Bottlenecks in Blockchain Consensus Protocols

Salem Alqahtani
Computer Science Department
University at Buffalo,SUNY
salemmoh@buffalo.edu

Murat Demirbas
Computer Science Department
University at Buffalo,SUNY
demirbas@buffalo.edu

***Abstract*—Most of the Blockchain permissioned systems employ Byzantine fault-tolerance (BFT) consensus protocols to ensure that honest validators agree on the order for appending entries to their ledgers. In this paper, we study the performance and the scalability of prominent consensus protocols, namely PBFT, Tendermint, HotStuff, and Streamlet, both analytically via load formulas and practically via implementation and evaluation. Under identical conditions, we identify the bottlenecks of these consensus protocols and show that these protocols do not scale well as the number of validators increases. Our investigation points to the communication complexity as the culprit. Even when there is enough network bandwidth, the CPU cost of serialization and deserialization of the messages limits the throughput and increases the latency of the protocols. To alleviate the bottlenecks, the most useful techniques include reducing the communication complexity, rotating the hotspot of communications, and pipelining across consensus instances.**

Keywords: Consensus, PBFT, Tendermint, HotStuff, Streamlet.

I. Introduction

BLOCKCHAIN systems aim to provide trustless decentralized processing and storage of transactions, immutability, and tamper-resistance. Most of the Blockchains employ BFT [1] consensus protocols to ensure that the validators agree on the order for appending new transactions to their ledgers. In particular, the Practical Byzantine Fault Tolerance (PBFT) [2] protocol forms the basis for most BFT consensus protocols, such as Tendermint [3], and HotStuff [4].

PBFT builds on the Paxos [5] protocol and extends its crash failure to Byzantine fault-tolerance to defend against adversarial participants that can arbitrarily deviate from the protocol. PBFT upholds the safety of consensus with up to $1/3$ of the validators being Byzantine even in the asynchronous model, and maintains progress in a partially synchronous model. Since PBFT provides low latency, energy efficiency [6], and instant deterministic finality of transactions, PBFT is deemed suitable for many E-commerce applications that cannot tolerate long delays for transaction to be finalized and added to the ledger.

Unfortunately, the PBFT protocol has performance and availability problems. PBFT incurs quadratic message complexity and this curbs the scalability and performance of the consensus protocol. Secondly, PBFT leverages on a stable leader and changes it only if the leader is suspected to be Byzantine. Triggering a leader change requires a slow, costly, and prone to faults protocol which is called view change protocol.

To address these shortcomings of PBFT, blockchain systems mostly adopt rotating leader variants of PBFT. Tendermint [3] incorporates the leader rotation as part of the normal consensus path. While this adds some cost in terms of performance, it pays off in terms of fault-tolerance, availability, and fairness.

Streamlet [7] gives a two-phase rotating leader solution avoiding a lot of overhead in Tendermint. HotStuff [4] incorporates pipelining to rotation of leaders to improve throughput further. It also addresses the quadratic message complexity in PBFT and Tendermint, and provides a responsive protocol with linear complexity.

Although these rotating leader variants improve on PBFT, there has not been any study to investigate how they compare with each other and how effective different strategies for leader rotation are for alleviating bottlenecks in BFT protocols.

Contributions. In this paper, we provide a comprehensive systematic investigation of bottlenecks in deterministic finality BFT consensus protocols, namely PBFT, Tendermint, HotStuff, and Streamlet.

We take a two-pronged approach. We provide a theoretical analysis of complexity of these consensus protocols and we also implement, benchmark and evaluate them on AWS under identical conditions.

We study the bottlenecks of these consensus protocols and identify the factors that limit their scalability. Our investigations point to the communication complexity as the culprit. Even when there is enough network bandwidth, the CPU cost of serialization and deserialization of the messages limits the throughput and increases the latency of the protocols. We find that HotStuff performs significantly better than the other protocols because it (1) replaces all-to-all communication with all-to-one communication, (2) rotates the leaders at the hotspot of all-to-one communication across rounds to shed and balance load, and (3) employs pipelining across rounds to improve throughput further.

Our analysis and evaluation about the bottlenecks can pave the way for designing more efficient protocols that alleviate the identified performance bottlenecks. These analysis and evaluation results will also help researchers and developers to choose suitable consensus protocols for their needs.

Outline of the rest of the paper. After discussing the background and related work, we explain distributed consensus in Section III, and present rotated leader BFT consensus protocols in Second IV. We analyze the protocols in Section V.

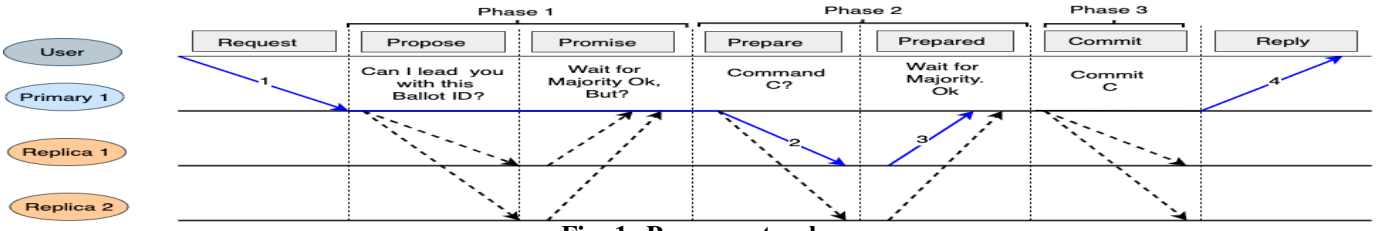

Fig. 1: Paxos protocol

We discuss our implementations in Section VI and present evaluation results in Section VII.

II. Background and Related Work

A. Background

State machine replication. State machine replication (SMR) is an abstraction employed in distributed systems for providing a fault-tolerant mechanism [8]. SMR implements a deterministic state machine that replicates on many machines for high availability and redundancy.

Byzantine fault tolerance. A Byzantine validator can behave arbitrarily, which causes inconsistency among all the validator states. BFT keeps system functioning correctly by preserving safety and liveness properties for the replicated state machines, using $N >= 3F+1$ validators, where F denotes the upper bound on the number of Byzantine validators. BFT protocols assume the existence of reliable communication channels that do not drop messages.

B. Related Work

A plethora of surveys on BFT consensus protocols in the permissioned model have come out recently, which focus on their comparisons on theoretical results. The survey [9] states that there is no perfect consensus protocol and presents their trade-offs among security and performance. A recent survey [10] provides an overview of the consensus protocols used in permissioned blockchain and investigates the algorithms with respect to their fault and resilience models. Another work [11] investigates the relationship between blockchain protocols and BFT protocols. A more recent work [12] classifies consensus protocols as proof-based and vote-based, and argues that vote-based protocols are more suitable for permissioned blockchain whereas proof of work/stake/luck based protocols are more suitable for public blockchains. There have been more exhaustive theoretical surveys [13] [14] on committee and sharding based consensus protocols. The work summarized variants of protocols, their challenges, and both their designs and their security properties.

While there has been a lot of work on consensus protocols, there has not been any work for evaluating and analyzing the performance bottlenecks in these consensus protocols. This is due to the fact that consensus protocols are more complex and not easy to implement. Motivated by this fact, we evaluate the performance of consensus protocols with finality property that work in a partial synchrony model.

III. Canonical Consensus Protocols

Paxos is widely used in research and in practice to solve decentralized consensus. Unlike the crash failure model in Paxos, the byzantine failure model is more complex and uses a number of cryptographic operations. As our best case scenario to compare consensus protocols performances, we have chosen Paxos as a performance bar to compare with other protocols instead of Raft [15] which uses in Hyperledger Fabric and has the same performance as Paxos [16].

A. Paxos

Paxos protocol [5] was introduced for achieving consensus among a set of validators in an asynchronous setup prone to crash failures. Paxos requires at least $N \geq 2F+1$ validators to tolerate the failure of F validators. By using majority quorums, Paxos ensures that there is at least one validator in common from one majority to another, and avoids the split-brain problem.

The Protocol: Paxos architecture is illustrated in Figure 1.

- A candidate leader tries to become the leader by starting a new round via broadcasting a propose message with its unique ballot number bal. The other validators acknowledge this propose message with the highest ballot they have seen so far, or reject it if they have already seen a ballot number greater than bal. Receiving any rejection fails the candidate leader.
- After collecting a majority quorum of acknowledgments, the candidate leader becomes the leader and advances to the prepare phase, where the leader chooses a value for its ballot. The value would be the value associated with the highest ballot learned in the previous phase. In the absence of any such pending proposal value, a new value is chosen by the leader. The leader asks its followers to accept the value and waits for the acknowledgment messages. Once the majority of followers acknowledge the value, it becomes anchored and cannot be revoked. Again a single rejection message nullifies the prepare phase, revokes leadership of the node, and sends it back to propose phase it cares to contend for the leadership.
- Upon successful completion of the prepare phase, the leader node broadcasts a commit message in the commit phase. This informs the followers that a majority quorum accepted the value and anchored it, so that the followers can also proceed to commit the value.

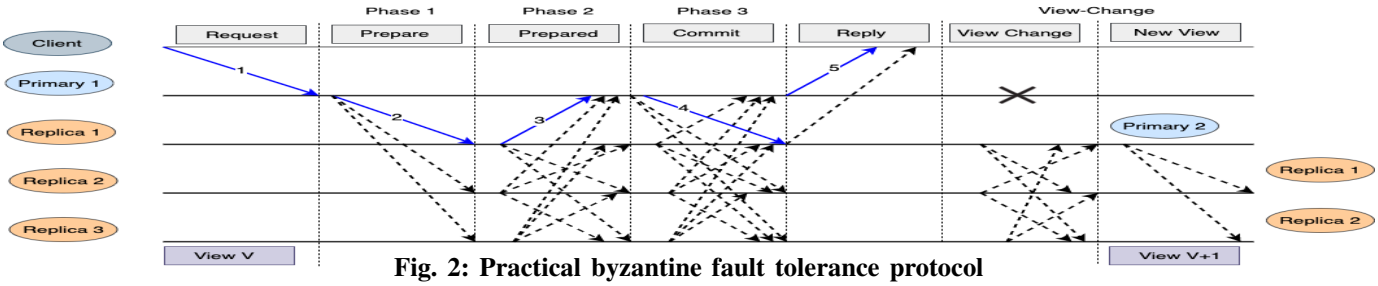

Fig. 2: Practical byzantine fault tolerance protocol

B. PBFT

PBFT protocol [2] provided the first practical solution to the Byzantine problem [1]. PBFT employs an optimal bound of $N \geq 3F+1$ validators, where the Byzantine adversaries can only control up to F validators. PBFT uses encrypted messages to prevent spoofing and replay attacks, as well as detecting corrupted messages. PBFT employs a leader-based paradigm, guarantees safety in an asynchronous model, and guarantees liveness in a partially synchronous model. When the normal path does not make progress, PBFT uses a view change protocol to elect a new leader.

The Protocol: PBFT architecture is illustrated in Figure 2.

- The leader receives the encrypted client's request and starts its prepare phase by proposing the client's request along with its view number to all followers. The followers broadcast the client's request either to acknowledge the leader or reject it if they have already seen a higher view number.
- In the absence of a rejection, each follower waits for $N-F$ matching prepared messages. This ensures that the majority of correct validators has agreed on the sequence and view numbers for the client's request.
- The followers advance to the commit phase, re-broadcast the proposal, and waits for $N-F$ matching commit messages. This guarantees the ordering across views.
- Finally, $F+1$ validators reply to the client after they commit the value.

In case of a faulty leader, a view-change protocol is triggered by the non-faulty validators that observe timer expiration or foul play. Other validators join the view change protocol if they have seen $F+1$ votes for the view change and the leader for the next view tries to take over. The new leader must decide on the latest checkpoint and ensure that non-faulty validators are caught up with the latest states. View change is an expensive and bug-prone process for even a moderate system size.

IV. Rotated Leader Protocols

In this section, we provide an overview of Tendermint, Tendermint*, Streamlet, and HotStuff BFT protocols.

A. Tendermint BFT

Tendermint protocol [3], used by Cosmos network [17], utilizes a proof-of-stake for leader election and voting on appending a new block to the chain. Tendermint rotates its leaders using a predefined leader selection function that priorities selecting a new leader based on its stake value. This function points to a proposer responsible for adding the block in blockchain. The protocol employs a locking mechanism after the first phase to prevent any malicious attempt to make validators commit different transactions at the same height of the chain. Each validator starts a new height by waiting for prepare and commit votes from $2F+1$ validators and relies on the gossip network to spread votes among all validators in both phases.

Tendermint prevents the hidden lock problem [3] by waiting for δ time. The hidden lock problem occurs because receiving $N-F$ replies from participants (up to F of which may be Byzantine) alone is not sufficient to ensure that the leader gets to see the highest lock; the highest lock value may be hidden in the other F honest nodes which the leader did not wait to hear from. Such an impatient leader may propose a lower lock value than what is accepted and this in turn may lead to a liveness violation. The rotation function that elects a next leader enables Tendermint to skip a faulty leader in an easy way that is integrated to the normal path of the protocol.

The Protocol: Tendermint protocol is illustrated in Figure 3.

- A validator becomes a leader if it has the highest stake value. It starts the prepare phase by proposing the client's request to all followers. Followers wait δ time for the leader to propose the value of the phase. If the followers find that the request came from a lower height than their current blockchain height, or that they did not receive any proposal from the leader, they gossip a nil block. Otherwise, the followers acknowledge the leader's request, then gossip the request and prepared message to other nodes.
- Upon receiving a majority of prepared messages in the prepared phase, a node locks on the current request and gossips a commit message. Otherwise, a follower rejects the prepared value and gossips the previous locked value.
- Upon receiving the majority votes in the commit phase, the nodes commit the value and reply to the client's request. Otherwise, they vote nil.
- If the leader is able to finish the view and commit the block, all validators move to the next height of the chain.

Tendermint* is a hypothetical variant of Tendermint we consider for evaluation purposes. It differs from Tendermint only in two parts. It forgoes the δ time in commit phase and the all-to-all communication in Tendermint, replacing that instead with a direct communication with just the leader. Even though the protocol violates correctness properties of BFT, we employ it in order to demonstrate which components of the protocols

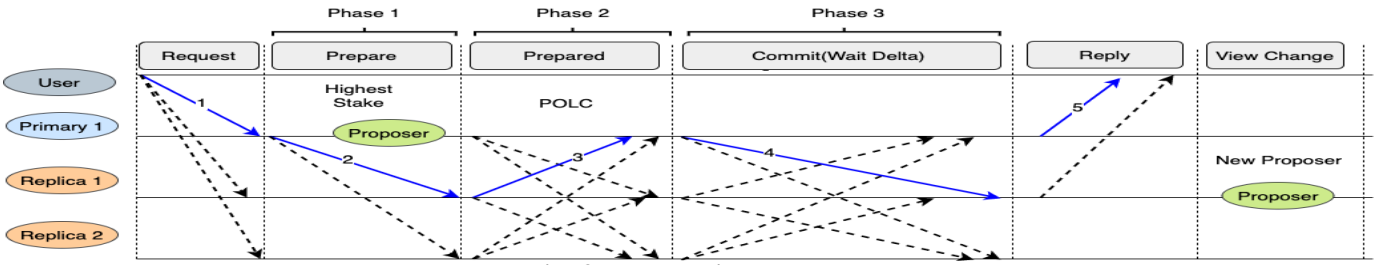

Fig. 3: Tendermint protocol

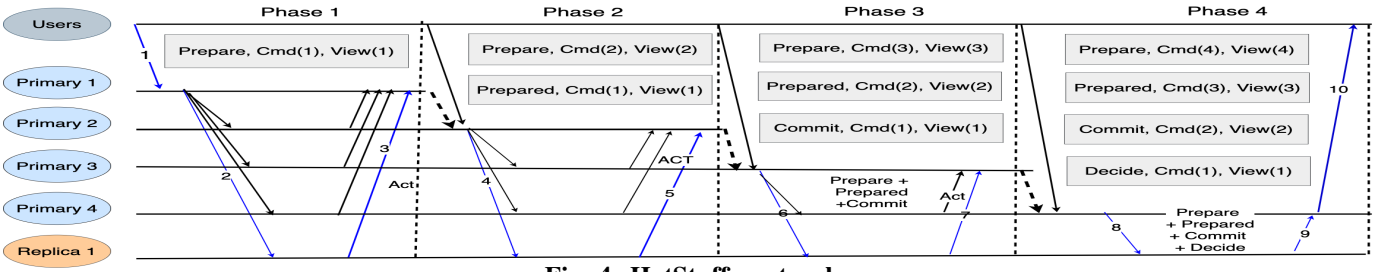

Fig. 4: HotStuff protocol

are responsible for how much performance gains/penalties and explore these in Sections VII and V.

B. HotStuff BFT

HotStuff protocol [4], is used in Facebook's Libra [18]. HotStuff rotates leaders for each block using a rotation function. HotStuff is responsive; it operates at network speed by moving to the next phase after the leader receives $N-F$ votes. This is achieved by adding a pre-commit phase to the lock-precursor. To assign data and show proof of message reception and progression, the protocol uses Quorum Certificate(QC), which is a collection of $N-F$ signatures over a leader proposal. Moreover, HotStuff uses one-to-all communication. This reduces the number of message types and communication cost to be linear. The good news is that, since all phases become the same communication-pattern, HotStuff uses pipeline mechanism and performs four leader blocks in parallel; thus improving the throughput by four.

The Protocol: HotStuff protocol is illustrated in Figure 4.

- A new leader collects new-view messages from $N-F$ followers and the highest prepare QC that each validator receives. The leader processes these messages and selects the prepare QC with the highest view. Then, the leader broadcasts the proposal in a prepare message.
- Upon receiving the prepare message from the leader, followers determine whether the proposal extends the highest prepare QC branch and has a higher view than the current one that they are locked on.
- The followers send acknowledgement back to the leader, who then starts to collect acknowledgements from $N-F$ prepare votes. Upon receiving $N-F$ votes, the leader combines them into a prepare QC and broadcasts prepare QC in pre-commit messages.
- A follower responds to the leader with a pre-commit vote. Upon successfully receiving $N-F$ pre-commit votes from followers, the leader combines them into a pre-commit QC and broadcasts them in commit messages.
- Followers respond to the leader with commit votes. Then, followers lock on the pre-commit QC. Upon successfully receiving $N-F$ commit votes from followers, the leader combines them into a commit QC and broadcasts the decide messages.
- Upon receiving a decide message, the followers execute the commands and start the next view.

HotStuff pipelines the four phase leader-based commit to a pipeline depth of four, and improves the system throughput to commit one client's request per phase. As per this pipelining, each elected leader proposes a new client request on every phase in a new view for all followers. Then, the leader simultaneously piggybacks pre-commit, commit, and decide messages for previous client requests passed on to it from the previous leader through commit certificate.

C. Streamlet BFT

Streamlet protocol proposed in 2020 [7]. Streamlet leverages the blockchain infrastructure in addition to the longest chain rule in Nakamoto protocol [19] to simplify consensus. Streamlet rotates its leader for each block using a rotation function. The protocol proceeds in consecutive and synchronized epochs where each epoch has a dedicated leader known by all validators. Each epoch has a leader-to-participants and participants-to-all communication pattern. This reduces the number of message types, but the communication cost is $O(N^2)$. Streamlet has a single mode of execution and there is no separation between the normal and the recovery mode. Streamlet guarantees safety even under an asynchronous environment with arbitrary network delays and provides liveness under synchronous assumptions.

The Protocol: Streamlet protocol is illustrated in Figure 5.

- The candidate leader for epoch(e_i) broadcasts a block that extends the longest finalized blockchain it has seen.
- Upon receiving propose message from the leader, validator nodes acknowledge the proposed block with the

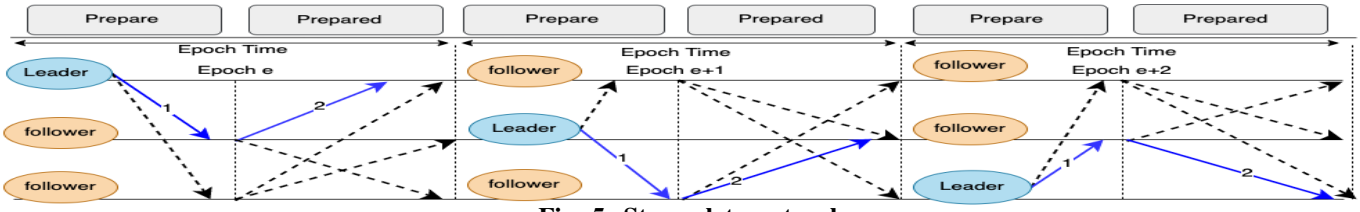

Fig. 5: Streamlet protocol

highest view number and the longest chain that they have seen so far. Then validator nodes broadcasts a vote message in the vote phase.

- Both leader and followers collect a majority quorum of acknowledgments equals to $2N/3$ for the proposal block in epoch(e_i) and mark the block as notarized block.
- If a validator node finds three consecutive notarized blocks in the blockchain(e_i, e_{i+1}, e_{i+2}), the validator node finalize up the chain.

V. Analysis and Discussion

In this section, we compare the strengths and weaknesses of the consensus protocols considered and provide back-of-the-envelope calculations for estimating the latency and throughput performance.

A. Theoretical analysis

Table I provides a synopsis of the blockchain protocols characteristics we studied. We elaborate on these next.

Synchrony Requirements. All protocols that we considered assume partially synchronous network model [20]. In this model, after a period of asynchrony, the network starts to satisfy synchrony assumptions and honest messages will be delivered within the synchronous period. Streamlet protocol assumes a synchronous clock and proceeds in synchronized epochs. Honest validators requests should be committed in the epoch time.

Time Complexity. PBFT normal execution has a quadratic complexity. When the leader is a malicious, the protocol changes the view with a different leader using a view-change which contains at least $2F+1$ signed messages. Then, a new leader broadcasts a new-view message including the proof of $2F+1$ signed view-change messages. Validators will check the new-view message and broadcast it to have a match of $2F+1$ new-view message. The view-change has then $O(N^3)$ complexity and $O(N^4)$ in a cascading failure [16].

Tendermint reduces the message complexity that is caused by view-change in PBFT, to a total $O(N^3)$ messages in the worst case. Since at each epoch all validators broadcast messages, it happens that during one epoch the protocol uses $O(N^2)$ messages. Thus, in the worst case scenario when there is F faulty validators, the message complexity is $O(N^3)$ [16].

Paxos, Tendermint*, and HotStuff all have linear message complexity. The worse case communication cost in these protocols is $O(N^2)$ considering worst-case consecutive view-changes.

Streamlet has communication message complexity $O(N^2)$. Streamlet loses linear communication complexity due to all-to-all communication in vote message. In the worst case scenario when there is a leader cascading failure, the Streamlet message complexity is $O(N^3)$.

All of the protocols provide responsiveness except for the Tendermint due to δ waiting time in commit phase and for the Streamlet due to its fixed epoch length.

B. Load and Capacity

Our considered protocols reach consensus once a quorum of participants agrees on the same decision. A quorum can be defined as sets containing majority validators in the system with every pairs of set has a non-empty intersection. To select quorums Q, quorum system has a strategy S in place to do that. The strategy leads to a load on each validator. The load $\ell(S)$ is the minimum load on the busiest validator. The capacity $Cap(S)$ is the highest number of quorum accesses that the system can possibly handle $Cap(S) = \frac{1}{\ell(S)}$ [21].

In single leader protocols, the busiest node is the leader [22].

$$\ell(S) = \frac{1}{L}(Q-1)NumQ + (1-\frac{1}{L})(Q-1)NumQ \quad (1)$$

where Q is the quorum size chosen in both leader and followers, NumQ is quorums number handled by leader/follower for every transaction, and L is the number of operation leaders. There is a $\frac{1}{L}$ chance the validator is the leader of a request. Leader communicates with $N-1 = Q$ validators. The probability of the node being a follower is $1-\frac{1}{L}$, where it only handles one received message in the best case. In the equations below, we present the simplified form of all protocols, and calculate the result for $N=9$ validators. The protocols perform better as the load decreases.

$$\ell(Paxos) = 4 \quad (2)$$

In the single leader Paxos protocol, equation 2 with N validators, and L = 1, quorum size Q = $\lfloor \frac{N}{2} \rfloor$ + 1, and number of quorums $NumQ = 1$.

The equation 3 is a single leader PBFT protocol with Q = $\lfloor \frac{2*N}{3} \rfloor$, and $NumQ = 2$.

$$\ell(PBFT) = 10 \quad (3)$$

The equation 3, PBFT III-B has high load which implies that the throughput is low. In Section VII, our evaluation illustrates how low throughput is comparing to other protocols. This is an indication how load is related to the throughput in our equation 1. PBFT bottleneck becomes quicker fast due to high load that comes form all-to-all communications.

The equation 4 is a rotated leader HotStuff protocol with a leader Q = $\lfloor \frac{2*N}{3} \rfloor$, $NumQ = 4$, pipeline = 4, and $L = N$. Unlike PBFT, HotStuff followers have no quorums. So, the $NumQ = 0$ in the followers nodes.

$$\ell(HotStuff) = (\frac{(NumQ)(L+Q-2)}{L*Pipeline}) = \frac{13}{9} \quad (4)$$

	Paxos [5]	PBFT [2]	Tendermint [3]	Tendermint* [3]	HotStuff [4]	Streamlet [7]
Synchrony	Partially synchronous					
Communicating Node	Centralized	Broadcast	Gossip	Centralized	Centralized	Broadcast
Critical Path Messages	4	5	5	8	10	4
Normal Message Complexity	$O(N)$	$O(N^2)$	$O(N^2)$	$O(N)$	$O(N)$	$O(N^2)$
Multiple View Change	$O(N^2)$	$O(N^4)$	$O(N^3)$	$O(N^2)$	$O(N^2)$	$O(N^3)$
Responsive	Yes	Yes	No	Yes	Yes	No
Failure Model	Only Crash	Byzantine	Byzantine	Byzantine	Byzantine	Byzantine

TABLE I: Characteristics of BFT consensus protocols

The equation 4, HotStuff IV-B has lowest load which implies that the throughput is high. In Section VII, our evaluation illustrates how high throughput is comparing to other protocols. This is an indication how load is related to the throughput in our equation 1. HotStuff bottleneck did not grow fast due to low load that comes form one-to-all communications and pipeline techniques.

The rest of our studied protocols are Tendermint and Streamlet. Tendermint has δ waiting time before committing the value and Streamlet is a synchronous clock. We eliminate them from our load analysis because busiest node affected not by actual workload but also by waiting time.

C. Latency

The formula 5 calculates the latency of consensus in the protocols considered, except for Streamlet which has a fixed epoch time due to its synchronous clock for each instance of consensus.

$$Latency(S) = Critical\ Path + D_L + \delta \tag{5}$$

Critical Path is the round trip message between a designated leader and its followers. Paxos's critical path has a 2-message delay as illustrated in Figure 1. With the help of a stable leader, Paxos reduces message latency in the first phase. D_L is the round trip message between a client and designated leader. In Table I, PBFT and Tendermint have a 5-message delay as illustrated in Figures 2 and 3. Paxos and Streamlet have a 4-message delay. δ refers to the waiting time that the leader has to wait before committing transactions.

As the number of validators increases, bottlenecks arise and the above latency formula starts to break down, as we see in Section VII. The reasons are different communication patterns along with different loads imposed in each consensus protocol.

VI. Implementation Framework

Our experiments are performed on the Paxi [23] framework https://github.com/ailidani/paxi. The framework is written in Go to enable evaluation of consensus protocols. Paxi supports customization of workloads and deployment conditions. The Paxi architecture is shown in Figure 6.

Upper Layer. Developers design consensus protocols and message types along with the system configurations. This layer consists of three entities: config file, message file, and validator code. The config file is distributed among all validators in JSON format, which contains all validator addresses, quorum configurations, buffer sizes, networking parameters, and benchmark parameters. The developers specify the message structures that need to be sent between validators in the message file. Finally, in the validator file, the developers write the code to handle client requests and implement the replication protocol.

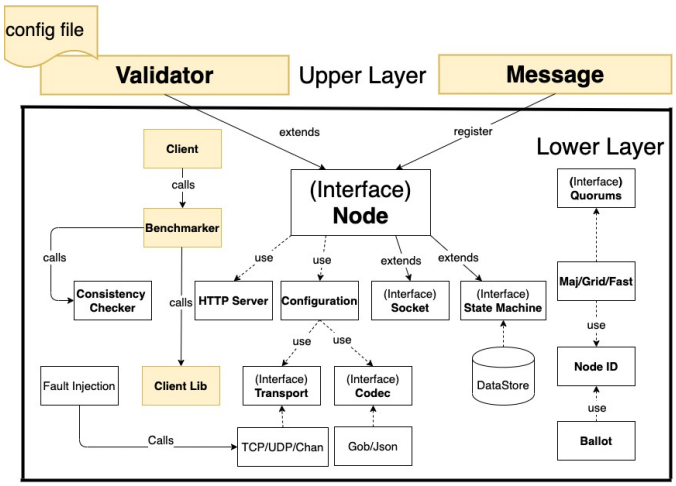

Fig. 6: The Paxi architecture

Lower Layer. The lower layer provides network implementations, multiple types of quorum systems, key-value store, client library, and benchmarker. The networking interface encapsulates a message passing model, exposes basic APIs for a variety of message exchange patterns, and transparently supports TCP, UDP, and simulated connection with Go channels. The Quorums interface provides multiple types of quorum systems. The key-value store provides an in-memory multi-version key-value datastore that is private to every node. The client library uses a RESTful API to interact with any system node for read and write requests. This allows users to run any benchmark (e.g. YCSB [24]) against their implementation in Paxi without porting the client library to other programming languages. Finally, the benchmarker component generates workloads with tunable parameters for evaluating performance and scalability.

Paxi-BFT. In order to implement BFT consensus protocols in Paxi framework, we redesigned the client library and the benchmarker module. We added the capability for the client library to send a request to all validators and to receive F+1 replies. Also, we modified the benchmark to be able to measure the latency for each request by waiting for F+1 replies. In the upper layer of Paxi, we described all BFT protocols by coding the protocols' phases, functions, and message types. Finally, in Figure 6, we highlighted the Paxi components that we designed/modified.

VII. Experimental Results

A. Experimental Setup

The experiments were conducted on AWS instances EC2 m5a.large, with 2 vCPU, 8GiB RAM. The instances are connected to each other through a 10Gbps Local Area Network(LAN). The experiments were performed with network sizes of 4, 6, 8, 10, and 12 validators. Based on our experiments results in Section VII-B, this network size is appropriate to state and conclude our findings. To push system throughput, we varied the number of clients from 1 to 90 and used a small message size. In our experiments, message size did not dominate consensus protocols performance, but the complexity of consensus protocols dominates the performance. We defined the throughput as the number of transactions per second (tx/s for short) that validator processes.

We conducted our experiments in LAN deployment. We avoided Wide Area Network(WAN) because the length of the network pipe was very large. As a result, pushing the system throughput to its limit was difficult. In LAN, pushing the system throughput to its limit to get the system bottlenecks was easy due to the short network pipe between instances.

In Tendermint, as we discussed in Section IV, the validator waits δ time before committing the block to solve hidden lock problem. This δ time includes one way message time and committing time. In Streamlet protocol, as we discussed in Section IV, the epoch time includes round trip communication time and propose-vote computing time. We set δ time in Tendermint to be 2 millisecond and epoch time in Streamlet to be 3 ms. Our experiments on AWS demonstrated that these choices of δ and epoch durations are sufficient and ensure safe execution of both protocols.

B. Evaluation Results

Paxos. We evaluated Paxos as our baseline system. Figure 7 shows that Paxos throughput declines as we increase the number of validators N. For example, when N is 4 and clients are 90, the number of transactions that the system can process is approximately 4600 tx/s. On the other hand, when N equals to 12, with the same number of clients, the system can only handle 2000 tx/s. This is due to the communication bottleneck at the single leader in Paxos [22]. The Paxos experimental result demonstrates that the load on single leader increased significantly which matches our loading Formula 2. Latency increases as N is increased because the leader struggles to communicate with more validators due to the cost of CPU being utilized in serialization/deserialization of messages.

PBFT. The throughput evaluation is shown in Figure 8. The all-to-all communication leads to a substantial throughput penalty. PBFT is also limited by a single leader communicating with the clients. When N is 4 and clients are 90, the number of transactions that the system can process is around 1500 tx/s. However, with the same number of clients, and $N = 12$, the system can only handle 600 tx/s. The PBFT experimental result shows how significant the performance bottlenecks become in comparison to Paxos. Theoretically, we captured this high load in PBFT loading Formula 3.

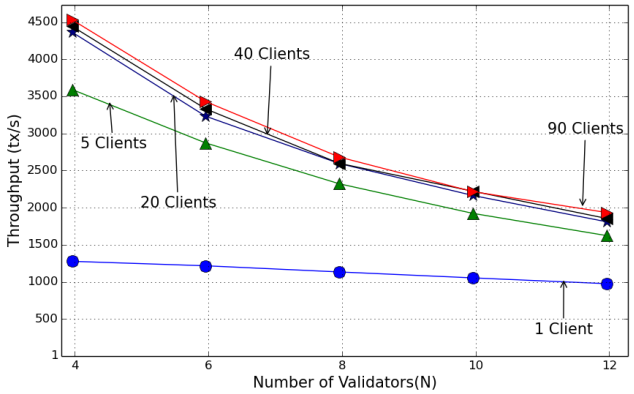

Fig. 7: Paxos throughput

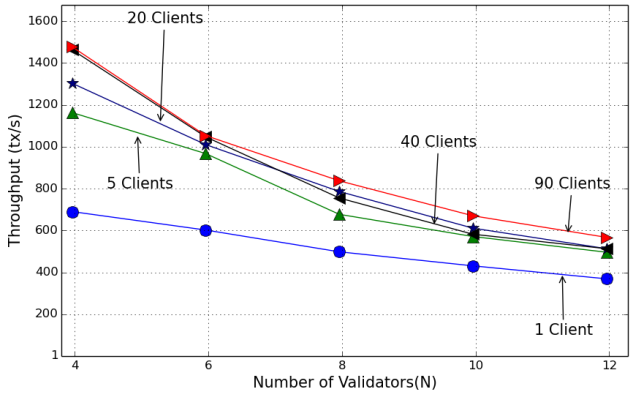

Fig. 8: PBFT throughput

Tendermint. Throughput results are shown in Figure 9. The clients are configured to communicate with all validators for all operations. Tendermint performance is bad because the protocol inherits all of the PBFT bottlenecks and tops them with waiting maximum network delay δ for solving hidden lock problem. For $N = 12$, Tendermint degrades to 200 tx/s with around half a second latency.

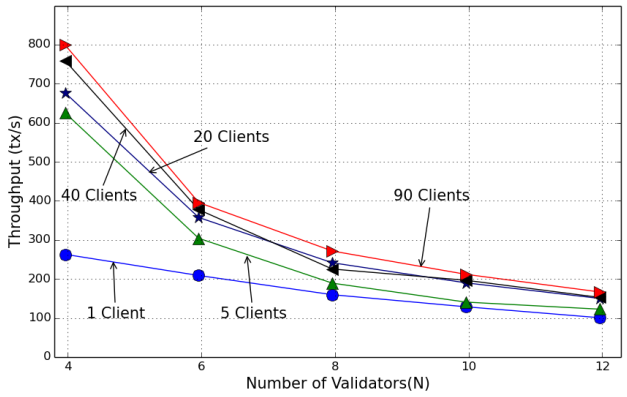

Fig. 9: Tendermint throughput

Tendermint*. The throughput is shown in Figure 11 and latency in Figure 12. Tendermint* is a hypothetical protocol that waives the all-to-all communication and the δ time delay

in Tendermint for evaluation/comparison purposes to identify those overheads. As such we can see that there is around 5 times improvement in throughput and latency in Tendermint* as compared to Tendermint.

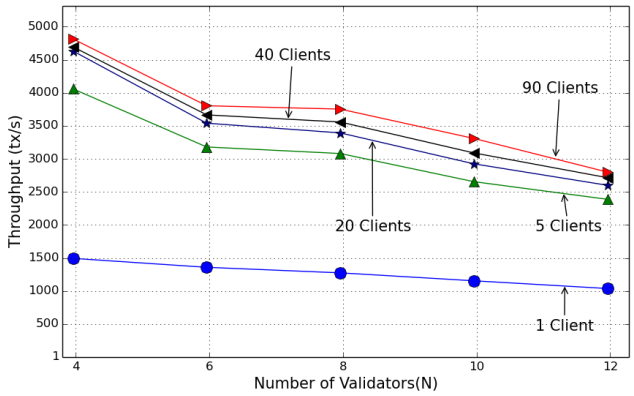

Fig. 10: HotStuff throughput

HotStuff. HotStuff achieves the best throughput compared to the other protocols, as shown in Figure 10. This is because HotStuff uses leader-to-all and all-to-leader communication, as in Paxos, and introduces pipelining of 4 different leaders' consensus slots. Compared to PBFT and Tendermint, HotStuff enables pipelining due to normalizing all the phases to have the same structure. It also adds an additional phase to each view, which causes a small amount of latency, and allows HotStuff to avoid the δ waiting time.

Streamlet. The throughput evaluation is fixed due to same size epochs. Maximum throughput limited to 330 tx/s with $epoch = 3$ ms. The synchrony clock, all-to-all communication in the second phase, and the lack of pipeline techniques result in a substantial loss in the protocol's throughput. On the other hand, the Streamlet protocol has only one phase (propose and vote), which simplifies its architecture.

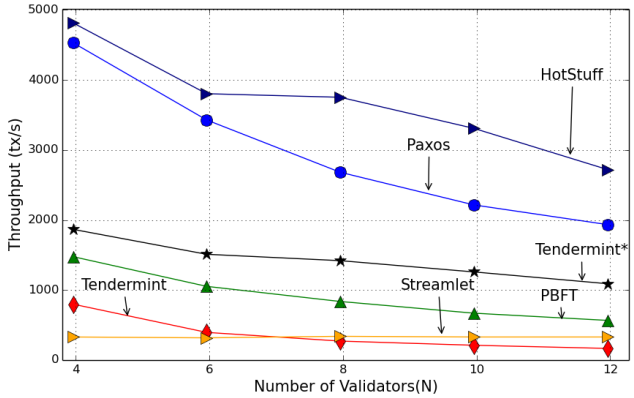

Fig. 11: Throughput comparison

C. Comparison of Throughput and Latency

In Figure 11, we discuss the throughput performance of all protocols under the same experimental conditions. The comparison in Figure 11 shows that HotStuff [4] achieves

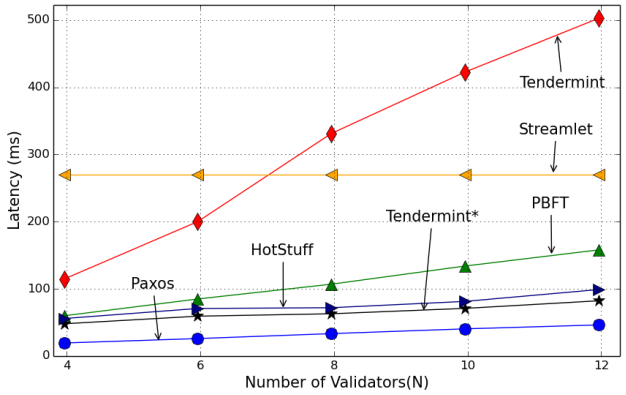

Fig. 12: Latency comparison

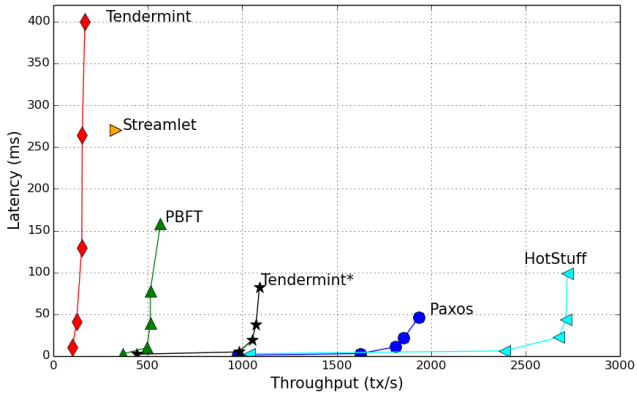

Fig. 13: The relationship between the system throughput and the latency

the maximum throughput with a large margin. This is due to responsive leader rotation and 4-leader pipelining in HotStuff. In Figure 12, we explore the average latency performance for all protocols with the same settings. Tendermint latency is the highest due to the δ wait time. In all protocols, as N increases, latency increases. This increase is more pronounced for PBFT and Tendermint, because of the all-to-all communication they employ. We also examined the relationship between the system throughput and the latency with $N = 12$ and 90 clients. The results are shown in Figure 13. The performance of BFT consensus algorithms is strongly impacted by the number of messages due to tolerance property.

VIII. Conclusion and Future Work

We studied popular deterministic-finality BFT consensus protocols. We analyzed the performance of these protocols, implemented, benchmarked, and evaluated them on AWS under identical conditions. Our results show that the throughput of these protocols do not scale well as the number of participants increases. PBFT and Tendermint suffer the most due to all-to-all communication they employ. HotStuff resolves that problem and shows improved throughput and scalability, comparable to Paxos which only provides crash fault tolerance.

We believe that this work will help developers to choose suitable consensus protocols for their needs. Our findings

about the bottlenecks can also pave the way for researchers to design more efficient protocols. As future work, we plan to adopt some bottleneck reduction techniques in non-BFT protocols such as communication relaying nodes [25] and applying them in the considered BFT protocols to improve performance.

REFERENCES

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[2] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[3] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on bft consensus," *arXiv preprint arXiv:1807.04938*, 2018.

[4] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. ACM, 2019, pp. 347–356.

[5] L. Lamport, "The part-time parliament," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 277–317.

[6] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. bft replication," in *International workshop on open problems in network security*. Springer, 2015, pp. 112–125.

[7] B. Y. Chan and E. Shi, "Streamlet: Textbook streamlined blockchains," Cryptology ePrint Archive, Report 2020/088, 2020, https://eprint.iacr.org/2020/088.

[8] F. B. Schneider, "The state machine approach: A tutorial," in *Fault-tolerant distributed computing*. Springer, 1990, pp. 18–41.

[9] A. Wahab and W. Mehmood, "Survey of consensus protocols," *arXiv preprint arXiv:1810.03357*, 2018.

[10] C. Cachin and M. Vukolić, "Blockchain consensus protocols in the wild," *arXiv preprint arXiv:1707.01873*, 2017.

[11] I. Abraham, D. Malkhi *et al.*, "The blockchain consensus layer and bft," *Bulletin of EATCS*, vol. 3, no. 123, 2017.

[12] G.-T. Nguyen and K. Kim, "A survey about consensus algorithms used in blockchain." *Journal of Information processing systems*, vol. 14, no. 1, 2018.

[13] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Sok: Consensus in the age of blockchains," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 183–198.

[14] J. Garay and A. Kiayias, "Sok: A consensus taxonomy in the blockchain era," in *Cryptographers' Track at the RSA Conference*. Springer, 2020, pp. 284–318.

[15] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319.

[16] Y. Amoussou-Guenou, A. Del Pozzo, M. Potop-Butucaru, and S. Tucci-Piergiovanni, "Dissecting tendermint," in *International Conference on Networked Systems*. Springer, 2019, pp. 166–182.

[17] J. Kwon and E. Buchman, "Cosmos: a network of distributed ledgers (2016)," *URL https://cosmos. network/whitepaper*, 2016.

[18] Facebook, "Libra framework," https://github.com/libra/libra, 2018.

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

[20] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[21] M. Naor and A. Wool, "The load, capacity, and availability of quorum systems," *SIAM Journal on Computing*, vol. 27, no. 2, pp. 423–447, 1998.

[22] A. Ailijiang, A. Charapko, and M. Demirbas, "Dissecting the performance of strongly-consistent replication protocols," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1696–1710.

[23] ——, "Paxi framework," https://github.com/ailidani/paxi, 2018.

[24] S. Busbey, "ycsb," https://github.com/brianfrankcooper/YCSB.

[25] A. Charapko, A. Ailijiang, and M. Demirbas, "Pigpaxos: Devouring the communication bottlenecks in distributed consensus," *arXiv preprint arXiv:2003.07760*, 2020.