.

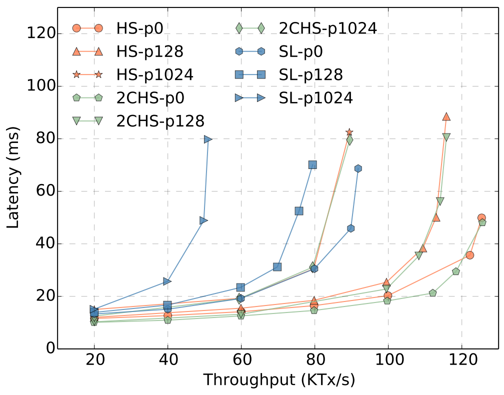

Fig. 10: Throughput vs. latency with transaction payload sizes of 0, 128, and 1024 bytes.

us to easily reproduce their basic results showed in the paper [2]. This benchmark uses requests with zero payload and block sizes of 100, 400, and 800, denoted by “b100”, “b400”, and “b800”, respectively. We only use block sizes of 100 and 800 for OHS as we did not obtain meaningful results from the block size of 400. Bamboo uses a simple batching strategy in which the proposer batches all the transactions in the memory pool if the amount is less than the target block size. We gradually increase the level of client concurrency and request rate until the systems are saturated.

Figure 9 shows the results of the above experiment in a throughput versus latency plot, listing different protocols with varying block sizes. We first observe that all the protocols have a typical L-shaped latency/throughput performance curve. The Bamboo-HotStuff implementation has similar performance to the original one. The slight gap is likely due to the original HotStuff’s use of TCP instead of HTTP to accept requests from clients, different batching strategies, and language distinction. Streamlet has lower-level throughput in all block sizes due to its broadcasting of votes and echoing of messages. The performance gain by increasing the block size from 100 to 400 is remarkable. However, this gain is reduced above 400 requests per block. This is partly because the latency incurred by batching becomes higher than the cost of the crypto operations. We use a block size of 400 in the rest of the experiments.

Transaction payload sizes. Figure 10 illustrates three transaction payload sizes of 0, 128, 1024 (in bytes), denoted by “p0”, “p128”, and “p1024”, respectively. At all payload sizes, we can see a similar pattern for the three protocols. Streamlet has the worst performance and is more sensitive to the payload size due to its message echoing design. The results also show that the latency difference between HotStuff and its two-chain variant becomes less obvious with a larger payload. This is because the transmission delay of a block dominants the latency. We set the payload size to 128 bytes in the rest of the experiments.

Network delays. We introduce additional network delays between replicas, which are configurable in Bamboo’s network module. Figure 11 depicts results under no added network delays and additional network delays of 5ms $\pm$ 1.0ms and 10ms $\pm$ 2.0ms, denoted by “d0”, “d5”, and “d10”, respectively. We can observe that all the protocols suffer when network delay is increased. We already know that without added delays, the two HotStuff protocols have the same throughput while there is a clear gap with Streamlet in terms of latency and throughput. However, when the network delay is 5ms $\pm$ 1.0ms, this gap starts to decrease and at 10ms $\pm$ 2.0ms, Streamlet has comparable performance to 2CHS. This is because the long network delay compensates for the adverse effect of message echoing in Streamlet.

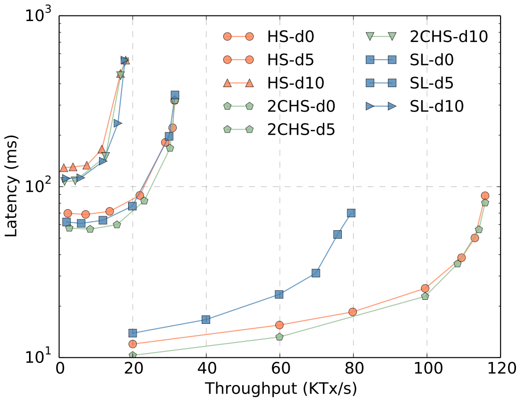

Fig. 11: Throughput vs. latency with additional network delays of 0ms, 5ms ($\pm$ 1ms) and 10ms ($\pm$ 2ms).

Scalability. We evaluate the scalability of the three protocols with nodes from the same cluster. There are no added network delays. We derive each data point by averaging the results of three repeated runs for 10,000 views under the same setting. We show error bars to present the standard deviation. The results are plotted in Figure 12. Streamlet has the worst scalability and its results obtained at the network size above

64 are meaningless due to high message complexity. HotStuff and two-chain HotStuff have comparable scalability and the latency difference decreases as the network size increases. Note that one reason for the sharp increase of latency is the message complexity of the protocols, while another reason is the increase of the waiting time for leader election in which each replica has a local memory pool for storing transactions.

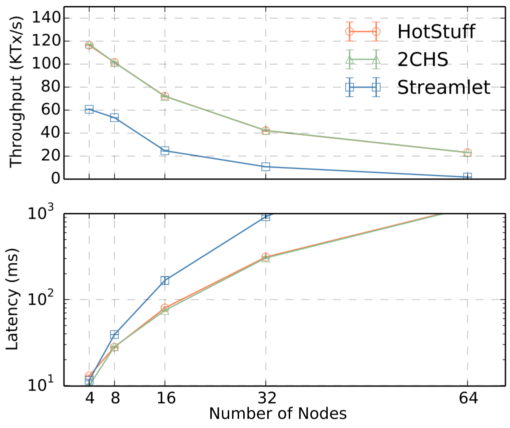

Fig. 12: Scalability for 128 byte payload and a block size of 400 txns.

C. Byzantine Strategies

To evaluate the ability to tolerate Byzantine attacks discussed in Section IV-A, we conduct two experiments, one for each Byzantine strategy. Besides throughput and latency, we also measure two micro metrics chain growth rate (CGR) and block intervals (BI) described in Section IV-B. This adds another dimension for considering the results while avoiding the influence of timeout settings. The transactions contained in the forked blocks are collected back and inserted at the front of the queue of the memory pool. In the two experiments, we run 32 nodes in total with an increasing number of Byzantine nodes. The rest of the settings are the same as the scalability test.

Forking attack. As described in IV-A, attackers aim to overwrite as many uncommitted blocks as possible. They propose blocks that could cause forks without violating the voting rule. Figure 13 depicts the first setting with Byzantine nodes performing a forking attack. The line for Streamlet is flat across all the metrics, indicating that it is immune to the forking attack. This is due to the broadcasting of votes and the longest-chain rule: honest (non-Byzantine) replicas only vote for the longest notarized chain they have ever seen. Therefore, a fork cannot happen in Streamlet as long as the network is synchronous. Two-chain HotStuff outperforms HotStuff in all the metrics. This is because the two-chain commit rule is naturally more resilient against forking in the sense that the attacker can overwrite only one block in 2CHS but can overwrite two blocks in HotStuff. This also explains why

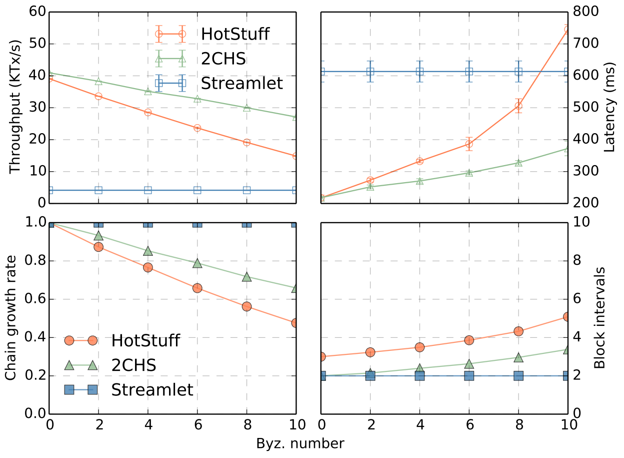

Fig. 13: Protocols under forking attack with fixed 32 nodes in total and increasing Byzantine nodes.

HotStuff and two-chain HotStuff have the same pattern in throughput and CGR. In terms of block intervals, 2CHS and HotStuff have the same pattern but start from 2 and 3 due to their two-chain and three-chain commit rules, respectively. However, there is a rapid growth in HotStuff's latency. This is because the transactions in the forked blocks will be collected back into the memory pool, which increases latency.

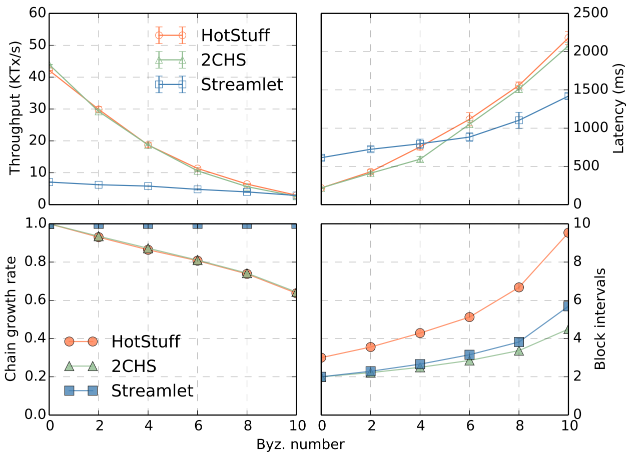

Fig. 14: Protocols under silence attack with fixed 32 nodes in total and increasing Byzantine nodes.

Silence attack. In the silence attack attackers simply remain silent during their leadership, aiming to break the commit rule. When the leader is performing a silence attack, the other replicas wait until timeout and try to enter the next view. In this experiment, we set the timeout to be 50ms to ensure that no additional timeouts are triggered other than the silence attack. Figure 14 shows the results of this experiment. Both HotStuff and 2CHS have the same pattern in CGR and throughput because the silence attack overwrites the last block due to the loss of the QC. The CGR of Streamlet remains 1 at any run because of the broadcasting of votes and its longest-chain rule so that no fork could happen in a synchronous network. All protocols have a drop in throughput because of the increasing number of silent proposers. One note is that although HotStuff and 2CHS show consistently better throughput than Streamlet, Streamlet exhibits graceful degradation due to its immunity to forking. In terms of block intervals, all protocols show higher BI and more rapid growth than when they are under forking attack in Figure 13 because for an attacker it is easier to break the commit rule than it is to cause a fork. Both Streamlet and HotStuff have higher BI than 2CHS because their commit rules are more strict. In the throughput sub-figure on the top right, although Streamlet shows worse latency than HotStuff and 2CHS when the Byz. number is less than 4; but, it outperforms the other two protocols when the number is above 4. This is partly due to its immunity to forking. HotStuff and 2CHS have comparable latency because they have similar CGR.

D. *Responsiveness*

To show the advantage of responsiveness, we conduct experiments using fault injection. In this experiment, we run four nodes under the same setting with the scalability test in Section VI-B except for specific timeout settings for each view. In the first setting, each protocol's timeout is set to 10ms, denoted by "t10". We let both 2CHS and Streamlet make proposals as soon as $2f+1$ messages are received after view change, which is the same as HotStuff. In the second setting, the timeouts are changed to 100ms, denoted by "t100", and we let all the protocols wait for the timeout if a view change occurs. We keep the request rate fairly high in both settings. During the two experiments under two different settings, we manually inject network fluctuation for 10 seconds during which the network delays between nodes fluctuate between 10 and 100 milliseconds. Additionally, we let one of the nodes perform a silence attack (crash) afterward.

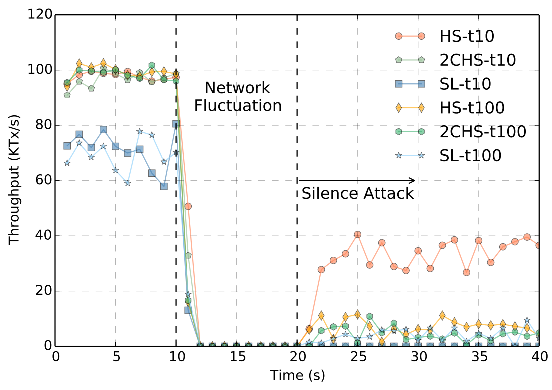

Fig. 15: Responsiveness test with timeout settings of protocols of 10ms and 100ms.

Figure 15 depicts the results from the two settings in one plot. In the first setting (t10), we observe an abrupt stop of all three protocols during the 10 seconds of network fluctuation. The HotStuff protocol proceeds instantly after the network fluctuation stops and presents a graceful wave of throughout afterward due to the silence attack. However, the other protocols never make any progress. This is because 2 non-Byzantine replicas are locked on a block while the other non-Byzantine replica is locked on a different block during the fluctuation. When the silence attack is launched, both protocols will not make any progress.

In the second setting (t100), the timeout of each protocol is set to the maximum network delay during the network fluctuation, therefore, the timeout is not triggered until the crashed node is chosen to be the leader. While all the protocols retain liveness, the throughput is much lower in this setting due to the long timeout.

E. *Discussion*

Our experimental results show that the view of *one-size-fits-all protocols may be tough, if not impossible, to build* [26] also applies to chained-BFT protocols. Different design choices have different trade-offs within a given set of deployment conditions. In a LAN setup, since HotStuff is one round of voting time slower than 2CHS with comparable throughput, 2CHS might be the better choice. But this latency advantage is reduced if the transaction has a large payload (e.g., a smart contract deployment). Optimistic responsiveness makes HotStuff a better candidate for a WAN environment in which the networking conditions are unpredictable. Whereas the 2CHS protocol, which is not responsive, is subject to the timeout attack [27] in a WAN setting. As for attacks that cause forks, the longest chain rule of Streamlet is more resilient at the cost of high message complexity. Adapting this rule into HotStuff is an open question and is a good direction for future work in this space. An alternative way of addressing forking issues is meticulous incentive design.

VII. Related Work

Paxi [21] studies the family of Paxos protocols in a two-pronged systematic approach similar to ours. They present a comprehensive evaluation of their performance both in local area networks (LANs) and wide area networks (WANs). A recent work [28] studies the bottleneck of Tendermint, HotStuff, and Streamlet on top of Paxi but does not evaluate them in Byzantine scenarios. BFTSim proposes a simulation environment for BFT protocols that combines a declarative networking system with a robust network simulator [26]. Nevertheless, we still need real testing on top of simulation, and BFTSim lacks the ability to produce empirical results.

BFT-Bench [29] is the first benchmark framework that evaluates different BFT protocols in various workloads and fault scenarios. It provides systematic means to inject different networking faults and malicious intent into the system, and compares several important BFT protocols. However, although the proposed framework uses a modular approach to specify load injection and performance evaluation, it does not provide enough granularity and modularity for designing the BFT protocol itself. In this paper, we leverage the similarities between cBFT protocols to provide a set of common abstractions that make Bamboo possible.

BlockBench [30] is the first evaluation framework for analyzing existing private blockchains, comparing Hyperledger Fabric with Ethereum and Parity. It dissects a blockchain system into different layers: consensus, data model, execution, and application. Some follow-up works [31], [32] focus on benchmarking a specific version of Hyperledger Fabric and optimize it using various architectural changes. By contrast, Bamboo only focuses on the consensus layer and provides horizontal comparisons among different consensus protocols with identical implementations of the other layers.

Twins [33] provides a novel approach to systematically validate the safety of BFT protocol implementations, while Bamboo only focuses two simple Byzantine cases that target at performance, not safety. A concurrent work called SFT [9] presents an abstraction of cBFT protocols similar to ours. The main contribution of [9] is a new chained-BFT protocol called SFT to strengthen the ability of BFT SMR to tolerate Byzantine faults. In our future work we plan to express and evaluate this protocol with Bamboo.

VIII. Conclusion

We present Bamboo, a prototyping and evaluation framework to compare chained-BFT protocols. We also introduce a mathematical model to validate our experimental results, which also helps to distill the performance of cBFT protocols. We build multiple cBFT protocols with Bamboo and present a comprehensive evaluation of three representatives under various scenarios, including two Byzantine attacks. Our results echo the adage that there is no single best BFT protocol for every situation. Bamboo allows developers to analyze cBFT protocols in light of their fundamental trade-offs, and to construct systems that are best suited to their context. In our future work, we will use Bamboo to further explore cBFT protocol designs, focusing on protocols with relaxed consistency guarantees.

IX. Acknowledgement

This work was supported by Natural Sciences and Engineering Research Council of Canada (NSERC) CREATE 528125-2019, Joint NSERC Engage and Mitacs Grant 538022-19, and NSERC Discovery Grant RGPIN-2016-05310. Ivan Beschastnikh also acknowledges the support of the NSERC Discovery Grant RGPIN-2020-05203.

References

[1] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proc. of OSDI, 1999.*

[2] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proc. of PODC, 2019.*

[3] Benjamin Y. Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. *IACR Cryptology ePrint Archive*, 2020:88, 2020.

[4] Mohammad M. Jalalzai, Jianyu Niu, and Chen Feng. Fast-hotstuff: A fast and resilient HotStuff protocol. *CoRR*, abs/2010.11454, 2020.

[5] Jianyu Niu and Chen Feng. Leaderless Byzantine fault tolerant consensus. *CoRR*, abs/2012.01636, 2020.

[6] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[7] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, and Ling Ren. Dfinity consensus, explored. *IACR Cryptol. ePrint Arch.*, 2018:1153, 2018.

[8] T.-H. Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *IACR Cryptology ePrint Archive*, 2018:981, 2018.

[9] Zhuolun Xiang, Dahlia Malkhi, Kartik Nayak, and Ling Ren. Strengthened fault tolerance in Byzantine fault tolerant replication. *CoRR*, abs/2101.03715, 2021.

[10] Novi. Diembft. https://www.novi.com/.

[11] Dapper Lab. Flow platform. https://www.onflow.org/.

[12] Qulian. Hyperchain. https://www.hyperchain.cn/.

[13] Arnold O. Allen. *Probability, statistics and queueing theory - with computer science applications (2. ed.).* Academic Press, 1990.

[14] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Dissecting Tendermint. *CoRR*, abs/1809.09858, 2018.

[15] Shir Cohen and Dahlia Malkhi. What they did not teach you in Streamlet. https://dahliamalkhi.github.io/posts/2020/12/what-they-didnt-teach-you-in-streamlet/.

[16] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the Libra blockchain, 2019.

[17] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *Comput. J.*, 46(1):16–35, 2003.

[18] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. *arXiv preprint arXiv:1909.05204*, 2019.

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of ACM SoCC, 2010.*

[20] Hyperledger caliper. https://www.hyperledger.org/projects/caliper.

[21] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. Dissecting the performance of strongly-consistent replication protocols. In *Proc. of ACM SIGMOD, 2019.*

[22] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[23] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on Bitcoin's peer-to-peer network. In *Proc. of USENIX Security, 2015.*

[24] D. Teichroew. Tables of expected values of order statistics and products of order statistics for samples of size twenty and less from the normal distribution. *Ann. Math. Statist.*, 27(2):410–426, 1956.

[25] Jianyu Niu, Fangyu Gai, Mohammad M. Jalalzai, and Chen Feng. On the performance of pipelined HotStuff. In *Proc. of IEEE INFOCOM, 2021.*

[26] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. BFT protocols under fire. In *Proc. of NSDI, 2008.*

[27] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dependable Secur. Comput.*, 8(4):564–577, 2011.

[28] Salem Alqahtani and Murat Demirbas. Bottlenecks in blockchain consensus protocols. *CoRR*, abs/2103.04234, 2021.

[29] Divya Gupta, Lucas Perronne, and Sara Bouchenak. BFT-Bench: Towards a practical evaluation of robustness and effectiveness of BFT protocols. In *Proc. of IFIP ICDAIS, 2016.*

[30] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. BLOCKBENCH: A framework for analyzing private blockchains. In *Proc. of ACM SIGMOD, 2017.*

[31] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *Proc. of IEEE MASCOTS, 2018.*

[32] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20, 000 transactions per second. In *Proc. of IEEE ICBC, 2019.*

[33] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, and Dahlia Malkhi. Twins: White-glove approach for BFT testing. *CoRR*, abs/2004.10617, 2020.