ANALYZING AND IMPROVING PERFORMANCE IN BFT CONSENSUS PROTOCOLS

by

Salem Mohammed Alqahtani

October 2021

Advisor:

Dr. Murat Demirbas

Dissertation Committee Members:

Dr. Tevfik Kosar

Dr. Steven Y. Ko

A dissertation proposal in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering

Table of Contents

List of Tables

List of Figures

Abstract

We studied the performance and the scalability of prominent Byzantine fault tolerant(BFT) consensus protocols both analytically via load formulas and practically via implementation and evaluation. We identified that BFT consensus protocols do not scale well as the number of validators increases due to the communication bottleneck.

To address this scalability bottleneck, we propose using communication pipelining and communication aggregation techniques. We show that these techniques have a significant effect on improving system performance. We also investigate suitable communication topologies; we propose a ring topology because it provides constant communication costs between replicas and allows the use of piggybacking technique.

Chapter 1

Introduction

Blockchain systems aim to provide trustless decentralized processing and storage of transactions, immutability, and tamper-resistance. Most of the Blockchains employ BFT [43] 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) [22] protocol forms the basis for most BFT consensus protocols, such as Tendermint [16], Streamlet [24], and HotStuff [65].

PBFT builds on the Paxos [42] 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 [62], 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 [16] 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 [24] gives a two-phase rotating leader solution avoiding a lot of overhead in Tendermint. HotStuff [65] 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.

To alleviate the single leader bottleneck, multiple leader protocols were introduced [61, 8, 37, 11]. Mir-BFT is the first protocol that enables leaders to operate independently on different sequence spaces and reach consensus as long as there are no conflicts. However, the protocol inherits PBFT's message complexity and does not use pipelining techniques. Mir-BFT performance is reduced due to view change protocol.

Contributions. In this dissertation proposal, we provide a comprehensive systematic investigation of bottlenecks in deterministic finality BFT consensus protocols, namely PBFT, Tendermint, HotStuff, and Streamlet. Then, we design BigBFT, a parallel-leader BFT protocol that addresses bottlenecks and shortcomings. To further improve systems scalability, we evaluated the computation and communication performance of P2P and Ring-Allreduce system architectures of the systems via experiments. We found that the system architecture has a very significant effect on the performance of training. RA-based systems achieve scalable performance as they successfully decouple network usage from the number of workers in the system.

We take a two-pronged approach for solving system bottlenecks:

- We studied the performance and the scalability of prominent Byzantine fault tolerant(BFT) consensus protocols and identified the factors that limit their scalability. We provided a theoretical analysis of complexity of these consensus protocols. We also provided a novel framework called PaxiBFT [5] to implement, benchmark, and evaluate BFT protocols performance under identical conditions.

- To address this scalability bottleneck, we propose using communication pipelining and communication aggregation techniques. We show that these techniques have significant effect on improving system performance. We also investigate suitable communication topologies; we propose a ring topology because it provides constant communication costs between replicas and allows the use of piggybacking technique.

Our results show that BigBFT's latency is 70% better than HotStuff, 55% better than PBFT, and 65% better than Streamlet. BigBFT in WAN can provide 100% higher throughput than PBFT, and 190% higher throughput than Streamlet, and can match HotStuff's throughput. Our results also show that RA achieves high throughput and low latency compared to P2P systems. This is because, in RA the available network bandwidth is constant between replica whereas for the P2P systems, the bandwidth is a shared resources among all nodes. We also find that the RA system achieves a high overlapping between computation time and communication time than P2P systems.

1.1 Dissertation Proposal Outline

The following chapters are organized as follows. Chapter 2 provides the necessary background of State Machine Replication(SMR), leader bottleneck, cryptography, and related work. Chapter 3 provides bottlenecks in Blockchain consensus protocols. Chapter 4 introduces a BigBFT protocol for high throughput. Chapter 5 introduces a ring based system to further improve system scalability. Chapter 6 concludes this proposal and discusses future work.

Chapter 2

Background

This chapter provides the necessary background of Blockchain technology and its applications, State Machine Replication, and several influential blockchain protocols including PBFT based protocols and multi-leader protocols.

2.1 Blockchain Technology and its applications

Blockchain became widely known in 2009 with the launch of the Bitcoin [49], the first of many modern cryptocurrencies [29, 30]. Blockchain is essentially a decentralised and distributed data structure, replicated over a Peer-to-Peer (P2P) network. It mainly consists of consecutive chained blocks, each one linked with the hash of the previous, containing records that witness transactions occurred among participants, such transactions represents an asset exchange between the nodes of the network.

The number of cryptocurrencies illustrates Blockchain's importance, currently exceeding 6800 and growing[18]. Beyond cryptocurrencies, Blockchain is used in other fields with Smart Contracts playing a central role. In particular, blockchain-based systems supporting Smart Contracts enable more complex processes and interactions so they establish a new paradigm with practically limitless applications. As a result, researchers and developers are already aware of the capabilities of the new technology and explore various applications across a vast array of sectors[28, 21].

For instance, Cosmos[39] apps and services connect using the Inter-Blockchain Communication protocol that enables users to exchange assets and data across blockchains (Interoperability). Interoperability happens between multiple blockchains called zones. Cosmos zones all run on the Proof-of-Stake consensus mechanism Tendermint. One zone, called the Cosmos hub, acts as a central communication blockchain between the other zones. The Cosmos hub keeps track of all committed block headers occurring in the other zones and likewise the zones keep track of the blocks of the hub. Via Merkle proofs, zones can prove to each other the existence of messages on their respective blockchains, this way enabling interchain communication.

2.2 Leader Bottleneck

Consensus protocols rely on a strong single leader to coordinate and to order the client requests in the system. This strong leader, however, is often a bottleneck, especially when every read and write operation has to go through it. A strong leader needs to send messages to all the replicas, and receive responses to know when the operation has been successfully replicated in the state machine. In our recent work in BFT protocols, we identified and studied single-leader bottlenecks [6]. In the more common case, the leader will be bottlenecked at the CPU serializing, deserializing, and processing these messages. Too many messages are sent, received, and processed by one node. To solve leader bottlenecks in a total ordering, we alleviate the single leader bottlenecks by using multi-leaders.

2.3 Multi-Leaders

2.3.1 Mir-BFT

Mir-BFT [61] is a multi-leader consensus protocol that aims to improve the scalability and throughput of the system. Mir-BFT starts by partitioning the request hash space among all leaders to solve duplication attacks and rotates request hash space among all leaders to solve censorship attacks. It also uses batching and watermarks to facilitate concurrent proposals of batches by multiple parallel leaders. Mir-BFT proceeds in epochs and each epoch has a single primary and a set of leaders. Each leader will run an independent instance of PBFT [23]. Mir-BFT improves the performance throughput in WAN deployment and introduces a more robust BFT protocol. In terms of leaders failure, the throughput can only recover after multiple view changes discard the faulty leaders. If many leaders suspect the primary, then they timeout the epoch, and ask for epoch to be changed.

2.3.2 FnF-BFT

FnF-BFT [8] is a parallel-leader BFT consensus protocol that provides high throughput under malicious behaviors. FnF-BFT uses a Byzantine resilient performance metric to evaluate a BFT's performance. FNF-BFT has view change protocol, but has linear communication complexity during the synchrony. FnF-BFT can achieve Byzantine-resilient performance with a ratio of $16/27$ while maintaining both safety and liveness. FnF-BFT provides three properties under a stable network which are optimistic performance, Byzantine-resilient performance, and efficiency. To achieve these three properties, FnF-BFT enables all replicas to continuously act as leaders in parallel to share the load of client's requests and does not replace leaders upon failure but based on the performance history.

2.3.3 RCC

RCC [37] is concurrent leader protocol that introduced a paradigm called RCC that enables any message exchange patterns to run in parallel. The protocol requires instances to unify after each request creating a significant overhead. Additionally, the protocol relies on failure detection, which is only possible in synchronous networks. With BigBFT, we allow leaders to make progress independently of each other without any affect of failure detection.

2.3.4 Categories of BFT based systems

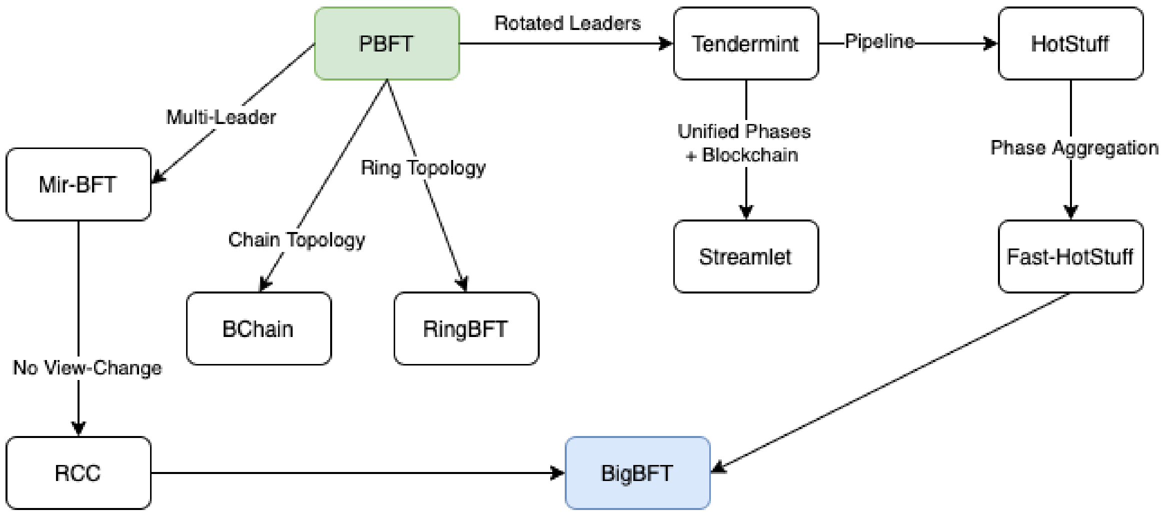

Figure 2.1: Categories of BFT based systems

2.4 Blockchain Components

2.4.1 State Machine Replication

State machine replication(SMR) is an abstraction employed in distributed systems for providing a fault-tolerant mechanism [57, 56, 44, 54]. SMR implements a deterministic state machine that replicates on many machines for high availability and redundancy of the system. The redundancy helps in increasing the system capacity and the reading throughput.

The theoretical basis of state machine replication is as follows. If each node in the cluster runs the same deterministic state machines S, which are initially all in the same initial state state_0, and each state machine is given the same data input sequence: input_1, input_2, and input_n, then these state machines would go through the same state transition path: $state\_1 \rightarrow state\_2 \rightarrow state\_n$. They would all reach the same state state_n, generating the same output sequence output_0(state_0), output_1(state_1), output_2(state_2), output_n(state_n).

In practice, SMR uses consensus protocols [54, 44, 23, 66, 16] to reach consensus among all nodes on the system state. Consensus protocols guarantee $Non-triviality$ (the decided value v was proposed by a correct node), $Safety$ (all correct nodes output the same value v), and $Liveness$ (eventually all correct nodes output some value).

The famous FLP impossibility result [34] proved that a deterministic agreement protocol in an asynchronous systems cannot guarantee liveness if one node may crash. Many consensus protocols have been proposed to circumvent the FLP impossibility to achieve an asynchronous consensus such as failure detectors, randomness, and time assumptions. One example of consensus protocol that deals with FLP impossibility result to tolerate Byzantine nodes is PBFT [23](details can refer to Section 3.1.2). PBFT guarantees safety and liveness in the partial synchronous network model [31], and this is achieved under the $1/3$ optimal resilience bound [12].

Byzantine fault tolerance. A Byzantine node can depart from the protocol and behave arbitrarily. In particular, a Byzantine node may send different messages to different nodes or not send at all, which causes inconsistency among all the node states. BFT keeps system functioning correctly by preserving safety and liveness properties for the replicated state machines. Precisely, BFT requires at least $(N >= 3F+1)$ nodes where F denotes the upper bound on the number of Byzantine faulty nodes. BFT protocols assume the existence of reliable authenticated communication channels that do not drop messages. This reliable communication can only be implemented if either a Public-key infrastructure is available for supporting the use of asymmetric cryptography for message signatures or there exists shared secrets between each pair of processes enabling the use of Message Authentication Codes (MAC). BFT protocols also employ collision-resistant hash functions, such that any node can compute a message digest $D(m)$ that detects changes to any part of the message m, where $D(m) \neq D(m')$.

Blockchain data structure. The ledger data structure is typically implemented as a linked list of blocks containing transactions stored as a Merkle tree [48]. Each block header keeps a pointer to the previous block, implemented as a cryptographic hash of the previous block. This hash serves to verify the block's identity. When this pattern is repeated in every block, we get a hash chain, as in Bitcoin, Tendermint, and HotStuff.

2.4.2 Cryptographic Hash Functions

In this part, we introduce the cryptographic algorithms used in BigBFT. We take the advantages of existing cryptographic tools that are available. Similar to many BFT protocols [66, 1, 36, 64], we assume standard digital signatures and public-key infrastructure (PKI) that identify all leader and client processes. The BigBFT's message exchange patterns combined with some cryptographic primitives to create digital signatures. BigBFT uses a signature aggregation scheme [14] that reduces the message complexity [13] and enables leaders to convert a set of signatures into a single signature; however, this can only happen when the set contains threshold value which is $N - F$ in BigBFT.

The scheme allows leaders to receive $N-F$ partial signatures $\sigma_i = sign_i(B_j)$ from every leader i for every block B_j, and combine them into a single signature $\sigma =$AggSign(sign$_i(B_j)$ $_{i \in N}$ and $_{j \in B_j}$). The leader then aggregate all σ in a single signature $AggQC$.

BigBFT uses message digest to detect corrupted messages. Both clients and leaders must be able to verify each other's leader public key and messages. We assume that all cryptographic techniques cannot be broken. We also assume cryptographic hash function $H(.)$ that maps arbitrary input to a fixed size output. We assume the hash is collision resistant where is no $H(x) == H(y)$.

Chapter 3

Bottlenecks in Blockchain Consensus Protocols

This chapter studies 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.

3.1 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 [52] which uses in Hyperledger Fabric and has the same performance as Paxos [7].

3.1.1 Paxos

Paxos protocol [42] 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.

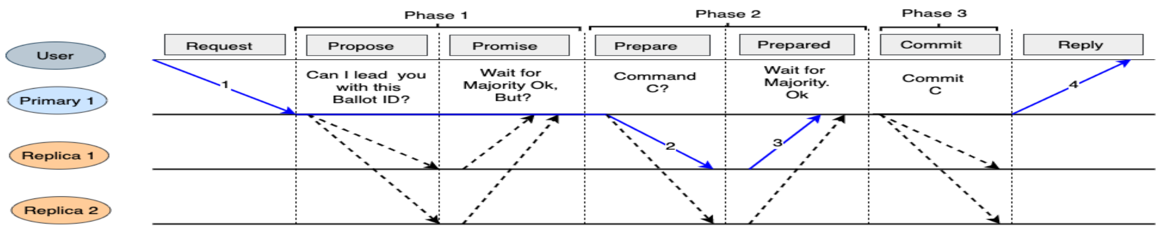

Figure 3.1: Paxos protocol

The Protocol

Paxos architecture is illustrated in Figure 3.1.

∗ 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.

3.1.2 PBFT

PBFT protocol [22] provided the first practical solution to the Byzantine problem. 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.

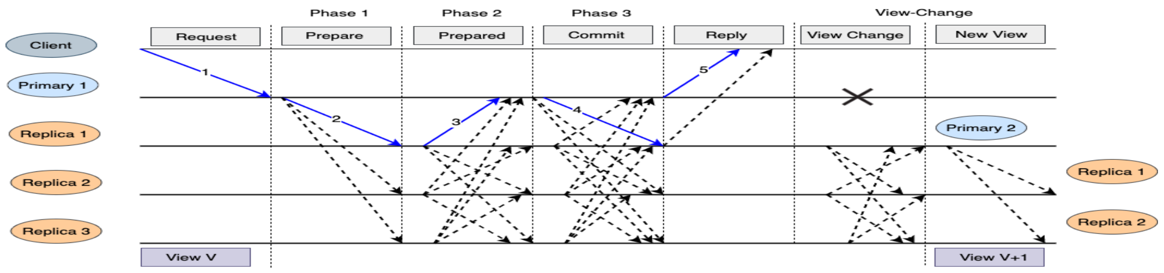

Figure 3.2: Practical byzantine fault tolerance protocol

The Protocol

PBFT architecture is illustrated in Figure 3.2.

- $*$ 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.

3.2 Rotated Leader Protocols

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

3.2.1 Tendermint BFT

Tendermint protocol [16], used by Cosmos network [40], 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 [16] 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.

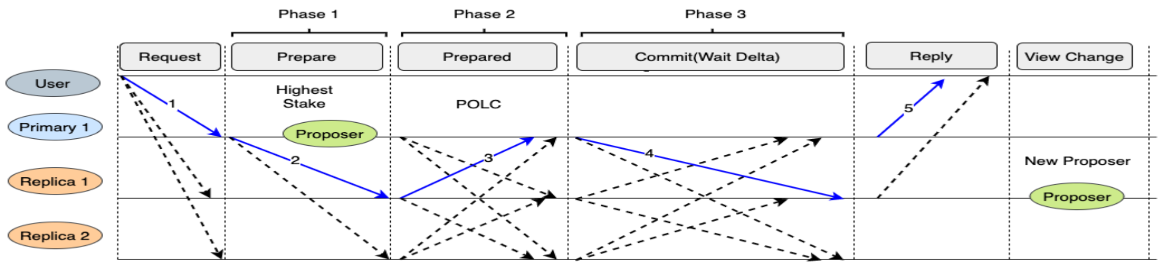

Figure 3.3: Tendermint protocol

The Protocol

Tendermint protocol is illustrated in Figure 3.3.

- 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 are responsible for how much performance gains/penalties and explore these in Sections 5.3 and 4.5.

3.2.2 HotStuff BFT

HotStuff protocol [65], is used in Facebook's Libra [33]. 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.

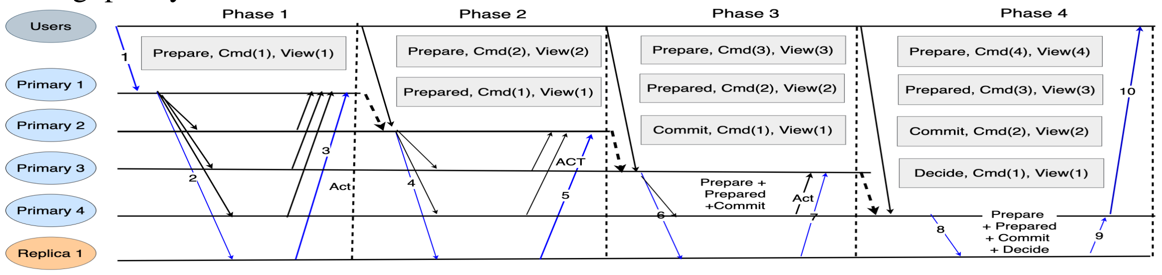

Figure 3.4: HotStuff protocol

The Protocol

HotStuff protocol is illustrated in Figure 3.4.

- 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.

3.2.3 Streamlet BFT

Streamlet protocol proposed in 2020 [24]. Streamlet leverages the blockchain infrastructure in addition to the longest chain rule in Nakamoto protocol [**bitcoin**] 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 $O(N^3)$. 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.

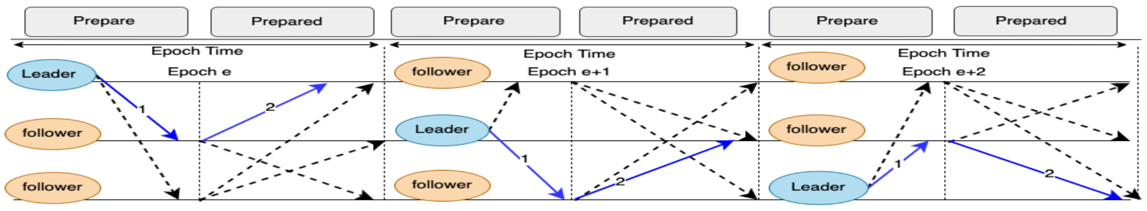

Figure 3.5: Streamlet protocol

The Protocol

Streamlet protocol is illustrated in Figure 3.5.

- 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 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.

3.3 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 performance.

3.3.1 Theoretical analysis

Table 4.1 provides a synopsis of the blockchain protocols characteristics we studied. We elaborate on these next.

	Paxos [42]	PBFT [22]	Tendermint [16]	Tendermint* [16]	HotStuff [65]	Streamlet [24]
Synchrony	Partially synchronous					
Communicating Node	Centralized	Broadcast	Gossip	Centralized	Centralized	Broadcast
Critical Path Messages	4	5	5	8	10	4
Normal Message Complexity	$O(N)$	$O(N^2)$	$O(N^2)$	$O(N)$	$O(N)$	$O(N^3)$
Multiple View Change	$O(N^2)$	$O(N^4)$	$O(N^3)$	$O(N^2)$	$O(N^2)$	$O(N^4)$
Responsive	Yes	Yes	No	Yes	Yes	No

Table 3.1: Characteristics of BFT consensus protocols

Synchrony Requirements. All protocols that we considered assume partially synchronous network model [32]. 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.

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 [7].

Tendermint reduces PBFT's message complexity to $O(N^3)$ in the worst case. Since at each epoch all validators broadcast messages, the protocol uses $O(N^2)$ messages. Thus, in the worst case scenario when there is F faulty validators, the message complexity is $O(N^3)$ [7].

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

Streamlet has message complexity $O(N^3)$. Streamlet loses linear communication complexity due to all-to-all communication in vote message. In the worst case when there is a leader cascading failure, the Streamlet complexity is $O(N^4)$.

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.

3.3.2 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 decides which quorums types to choose that 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 $Cap(S) = \frac{1}{\ell(S)}$ [50].

In single leader protocols, the busiest node is the leader [3].

$$\ell(S) = \frac{1}{L}(Q-1)NumQ + (1-\frac{1}{L})(Q-1)NumQ \tag{3.1}$$

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 and we assume $N = 9$. The probability of the node being a follower is $1 - \frac{1}{L}$, where it only handles one received message in the best case. The protocols perform better as the load decreases.

$$\ell(Paxos) = 4 \tag{3.2}$$

In Paxos, equation 4.2 with L = 1, quorum size Q = $\lfloor \frac{N}{2} \rfloor$ + 1, and number of quorums $NumQ = 1$. The equation 3.3 is a PBFT protocol with Q = $\lfloor \frac{2*N}{3} \rfloor$, and $NumQ = 2$.

$$\ell(PBFT) = 10 \tag{3.3}$$

The equation 3.3, PBFT 3.1.2 has high load which implies that the throughput is low. In Section 5.3, 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 4.1. PBFT bottleneck becomes quicker fast due to high load that comes form all-to-all communications.

The equation 3.4 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.

$$\ell(HotStuff) = 5 \tag{3.4}$$

The equation 3.4, HotStuff 3.2.2 has lowest load which implies that the throughput is high. In Section 5.3, 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 4.1. HotStuff bottleneck did not grow fast due to low load that comes form one-to-all communications and pipeline techniques.

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.

3.3.3 Latency

The formula 4.3 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.

$$Latency(S) = Critical\ Path + D_L + \delta \tag{3.5}$$

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 3.1. 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 4.1, PBFT and Tendermint have a 5-message delay as illustrated in Figures 3.2 and 3.3. 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 5.3. The reasons are different communication patterns along with different loads imposed on protocols.

3.4 PaxiBFT Framework

Our experiments are performed on the PaxiBFT [5] framework [1] written in Go. PaxiBFT enables evaluation of BFT consensus protocols and supports both customization of workloads and deployment conditions. The PaxiBFT architecture is shown in Figure 3.6. The PaxiBFT's purpose is to offer a fair environment for comparing BFT protocols.

To implement BFT consensus protocols in PaxiBFT framework, we designed BFT client library, benchmarker, message handling modules. For the network infrastructures, we borrowed the core network implementations from Paxi framework [4]. The client library can send a request to all validators and receive $F+1$ replies. We also enable the benchmark to be able to measure the latency for each request.

In PaxiBFT, all BFT protocols can be implemented by coding the protocols' phases, functions, and message types. In Figure 3.6, we highlighted some important components that can be modified by developers to implement new BFT protocol.

The top layer of PaxiBFT framework consists of config file, message file, and validator code. The config file is distributed among all validators in JSON format, which contains

[1] https://github.com/salemmohammed/PaxiBFT

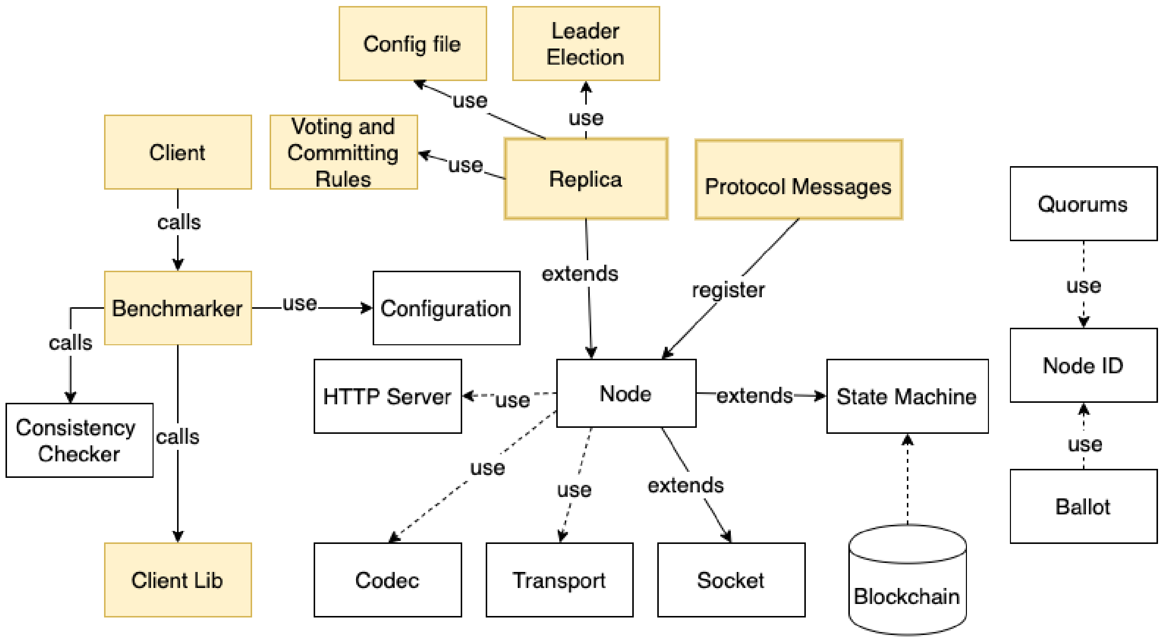

Figure 3.6: The PaxiBFT architecture

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.

In lower layer of PaxiBFT framework, the core network network implementations as we mentioned earlier borrowed from Paxi framework [4]. 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 [17]) 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.

3.5 Experimental Results

3.5.1 Experimental Setup

The experiments were conducted on AWS instances EC2 m5a.large, with 2 vCPU. The experiments were performed with network sizes of 4 to 20 nodes. Based on our experiments results in Section 3.5.2, this network size is appropriate to state and conclude our findings. To push system throughput, we varied the number of clients up 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 and Wide Area Network(WAN) across 4 AWS regions(Ohio, N.California, Oregon, and N.Virginia). In WAN, pushing the system throughput to its limit to get the system bottlenecks was difficult while it was easy in LAN due to the short network pipe between instances.

In Tendermint, as we discussed in Section 3.2, 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 3.2, the epoch time includes round trip communication time and propose-vote computing time. In LAN, We set δ time in Tendermint to be 2 ms and epoch time in Streamlet to be 3 ms. In WAN, We set δ and epoch time in Streamlet to be 50 ms. Our experiments show that these choices of δ and epoch durations are sufficient and ensure safe execution of both protocols.

3.5.2 Evaluation Results

Paxos. We evaluated Paxos as our baseline system. Figure 3.7 and Figure 3.8 show 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 4900 tx/s. On the other hand, when N equals to 16, with the same number of clients, the system can only handle 1500 tx/s. This is due to the communication bottleneck at the single leader in Paxos [3]. The Paxos experimental result demonstrates that the load on single leader increased significantly which matches our loading Formula 4.2.

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 3.7 and Figure 3.8. 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 1750 tx/s in LAN and 870 tx/s in WAN. However, with the same number of clients, and $N = 16$, the system can only handle around 500 tx/s and 350 tx/s in WAN. 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 3.3.

Tendermint. Throughput results are shown in Figure 3.7 and Figure 3.8. 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 $N = 16$, Tendermint degrades to 150 tx/s in LAN and around 90 tx/s in WAN.

Tendermint*. The throughput is shown in both Figure 3.7 and Figure 3.8, and latency in Figure 3.9. 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 4 times improvement in throughput and latency in Tendermint* as compared to Tendermint.

HotStuff. HotStuff achieves the best throughput compared to the other protocols, as shown in Figure 3.7 and Figure 3.8. This is because HotStuff uses leader-to-all and all-

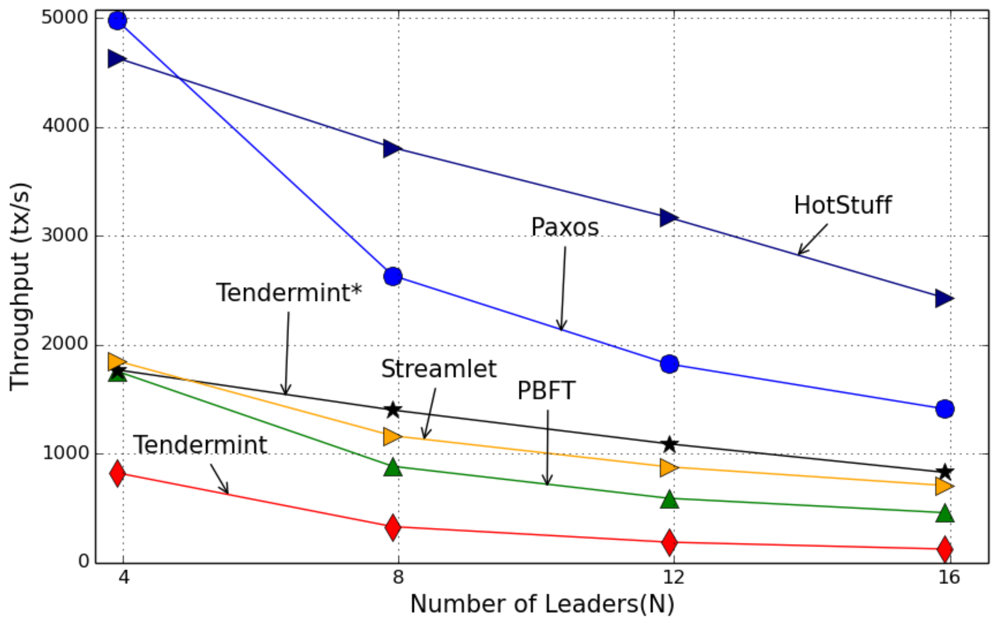

Figure 3.7: Throughput comparison in LAN

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 maximum throughput is around 700 tx/s with $epoch = 3$ ms while 300 tx/s with $epoch = 50$ ms in WAN. 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.

3.5.3 Comparison of Throughput and Latency

In Figures 3.7 and 3.8, we discuss the protocols' throughput performance under the same experimental conditions. In both Figures, HotStuff achieves the maximum throughput in LAN and is close to Paxos in WAN deployment. This is due to responsive leader rotation

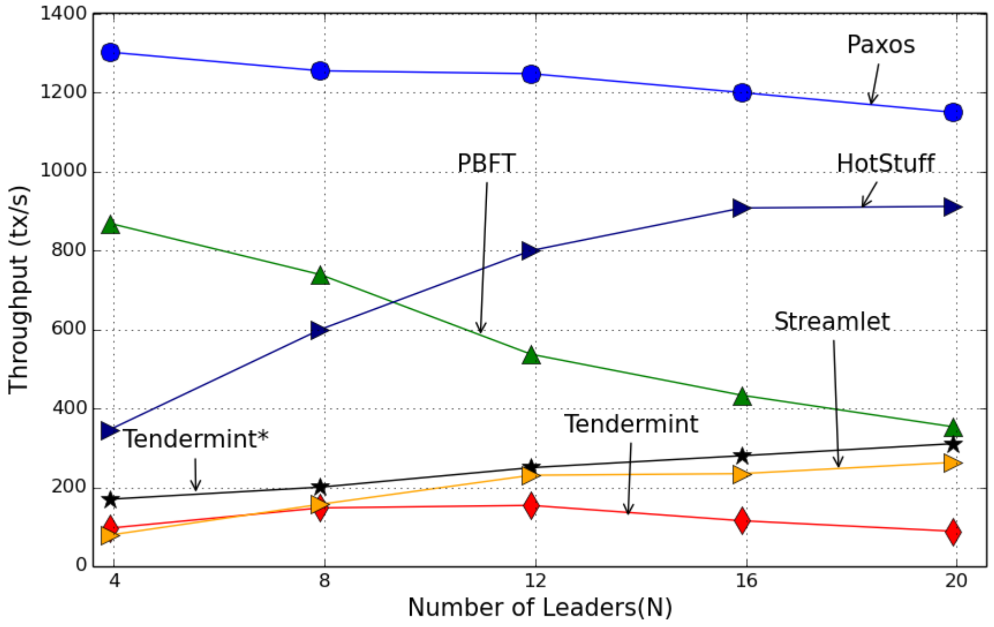

Figure 3.8: WAN's throughput comparison in Virginia, California, Oregon, and Ohio

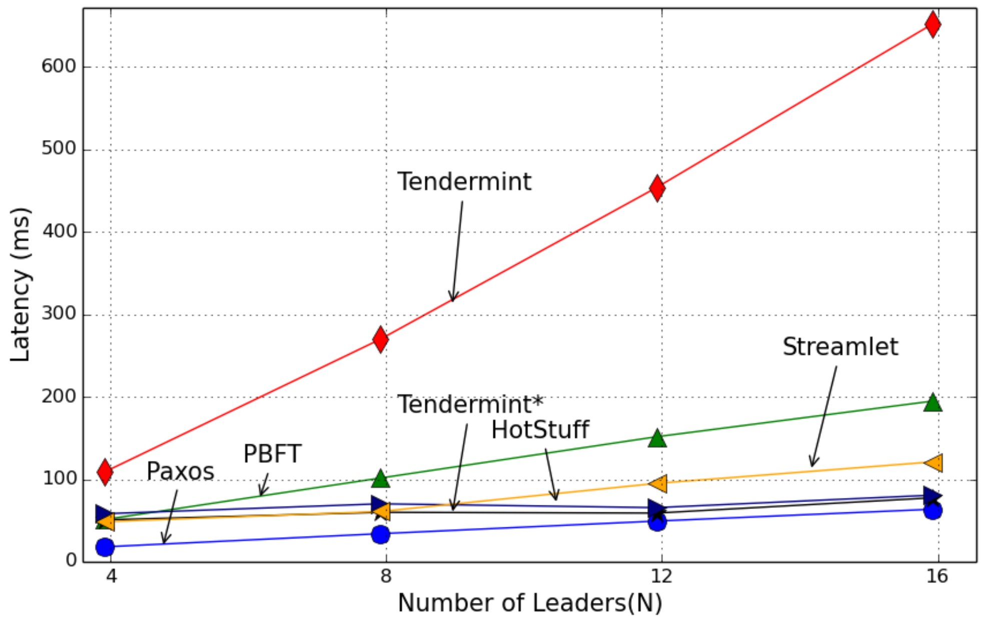

Figure 3.9: Latency comparison in LAN

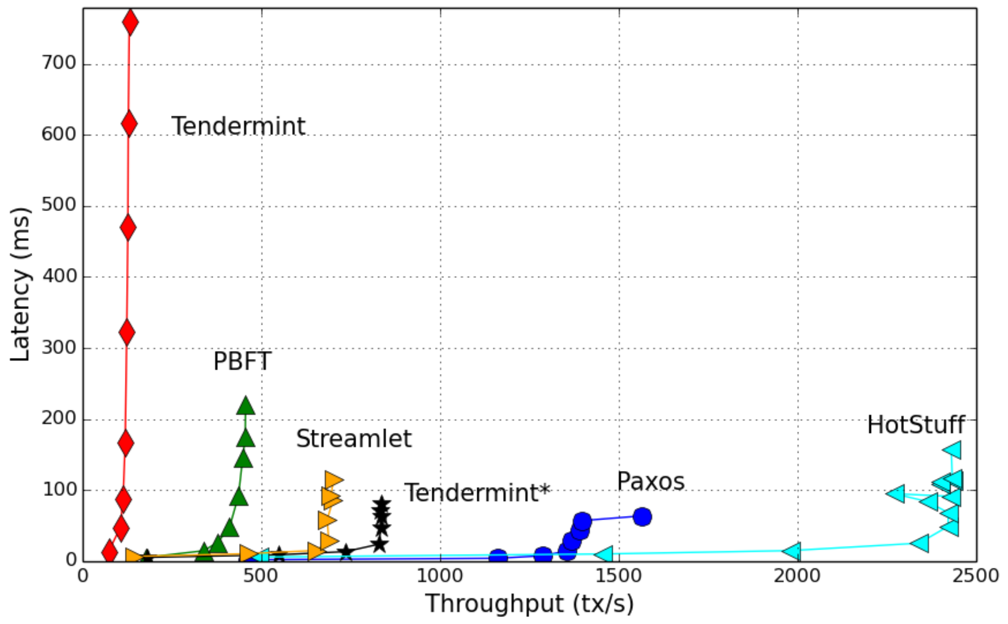

Figure 3.10: System throughput and the latency on 20-node LAN cluster

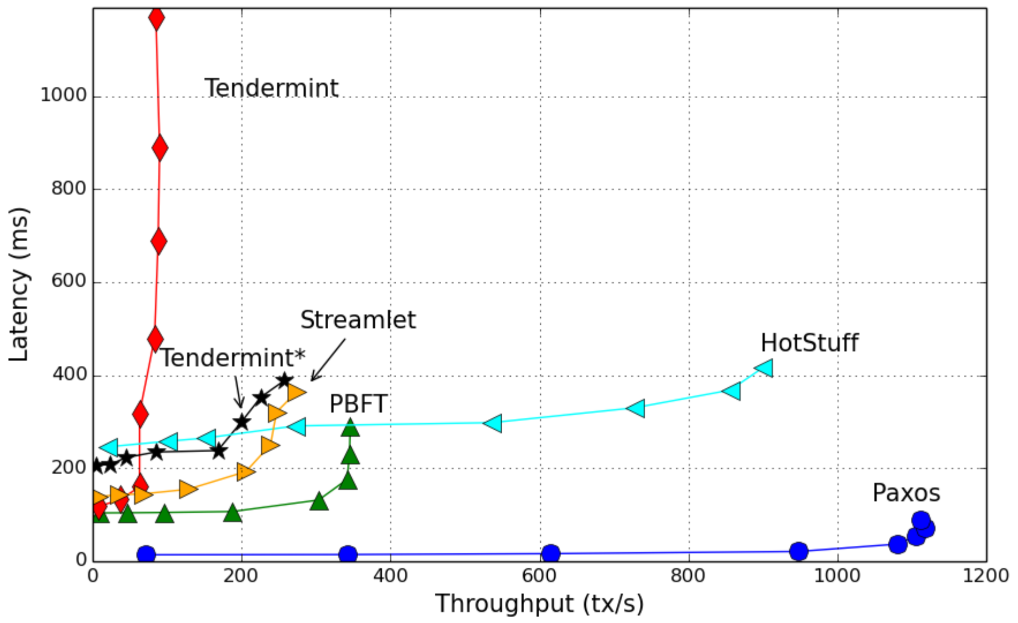

Figure 3.11: System throughput and the latency on 20-node WAN cluster in Virginia, California, Oregon, and Ohio

and 4-leader pipelining in HotStuff. In Figure 3.9, 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 in WAN and LAN with $N=20$ and 90 clients. The results are shown in Figure 3.10 and 3.11. The performance of BFT consensus algorithms is strongly impacted by the number of messages due to tolerance property.

3.6 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 [63] states that there is no perfect consensus protocol and presents their trade-offs among security and performance. A recent survey [20] 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 [2] investigates the relationship between blockchain protocols and BFT protocols. A more recent work [51] 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 such as [10] 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.

3.7 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 such as communication relaying nodes [27] and applying them in the considered BFT protocols to improve performance.

Chapter 4

BigBFT: A Multileader Byzantine Fault Tolerance Protocol for High Throughput

This chapter describes BigBFT, a multi-leader Byzantine fault tolerance protocol that achieves high throughput and scalable consensus in blockchain systems. BigBFT achieves this by (1) enabling every node to be a leader that can propose and order the blocks in parallel, (2) piggybacking votes within rounds, (3) pipelining blocks across rounds, and (4) using only two communication steps to order blocks in the common case.

BigBFT has an amortized communication cost of $O(n)$ over n requests. We evaluate BigBFT's performance both analytically, using back-of-the-envelope load formulas to construct a cost analysis, and also empirically by implementing it in our PaxiBFT framework. Our evaluation compares BigBFT with PBFT, Tendermint, Streamlet, and Hotstuff under various workloads using deployments of 4 to 20 nodes. Our results show that BigBFT outperforms PBFT, Tendermint, Streamlet, and Hotstuff protocols either in terms of latency (by up to 70%) or in terms of throughput (by up to 190%).

4.1 Preliminaries

4.1.1 System Model

We consider a permissioned blockchain system with arbitrary number of clients and a finite set of leader nodes $N = 3F+1$, indexed by $i \in \{1, ..., n\}$, where leaders can tolerate up to F Byzantine leaders. Leaders maybe in multiple geographical locations and have both different speeds and different physical machines. All leaders communicate and synchronize by sending and receiving messages through reliable channels. A correct leader follows its specification while Byzantine leaders control by an adversary and behave arbitrary including sending wrong messages and colluding with each other to harm the system. A computationally bounded adversary can control the faulty nodes to compromise the system if more than F compromised.

BigBFT protocol runs in rounds and each round consists of two phases called propose and vote phases. We assume that at the beginning of a round, every leader knows all other leaders in the round. The coordination phase is not on the critical path of the protocol. In each round, we assumes a unique coordinator is known to every leader for every round. The coordinator node can be one of the leaders. We also assume a round-robin rotation for coordinator elections as in many BFT protocols [25, 66, 16, 37, 8, 61].

PBFT has a complicated view-change sub-protocol with a quadratic message complexity. Chain based protocols [16, 15, 19, 66, 25, 60] have emerged recently that simplify view-change sub-protocol to have linear message complexity. However, chain based protocols requires sequential order to propose and commit proposals which affect concurrency executions, which lead to low resource utilizations. Fortunately, BigBFT coordination phase is not on the critical path of the protocol and both BigBFT and its coordination phase are working in parallel to maximize resource utilizations.

4.1.2 Communication Model

Following common practice in the literature, we assume a partial synchrony communication model in our protocol [31], as most BFT protocols of the same kind [23, 25, 26, 36, 66], where there is a known network delay bound δ that will hold after an unknown Global Stabilization Time (GST). After GST, all messages between honest leaders will arrive within time δ. When an honest leader sends a message in round r, an honest recipient leader is guaranteed to receive it by the beginning of round $(r + \delta)$. Although we assume partial synchrony, the protocol achieves consistency (i.e., safety) regardless of how long the message delays are or how badly the network might be partitioned. The protocol executes in parallel with a linear communication complexity over N nodes. Also, the protocol maintains responsiveness which proceeds as network delivers [66, 38].

4.1.3 Cryptographic Primitives

In this part, we introduce the cryptographic algorithms used in BigBFT. We take the advantages of existing cryptographic tools that are available. Similar to many BFT protocols [66, 1, 36, 64], we assume standard digital signatures and public-key infrastructure (PKI) that identify all leader and client processes. The BigBFT's message exchange patterns combined with some cryptographic primitives to create digital signatures. BigBFT uses a signature aggregation scheme [14] that reduces the message complexity [13] and enables leaders to convert a set of signatures into a single signature; however, this can only happen when the set contains threshold value which is $N - F$ in BigBFT.

The scheme allows leaders to receive $N - F$ partial signatures $\sigma_i = sign_i(B_j)$ from every leader i for every block B_j, and combine them into a single signature σ =AggSign(sign$_i(B_j)$ $_{i \in N}$ and $_{j \in B_j}$). The leader then aggregate all σ in a single signature $AggQC$.

BigBFT uses message digest to detect corrupted messages. Both clients and leaders must be able to verify each other's leader public key and messages. We assume that all cryptographic techniques cannot be broken. We also assume cryptographic hash function $H(.)$ that maps arbitrary input to a fixed size output. We assume the hash is collision resistant where is no $H(x) == H(y)$.

4.2 BigBFT Protocol

4.2.1 Overview

BigBFT protocol has a designated coordinator C_r that has chosen in a round-robin fashion for leading the round r, managing the leader set L_s, and partitioning the set of sequence numbers $\mathbb{Z}$ between leaders to avoid conflicts between leaders, $Partition_{ki} \leftarrow \mathbb{Z}/L_s$. The $Partition_{k \in \mathbb{Z} \text{ and } i \in L_s}$ means a partition k assigned to a leader i from the leader set L_s.

BigBFT executes in rounds $r = [0, +\infty)$ where each round has a dedicated leader set and a coordinator known to all. The communication patterns when the network is synchronous, no byzantine failures, and no contention, are two communication phases in a normal-case: (i) leaders propose client requests in parallel(Phase-1). (ii) vote on client requests in parallel(Phase-2). In Figure 4.1, we illustrate the communication flow of BigBFT protocol. The BigBFT would only perform coordination phase(round-change) after many consensus instances in parallel with BigBFT protocol(Phase-1 and Phase-2). Coordination phase invokes to replace a Byzantine coordinator, a leader set, and a new partition of the sequence numbers $\mathbb{Z}$.

Any full node in the system have two roles called coordinator (C) and leader (L). C is responsible for leading the round and partitioning the sequence number space. L is responsible to receive a partition space, proposing blocks and voting on blocks. These roles can be co-located, that is, a single process can be a coordinator or a leader.

Clients submit their requests to $F + 1$ leaders in the system that are responsible to handle the client request. In this case, non-faulty process can learn the request if the faulty leader tries to prevent that request from being in the next proposal. BigBFT's $client$ can

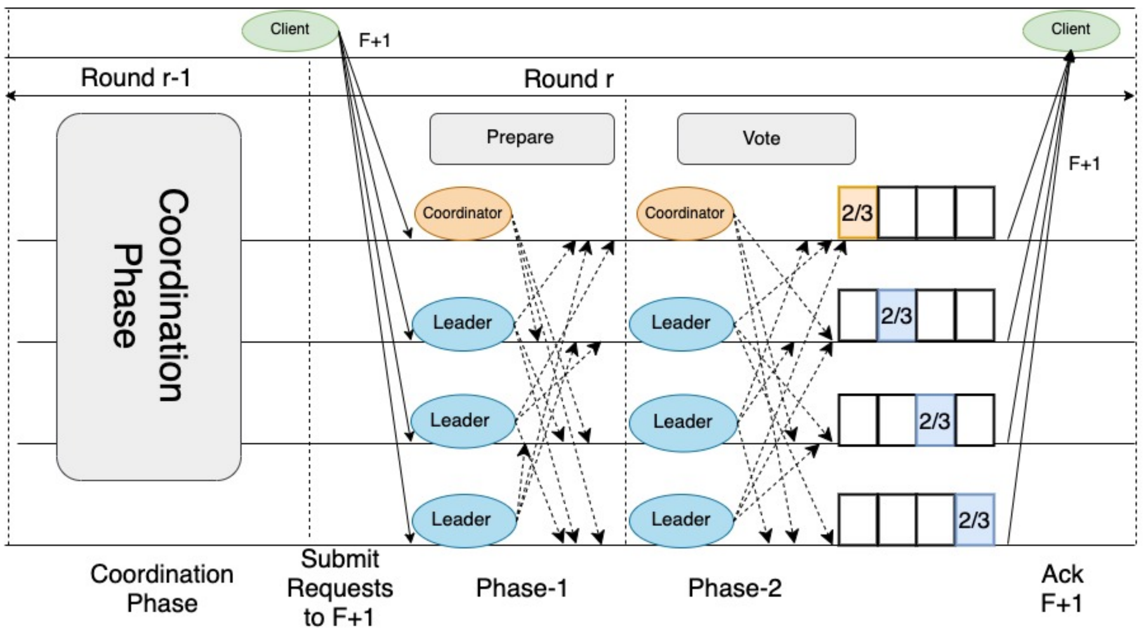

Figure 4.1: Communication pattern of BigBFT

send many independent requests on the fly to the leaders $\langle Request, t, O, id\rangle$, where t is the timestamp, O is the operation, id is the client id. The client will receive $\langle Reply, r, t, L\rangle$, where r is the round number and L is the leaders identifications who executed the client request.

4.2.2 Data Structures

This part introduces message types and the structure of the block in BigBFT before presenting the details of the protocol. As shown in the Algorithm 2, $RChange$ is the round change message to change the current round that carries a new round number $r = r + 1$ where r is the current round, space partitions $\mathbb{Z}$, and a leader set L_s.

$Prepare$ message in the Algorithm 3 carries the digest message of block B_j, the sequence number sn, the round r, and the previous round $AggQC_{r-1}$. The leaders always use $AggQC_{r-1}$ to proof the last blocks commit to propose a new block. $Vote$ message in the Algorithm 3 carries a set of partial signatures of blocks for B_j where $j \in B_j$. An aggregate quorum consists of set of QCs that each QC contains $N - F$ signed votes for a

Algorithm 1: Utilities for leader$_i$

```
Function Propose(type, AggQC, Block):
    $p.type \leftarrow type$
    $p.AggQC \leftarrow AggQC$
    $p.Block \leftarrow Block$
    return p
Function Vote(type, $\{votes\}$, $r$):
    $v.type \leftarrow type$
    $v.r \leftarrow r$
    $v.\{votes\} \leftarrow \{votes\}$
    return v
Function RoundMsg(type, $\mathbb{Z}$, $r$, $L$):
    $RC.type \leftarrow type$
    $RC.r \leftarrow r$
    while $k \in \mathbb{Z}_k$ and $i \in L_i$ do
        $RC.Partition_{k,i}$
    return RC
```

block from distinct leaders.

4.2.3 Coordination Phase

We design the coordination phase as a separate phase of BigBFT to avoid any leader bottlenecks. The coordination phase partitions the sequence space across replicas in r and prepares leader set L_s for next round $r + 1$. In each round, the new coordinator increases the round number by one and starts the round, $C_{i+1} \leftarrow next.C_i$. As described in Algorithm 2, the coordinator C_{i+1} sends round-change $RChange$ message to all leaders. Each leader receives and processes the $RChange$ against its state. Then, each leader signs the coordinator message $RChange$ and sends the reply back to coordinator. Upon receiving $N - F$ replies from leaders, the coordinator creates $RoundQC$ quorum and broadcasts it to all leaders in round r. The coordination phase executes in parallel with BigBFT 4.2.4 of previous round $r - 1$. After the execution of coordination phase, each leader will have a designated partition of sequence numbers assigned by the coordinator.

Algorithm 2: Coordination phase of BigBFT

```
▷ Coordination phase
foreach $r \leftarrow 0, 1, 2...$ do
    if $C_i$ is coordinator then
        RC $\leftarrow RoundMsg(RChange, \mathbb{Z}, r, L)$
        Broadcast RC
    if $L_i$ is leader then
        if RC is received from $C_i$ then
            if ( $r$ >= local r ) then
                $\sigma_i = Sign(Ack, sk_i)$
                // sk = secret key of leader i
                Send $\sigma_i$ to $C_i$
    if $C_i$ received (N-F $\sigma_i$ where $i \in N$ ) in r then
        $RoundQC \leftarrow$ combine N-F $\sigma_i$ where $i \in N$
        broadcast RoundQC to N-1
    if ($L_i$ receives an RoundQC from $C_i$) then
        Call Algorithm 3
```

Algorithm 3: BigBFT protocol

```
for each r $\leftarrow 0, 1, 2...$ do
    ▷ Prepare phase of $L_i$
    $l_i\langle$ waits for $B_j\rangle$ from client Then
    if $B_j == valid$ then
        $msg = Propose($prepare$, AggQC_{r-1}, B_j)$
        $broadcast(msg, d(B_j), r)$
    ▷ $L_i$ receives prepare Msgs
    if receives N-F prepare msgs then
        // Same block cannot be assign to more than one leader
        while $r = true$ do
            if $B_j.L_i == B_j.L_i + 1$ then
                $r \leftarrow$ False
        if $AggQC_{r-1} == true$ then
            $((N - F)B_j \in r - 1) \leftarrow$ committed
    ▷ Send Vote Phase of $L_i$
    if $L_i$ received (N-F $B_j$ where $j \in N$) in r then
        $\{\sigma\} = \bigcup_{j=1}^{N-F} \sigma_j = Sign_i(B_j)$
        broadcast Vote(vote, $\{\sigma\}$ , r)
    ▷ $L_i$ receives Vote Msgs
    $QC(B_j) \leftarrow \bigcup_{i=1}^{N-F} \sigma_i$
    $AggQC_r \leftarrow \bigcup_{i=1}^{N-F}$ QC$_i$
```

4.2.4 BigBFT protocol

After partitioning the set of sequence numbers across leaders in coordination phase, BigBFT starts two communication phases called prepare and vote. Below we describe how prepare and vote phases works.

Prepare Phase. Upon receiving the block in round r, each leader assigns the next available sequence number to the new block. The leader also attaches the proof of vote $AggQC$ from previous round $r-1$ in prepare message. Then, in prepare phase, each leader proposes the prepare message to all leaders.

In normal path, the leaders receive $N-F$ proposed blocks that do not conflict with any other blocks. Every leader checks the $AggQC_{r-1}$ in order to commit the blocks from previous round $r-1$. If the leader is faulty, the BigBFT guarantees that the faulty leader affect only its process and pending blocks go to other honest leaders.

Vote Phase. Upon receiving $N-F$ proposed blocks, the leader signs each block and sends the vote message to all leaders. To reduce the message complexity in the vote phase, instead of sending vote messages $N-F$ times, we combine all vote messages in a single vote set $\{votes\}$ and send it to all leaders. Each vote message represents a partial signature that signed by replica i for a block. Because we have $N-F$ blocks, we need to combined $N-F$ signatures $\{\sigma\}$ in one vote message.

We describe the BigBFT protocol as follow.

1. Client nodes broadcast their blocks/requests to $F+1$ leaders including the coordinator.

2. Upon receiving new block proposal from clients, every leader verifies the block, assigns sequence number to the new block, and attaches the proof of vote $AggQC$ from previous round in the prepare message.

3. Leader broadcasts prepare message to all other leaders.

4. Upon receiving prepare messages and commit $AggQC_{r-1}$, leaders commit previous round blocks. Leaders sign each proposal message and combine all signatures in a vote set $\{votes\}$.

5. Upon receiving the $N - F$ vote set messages, leader create a quorum certificate for every block and combined all quorum certificate in aggregated quorum certificate $AggQC$, Algorithm 3.

6. In the next round $r + 1$, leader nodes check the new propose blocks. If the prepare message carries the $AggQC_r$, then it commit the blocks and reply to the client. This is what we have called across rounds pipelining.

Importantly, the entire protocol follows a unified coordination-prepare-vote paradigm. The coordination protocol is pipelined and not on the critical path of BigBFT.

4.3 BigBFT Correctness

We prove that BigBFT achieves both safety and liveness properties by showing BigBFT algorithm solves agreement, validity, and termination in all possible distributed executions. We also performed model checking of the high level BigBFT protocol in TLA^+ [41]. The specification is available on the GitHub.[1]

4.3.1 Safety

BigBFT guarantees its safety in any circumstances regardless of the network delays and partitions. If there are less than $F < \frac{N}{3}$ Byzantine leaders and $N - F$ honest leaders decide on blocks $B_{j \in (j, j+N-1)}$ at blockchain height {h where $h = h$ to $h = h + N - 1$}, then no honest leader will decide on any blocks other than B_j.

[1]`https://github.com/salemmohammed/BigBFT/tree/main/tla`

Lemma 1 If we have $AggQC_1$ and $AggQC_2$ with $F<\frac{N}{3}$, then both $AggQCs$ are not on the same round r.

$Proof.$ Suppose that blocks from B_j to B_{j+N-1} are committed in $AggQC_1$ at round r and blocks from B_{j+N} to B_{j+2N-1} are committed in $AggQC_2$ at round r'. The number of blocks are determined by the number of active leaders who propose blocks in parallel. It must be that at least $N-F$ leaders denoted as S_0, signed the block $B_{j\in(j,N-1)}$, and at least $N-F$ leaders denoted as S_1, signed the blocks $B_{j\in(j+N,j+2N-1)}$. Since there are only N leaders in total, $S_0 \cap S_1$ must intersect in at least $\frac{1}{3}$, and thus at least one honest leader is in $S_0 \cap S_1$. According to our protocol, every honest leader votes for at most one time for each height in the blockchain. Therefore, it must be that $r \neq r'$ and $AggQC_1$ is a prefix of $AggQC_2$.

Lemma 2 If at least one correct leader has received $N-F$ votes for block B_j in round r, then if some leaders have increased their round numbers due to network partitions or node failures, the round r is the last round that have the latest valid votes before proposing the next round.

$Proof.$ The leaders receive client's request in round r. The leader also have $N-F$ votes from each request in previous round $r-1$. When each leader proposing both the new requests and the $AggQC$ for $r-1$ in round r, the leaders receive proofs for the last blocks to be committed. If there is a correct replica that received $N-F$ votes for r, then it means that there are $N-F$ replicas sent their votes in the same round. Then, after network partitions or node failures, some replicas increase their rounds number $r+k$. However, the round r is still the last round that have the parent blocks for the next round. For simplicity, we assume the leader l_{i+1} is in r and has proposed B_{j+2}. A leader, say l_{i+2}, received a $N-F$ votes for B_{j+1}. The latest valid votes in leader l_{i+1} should be B_{j+1} in r. This is because the leaders cannot accept any proposal without the proof of the previous block votes.

4.3.2 Liveness

BigBFT guarantees its liveness under the partial synchronous model. After GST, when network conditions are good, the network is stable and delay time is known. If there are less than $F<\frac{N}{3}$ Byzantine leaders, the honest leaders can reach a decision in a δ time.

Lemma 3 To ensure liveness, each round has a coordinator, a set of leaders, and the round number is incremented. The protocol after GST, with a correct coordinators and leaders, eventually become synchronized and the block will be added to the blockchain.

$Proof.$ Let us assume that we have a correct coordinator called C, and leaders L_s for round r at time t. This means that L_s have received the coordinator message $RoundQC$ that at least $N-F$ leaders signed in r where $F<\frac{N}{3}$. Thus, at least $F+1$ correct leaders assign C as a coordinator at time t. By time $t+\delta$, all correct leaders complete the protocol phases.

Lemma 4 A new leaders set is chosen based on the their stakes. If a leader does not make progress, the leader L_i will be discarded from the leader set L_s.

$Proof.$ At the beginning of the round, the leaders is chosen by coordination phase. As per assumption all correct leaders are in the same round, therefore the correct leader will propose a prepare message with proof containing the latest votes $AggQC$. Since all leaders are in the same round, therefore leaders who are not successfully complete prepare and vote will eliminate from the L_s.

4.4 Performance Evaluation

4.4.1 Implementation and setup

We implemented BigBFT in Go using PaxiBFT framework [5], which uses core network files from Paxi [4]. PaxiBFT is an open source for prototyping, evaluating, and benchmarking BFT consensus and replication protocols. As shown in Figure 3.6, PaxiBFT readily provides most functionality that any coordination protocol needs for replication protocols. The entire protocol is available as open source at `https://github.com/salemmohammed/BigBFT`.

We evaluate the performance of our BigBFT prototype protocol and compare it with PBFT [23], Tendermint [16], Streamlet [25], and HotStuff [66], using the Amazon EC2 instances. We chose both Wide Area Network(WAN) across 5 AWS regions(Ohio, N.California, Oregon, N.Virginia, and Canada) and Local Area Network(LAN).

We deployed BigBFT on up to 20 m5a.large virtual machines, each of which has 2 vCPU, 8GiB RAM, and 10Gbps network throughput. To ensure that the client performance does not impact the results, we used the larger m5a.xlarge instances with 4 vCPUs for the clients. Based on our experiments results, the network size is appropriate to state and conclude our findings. To push system throughput, we varied the number of clients up to 90 clients and used a small message sizes. In our experiments, message size did not dominate consensus protocols performance, but the complexity of consensus protocols dominates the performance.

We compare the performance of protocols when $F = 0$ and N ranges from 4 to 20 full nodes. The results are shown in Figures 4.2 and 4.4. In each graph, the Y-axis shows the throughput in tx/sec, and X-axis the number of nodes(N). We define the throughput as the number of transactions per second (tx/s for short) that leaders commit. As we can see in both Figures 4.2 and 4.4, BigBFT achieves a better performance than PBFT, Tendermint, and Streamlet in LAN deployment. In a comparison to HotStuff, BigBFT is close to Hotstuff's throughput but has better latency. This is because, in both wide and local environment, the network is the bottleneck and the message patterns of BFT protocols, namely PBFT, Tendermint, and Streamlet, tend to be expensive. BigBFT on the other hand maintains a low number of message latency for each request to be committed and simple message patterns.

The latency messages are 4 messages. However, Hotstuff has 10 messages regardless of chain protocol as we summarized in Table 4.1. These latency messages are not important in LAN due to the short distance between nodes. As a result, BigBFT performs well in LAN deployment as you can see in Figure 4.2. In contrast to LAN, there is a long delay in WAN between nodes that helps BigBFT due to its low number of message latency to commit the requests faster. Compared with PBFT, Tendermint, Streamlet, and Hotstuff, BigBFT puts less stress on the leaders. Moreover, the performance of BigBFT is very close to that of HotStuff and Paxos [6].

Hotstuff has point to point communication while BigBFT is still using broadcasting topology between nodes even though we amortized the messages over the number of requests. The combination of messages in the vote phase causes some delay, but that delay was unnoticed in WAN due to the distance/latency gain from sending messages in parallel.

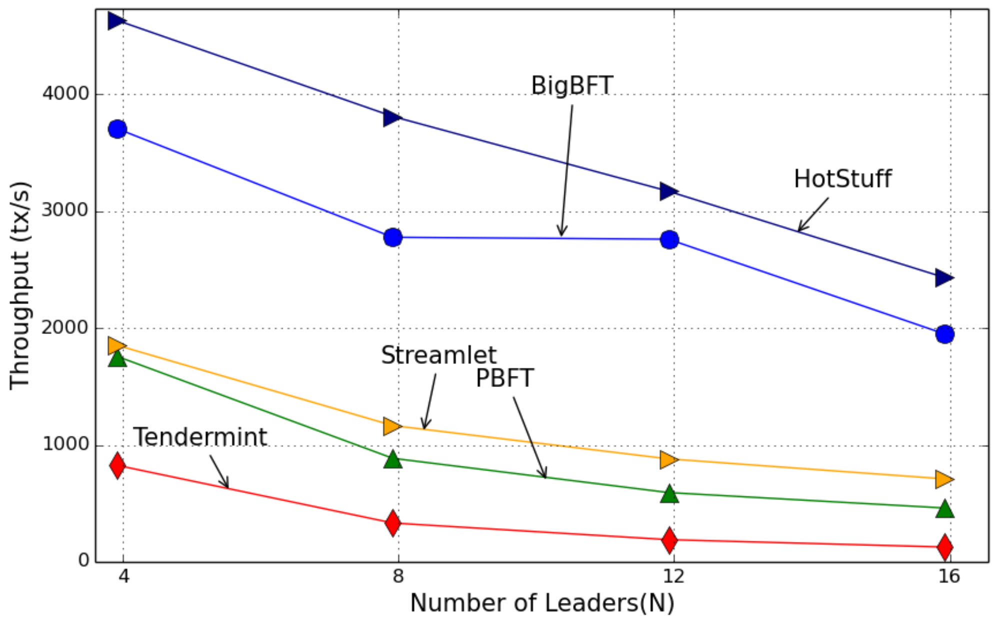

Figure 4.2: Throughput comparison in LAN

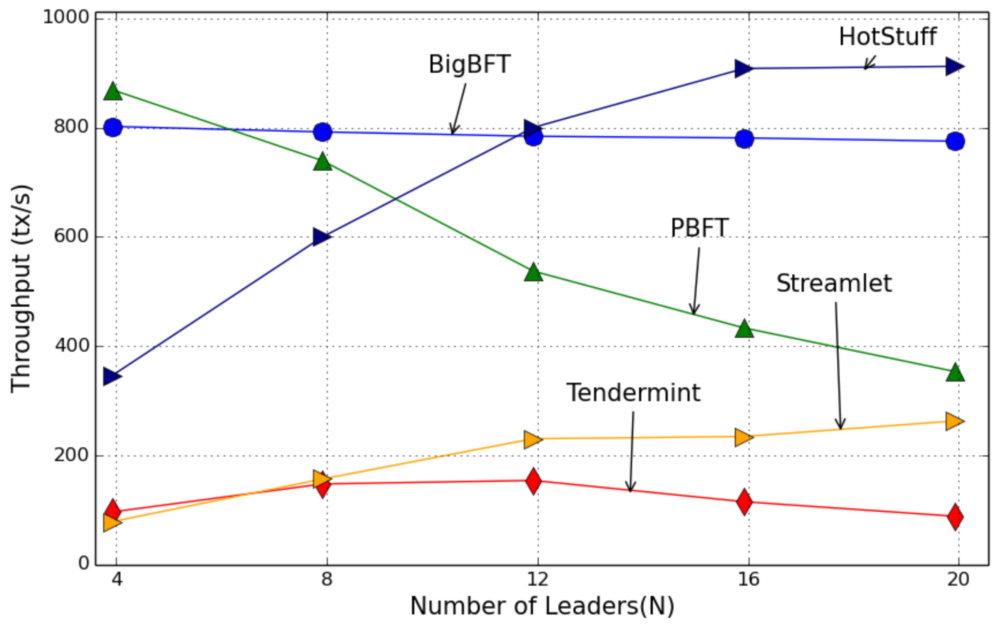

Figure 4.3: WAN's throughput comparison in Virginia, California, Oregon, Ohio, and Canada

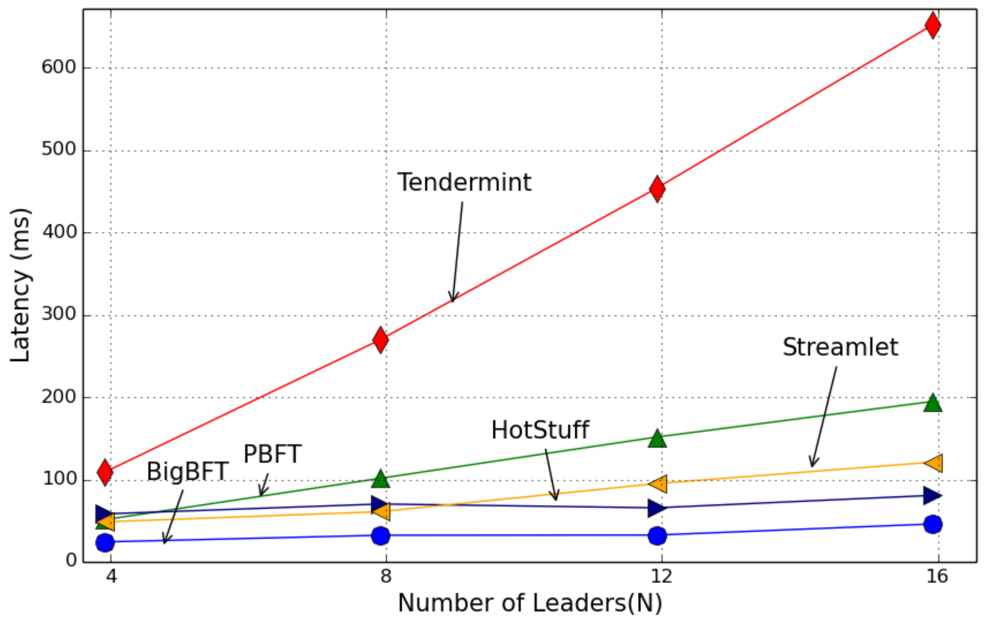

Figure 4.4: Latency comparison in LAN

4.4.2 Performance comparison between single and multi-leaders protocols

To study the performance of BigBFT, we compare BigBFT latency and throughput with PBFT [23], Tendermint [16], Hotstuff [66], and Streamlet [25]. We run all of these protocols with no faults to make sure we capture their absolute best performances. We chose these protocols to compare with BigBFT because we had studied and analyzed them in our previous work [6]. As a result of our previous work, we introduced BigBFT to alleviate the bottlenecks. BigBFT relies on a two phase common case commit protocol with $3F + 1$ replicas illustrated in Figure 4.1. To provide a fair comparison, all protocols implemented on the same framework PaxiBFT 3.6.

Figures 4.5 and 4.6 illustrate the latency versus throughput performance of these five protocols in a 20-node cluster. In each graph, the X-axis shows the throughput in tx/sec, and Y-axis the latency in ms. BigBFT and HotStuff did not saturated very quickly in Figure 4.6. Pushing the system throughput to its limit is difficult in WAN. In LAN, pushing the system throughput to its limit to get the system bottlenecks is easy due to the short network pipe between instances.

At this cluster size, BigBFT shows better latency than other protocols because it allows parallel executions. This removes the single leader bottleneck and allows BigBFT to have more throughput than many protocols. On the other hand, PBFT, Tendermint, and Streamlet are significantly limited by their increased communication costs.

PBFT is limited by a single leader and quadratic exchanging messages for every request. PBFT gets saturated quickly and reaches its limit of around 480 requests per second in LAN and 360 requests per second in WAN. While BigBFT have higher latency in LAN, but we see that it scales to higher throughput than other protocols except Hotstuff with better latency than all protocols. BigBFT provides great throughput improvements over traditional PBFT in wide area deployments.

BigBFT maintains low latency for much higher levels of throughput and shows higher throughput than Tendermint and Streamlet in WAN. Tendermint gets saturated quicker by a more complicated messaging and processing despite our workload having no conflicts.

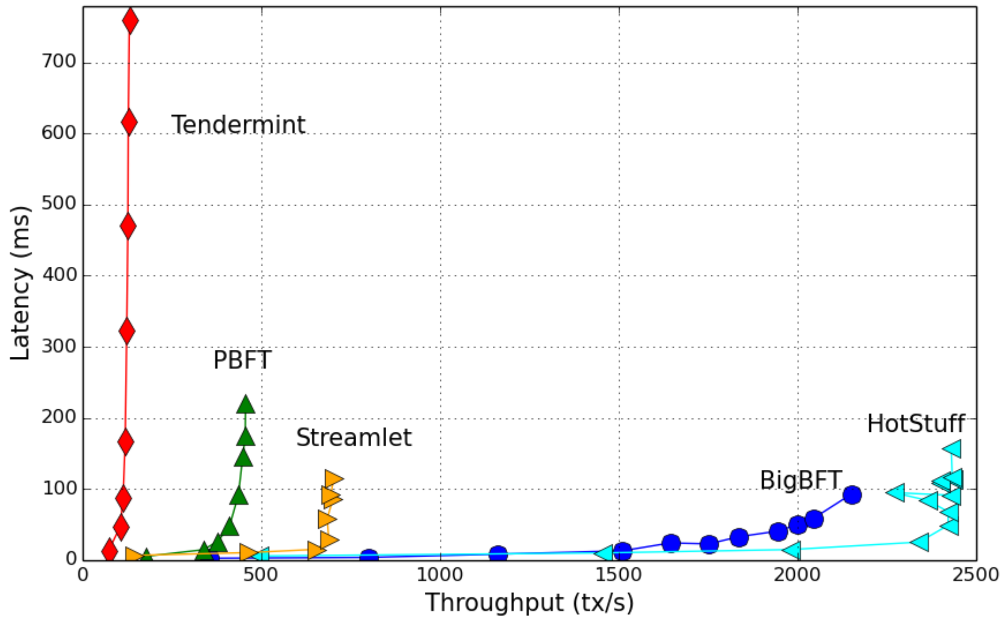

Figure 4.5: System throughput and the latency on 20-node LAN cluster

4.4.3 System Payload Size

Payload size have an impact on the communication performance of the system. With the large messages, system requires more resources for serialization and more network capacity for transmission. To study how different payload size impacts the performance of our studied protocols and BigBFT, we experiment with 16 node clusters. We measured the maximum throughput on each system under a write-only workload by 90 clients. Figure 4.7 shows the maximum throughput of BigBFT and Hotstuff at payload sizes varying from 128 to 2048 bytes. While BigBFT show less throughput than HotStuff at the beginning of payload sizes, both protocols exhibit a similar relative level of degradation as the payload size increases.

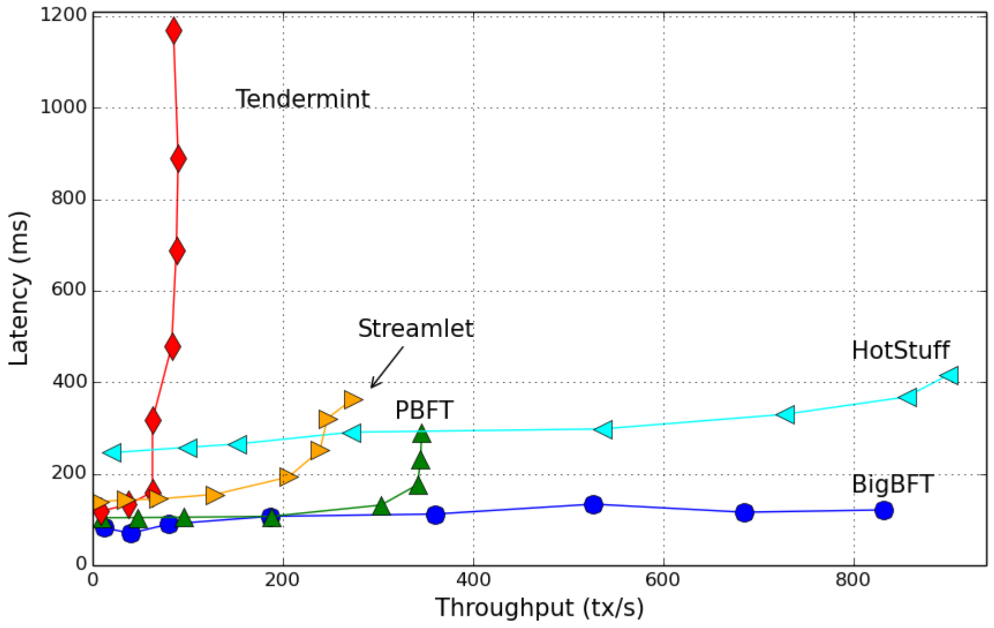

Figure 4.6: System throughput and the latency on 20-node WAN cluster in Virginia, California, Oregon, Ohio, and Canada

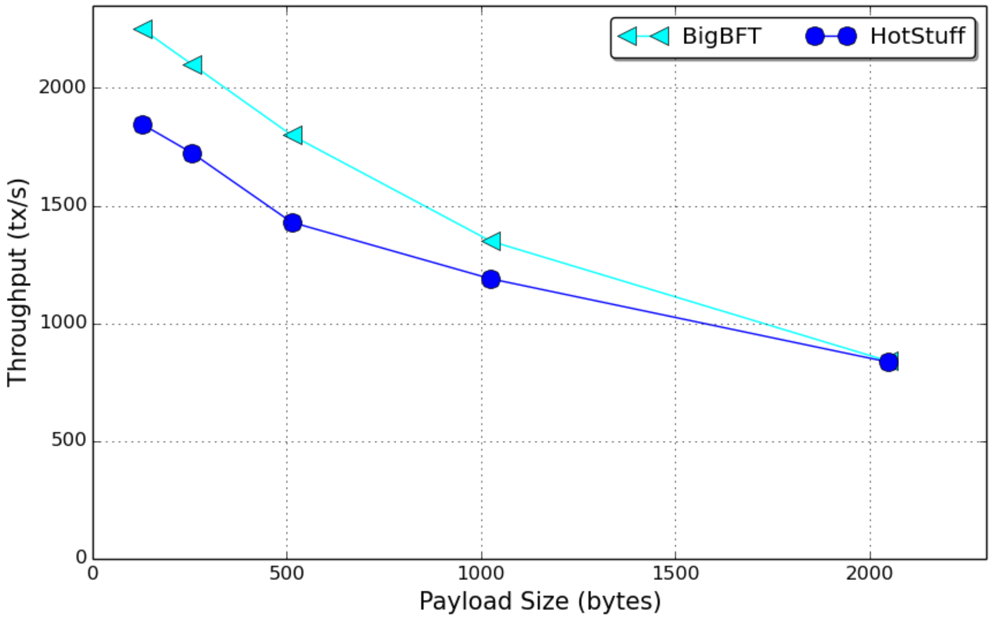

Figure 4.7: Maximum throughput at various payload sizes

4.5 Analysis and Discussion

In this section, we compare the strengths and weaknesses of the BigBFT and provide back-of-the-envelope calculations for estimating the latency and throughput performance. Table 4.1 provides a synopsis of the blockchain protocols characteristics we compared with BigBFT. We elaborate on these next.

Time Complexity. In Mir-BFT, and PBFT, the normal executions have 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. Leaders 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.

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 leaders 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 leaders, the message complexity is $O(N^3)$ [16].

Streamlet has communication message complexity $O(N^3)$. 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 $O(N^4)$.

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

	PBFT [23]	HotStuff [66]	Mir-BFT [61]	Streamlet [25]	Tendermint [16]	BigBFT
Critical path	5	10	5	4	5	4
Normal Message Complexity	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N^3)$	$O(N^2)$	$O(N)$
Multiple View Change	$O(N^4)$	$O(N^2)$	$O(N^4)$	$O(N^4)$	$O(N^3)$	$O(N)$
Responsive	Yes	Yes	Yes	No	No	Yes

Table 4.1: Characteristics of existing BFT consensus protocols and BigBFT

BigBFT has communication message complexity $O(N)$. BigBFT has a linear communication complexity in vote phase in the best case scenarios. In the worst case scenario when there is a leader cascading failure, the BigBFT message complexity does not change because overlapping between propose-vote phases and coordination phase.

4.5.1 Load and Capacity

BigBFT protocol reaches consensus once a quorum of participants agrees on the same decision. A quorum can be defined as sets containing $N - F$ majority leaders 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 leader. The capacity $Cap(S)$ is the highest number of quorum accesses that the system can possibly handle $Cap(S) = \frac{1}{\ell(S)}$.

In single leader protocols, the busiest node is the leader. In BigBFT, all nodes are busy all the time.

$$\ell(S) = \frac{1}{L}(Q-1)NumQ + (1 - \frac{1}{L})(Q-1) \tag{4.1}$$

where Q is the quorum size chosen in both leader and followers, NumQ is quorums number handled by leader/follower for every block request, and L is the number of operation leaders. There is a $\frac{1}{L}$ chance the node is the leader of a request. Leader communicates with $N - 1 = Q$ nodes. 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 BigBFT and PBFT protocols, and calculate the result for $N = 9$ leaders. The protocols perform better as the load decreases.

$$\ell(BigBFT) = \frac{50}{9} = 5\frac{5}{9} \tag{4.2}$$

In BigBFT protocol, equation 4.2 with N leaders, and L = N, quorum size Q = $\lfloor \frac{2N}{3} \rfloor$, and number of quorums $NumQ = 2$.

PBFT is a single leader protocol with $Q = \lfloor \frac{2N}{3} \rfloor$ and $NumQ = 2$. The load on PBFT is $\ell(PBFT) = 10$

4.5.2 Latency

The formula 4.3 calculates the latency of BigBFT.

$$Latency(S) = Critical\ Path + D_L + \delta \tag{4.3}$$

The critical path denotes the number of one-way message delays. BigBFT's critical path has a 4-message delay as illustrated in Table 4.1 for multiple consensus. D_L is the round trip message between a client and designated leader. In Table 4.1, PBFT has a 5-message delay for single consensus.

4.6 Future Work

4.6.1 Improving Scalability by Enabling Local Communication

One direction for future work is to enable hierarchical communications in order to scale the protocol to more nodes. At each round, a global coordinator selects a local coordinator in each region to maintain the configuration blockchain and to restrict the communication within each region. The global coordinator will reach consensus with local coordinators on sharding the space between regions. Local coordinator will further shard the space among all local leaders. Then, each region will run BigBFT locally among nodes to order blocks.

4.6.2 Client as a Coordinator

We are planning to enable client node to act as a system coordinator to further improve BFT scalability by reducing the interactions between replicas. The client can assign the request's sequence number, choose a set of trusted replicas to form a quorum, and learn the status of the request from the chain if committed. For instance, the client may choose a quorum of nodes including well-known validators, as shown in Stellar protocol [47], with the high stake values to commit its blocks. Client may use blockchain to check recent honest validators that are able to commit and add to the chain. For more flexibility, the client might develop some set of policies to choose its quorum such as choosing common nodes between recent last two committed blocks. However, the system should have a security mechanism to prevent a malicious client from violating the safety of the system.

4.7 Concluding Remark

We have presented BigBFT, a BFT-based consensus protocol for blockchains. BigBFT alleviates the communication bottlenecks in single leader BFT protocols by enabling multi-leader executions, and reduces the number of communication phases to be only two for reaching consensus on proposed blocks. This is achieved by pipelining blocks both within a round and across rounds. BigBFT also decouples the coordination phase from the main protocol and pipelines coordination round with consensus rounds to improve the system performance. We analyzed BigBFT performance by using a load formula and compared it with PBFT, Streamlet, Tendermint, and Hotstuff protocols. Our experimental evaluations show BigBFT's advantages in latency or throughput over other protocols.

Chapter 5

Performance Analysis and Comparison of Distributed Machine Learning Systems

Deep learning has permeated through many aspects of computing/processing systems in recent years. While distributed training architectures/frameworks are adopted for training large deep learning models quickly, there has not been a systematic study of the communication bottlenecks of these architectures and their effects on the computation cycle time and scalability. In order to analyze this problem for synchronous Stochastic Gradient Descent (SGD) training of deep learning models, we developed a performance model of computation time and communication latency under three different system architectures: Peer-to-Peer (P2P) and Ring allreduce (RA). To complement and corroborate our analytical models with quantitative results, we evaluated the computation and communication performance of these system architectures of the systems via experiments performed with Tensorflow and Horovod frameworks.

We found that the system architecture has a very significant effect on the performance of training. RA-based systems achieve scalable performance as they successfully decouple network usage from the number of workers in the system. In contrast, P2P systems suffer from significant network bottleneck. Finally, RA systems also excel by virtue of overlapping computation time and communication time, which P2P architecture fails to achieve.

5.1 Similarities of machine learning and blockchain systems

We investigate the architectural design of distributed machine learning systems as the design decisions inevitably affect the scalability of the systems. We study Ring-Allreduce as a representative machine learning system that is based on ring topology. We show that ring topology has improved system scalability comparing to P2P. Other work like RingBFT 5.4.1 investigates design decisions and checks the feasibility of ring topology in Blockchain. In both domains, ring communication topology improves both systems scalability comparing to P2P systems. We believe that some solutions such as communication topology will contribute for both domains.

5.2 Performance Modeling

To model the behavior of the system, and to estimate the system performance, we present a performance model that captures the computation time and communication latency based on varying system configurations. We deal with the systems at a high abstraction layer due to the complexity of the systems. Large-scale platforms have different underlining designs [67] such as TensorFlow, and Spark, and for that, it becomes difficult to design a performance model at a low level. Our model approximates both computation and communication runtime for a single epoch (a single pass through the full training set) of training DNN with mini-batch SGD. In this work, we present a performance model that is simple and accurate enough to calculate computation time and communication latency without intensive log data collection.

Our results have two network indicators, latency (the time it takes to send a message from point A to point B) and throughput (the processed amount of data every time unit). These indicators differ from one system design to another. Modeling network latency has two factors. Download time for workers receive data from the server while upload time for workers send gradients to servers. For further details, the parameters of the performance model P2P and RA in Table 5.1.

Notation	Meaning
Machine Learning Notation	
W^L	weight variables at layer L
b_i^L	bias variables i at layer L
$a^{(L)}$	forward Pass
$\sigma(.)$	activation function
Y	ground truth
C	loss function
n	Number of training examples
α	learning rate
b	batch size
t_i	iteration number i
m	mini-batch size
W	model size
Distributed Systems Notation	
w	number of worker
B	total bandwidth

Table 5.1: Performance mode notation table

5.2.1 Distributed Training with P2P System

In this system design, as shown in Figure 5.1, every node joins the system is a peer. Peers connect to one another and provide the functionality of saving model parameters and training the neural networks. The first P2P data-parallel system library to solve large-scale ML problems was introduced in [46]. At the high level, both client and server reside on the same machine, which allows replicas to send model updates to one-another, as shown in Figure 5.1. Initially, all nodes obtain the same model and subset of the dataset. Each client node calculates feedforward and back propagation passes over mini-batch SGD. At the end of each iteration, the workers push a subset of model parameter updates ΔW to parallel model replicas to ensure that each model receives the most recent updates from other nodes. The total epoch time that consists of communication delay of sending gradients for all other peers, receiving model from same machine, and computation time is shown in figure 5.3. In figure 5.4, we show the total system throughput that nodes have processed every time unit. In Figure 5.2, we compare the performance model with the actual running time. One advantage of this P2P model software simplicity because the developers write only one code and distributed on all active machines. However, this approach is limited by optimization algorithms and available hardware. Reading model parameters size and writing gradients size are different because workers read whole model size from the same machine while in writing the workers update subset of the model size through network. Recently, most DNN frameworks overlap computation time with gradients updates.

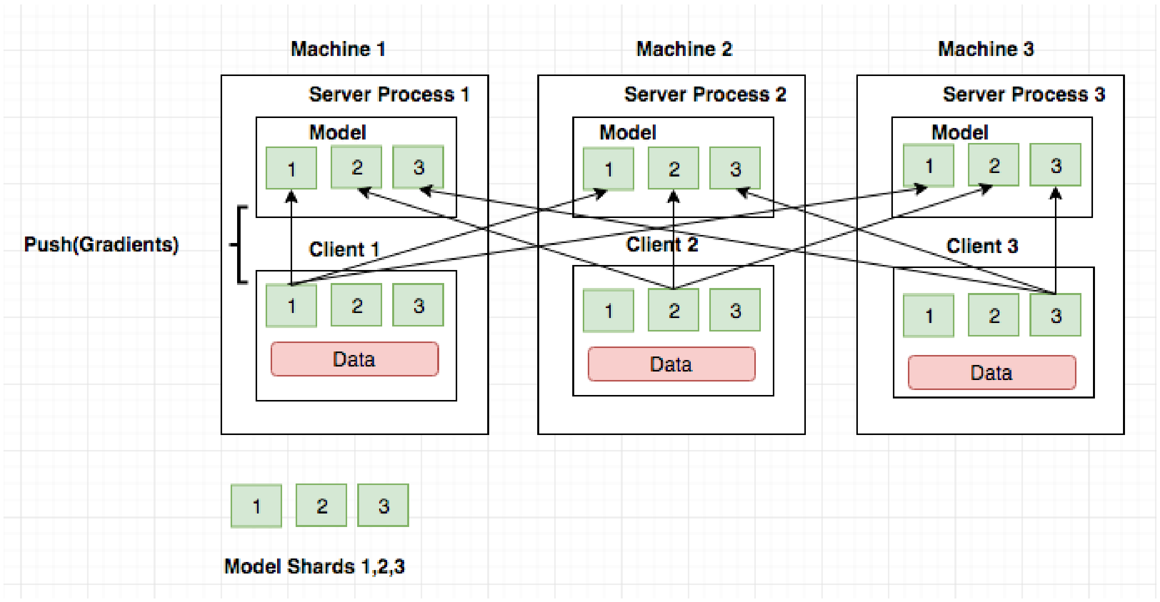

Figure 5.1: P2P based Architecture.

$$T_{(cpu-p2p)} = (epoch * (\frac{\frac{n}{b}}{w}) * (T_{processing} + update\_time)) \tag{5.1}$$

$$available_B = (\frac{TotalBandwidth}{2*(w-1)}) \tag{5.2}$$

$$T_{(tcp-p2p)} = (\frac{\frac{W}{w}}{available_B} * (\frac{(\frac{n}{b})}{w})) \tag{5.3}$$

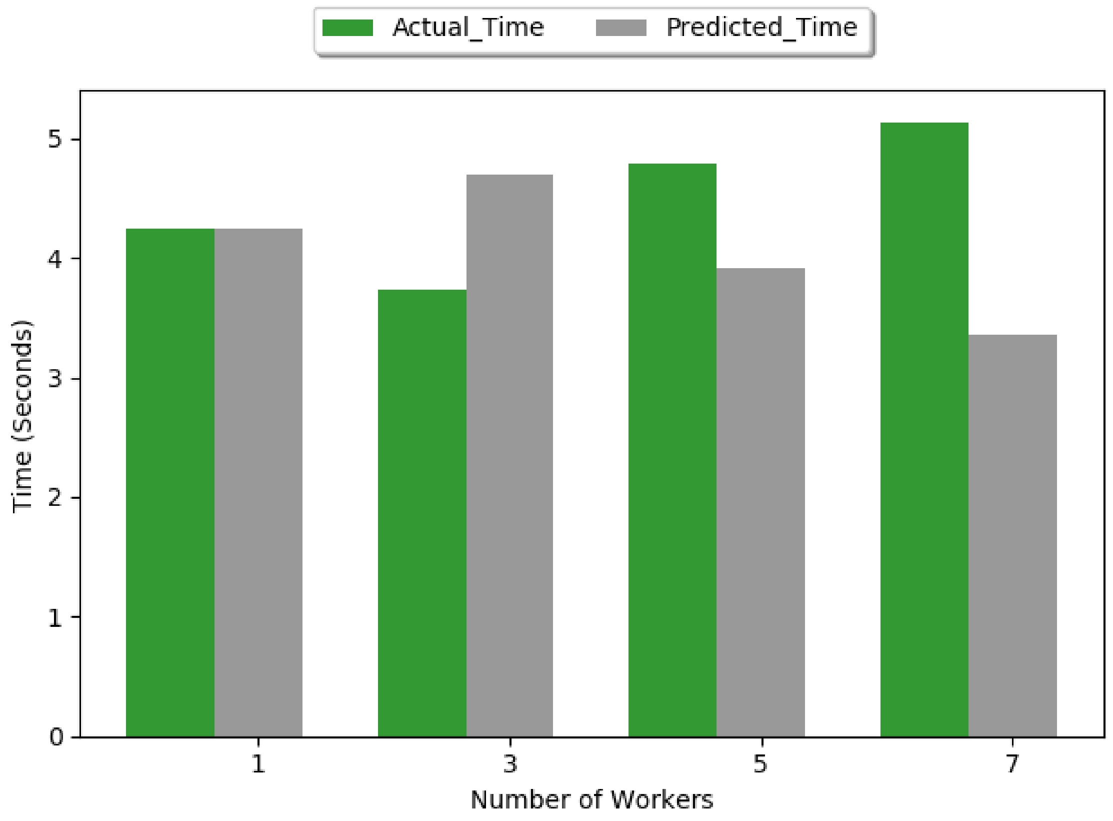

Figure 5.2: Estimated epoch time for P2P system.

$$T_{(total)} = T_{(cpu-p2p)} + T_{(tcp-p2p)} \tag{5.4}$$

Here, we are not interested in one iteration time, but we are interested in an epoch time. $available_B$ is the bandwidth between the peer and other peers. In this model, we noticed that gradients have not perfectly overlapped with computation time and have high overhead. In every iteration, server will send and receive $2 * (w - 1)$ messages. The message size will be equals to $(\frac{W}{w})$.

5.2.2 Distributed Training with Ring-allreduce

In this system architecture, as shown in Figure 5.5, the first paper was introduced in [53]. Uber Inc adapted the baidu RA algorithm [9] and $MPI - Allreduce()$ in its Horovod [59]

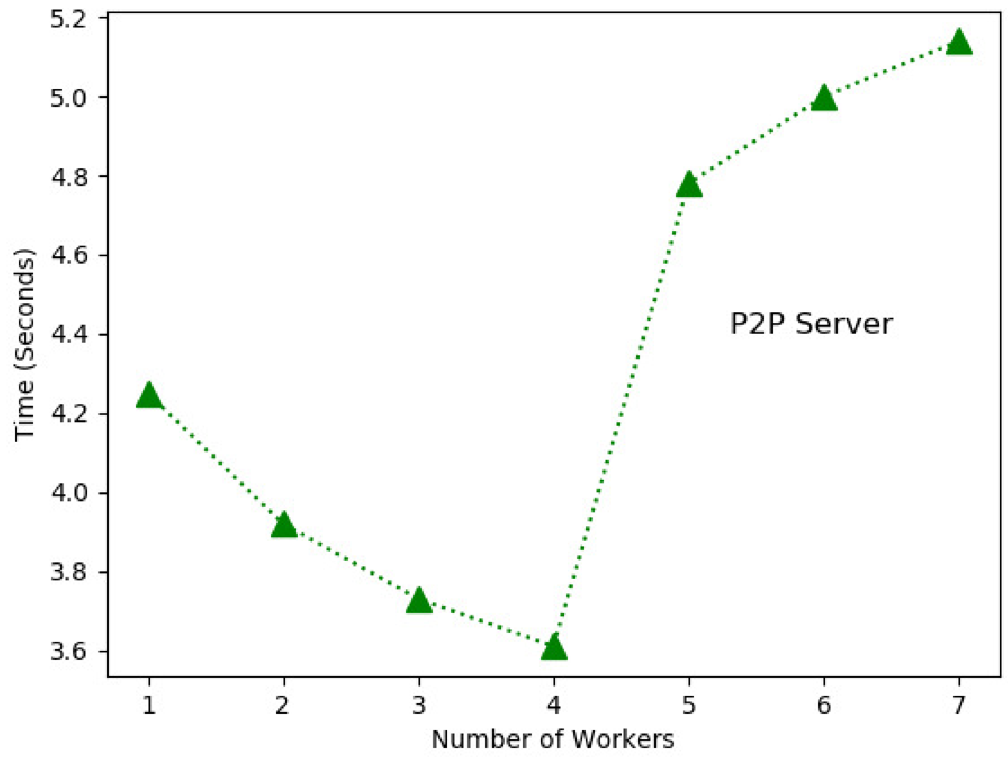

Figure 5.3: Epoch time of P2P system.

which is a distributed training framework for TensorFlow. In this model architecture, there is no central server that holds the model and aggregates gradients from workers as PS architecture. Instead, in distributed training, each worker reads its own subset data, calculates its gradients, sends its gradients to its successor node on the ring topology, and receives gradients from its node on the ring topology until all workers have the same values. Based on collected log information, there are many types of communications: negotiate broadcast, broadcast, MPI broadcast, allreduce, MPI allreduce, and negotiate allreduce. Also, MEMCPY IN FUSION BUFFER and MEMCPY OUT FUSION BUFFER are to copy data into and out of the fusion buffer. Each tensor broadcast/reduction in the Horovod involves two major phases. First is the negotiation phase where all workers send a signal to rank 0 that they are ready to broadcast/reduce the given tensor. Then, rank 0 sends a signal to the other workers to start broadcasting/reducing the tensor. Second is the processing phase where

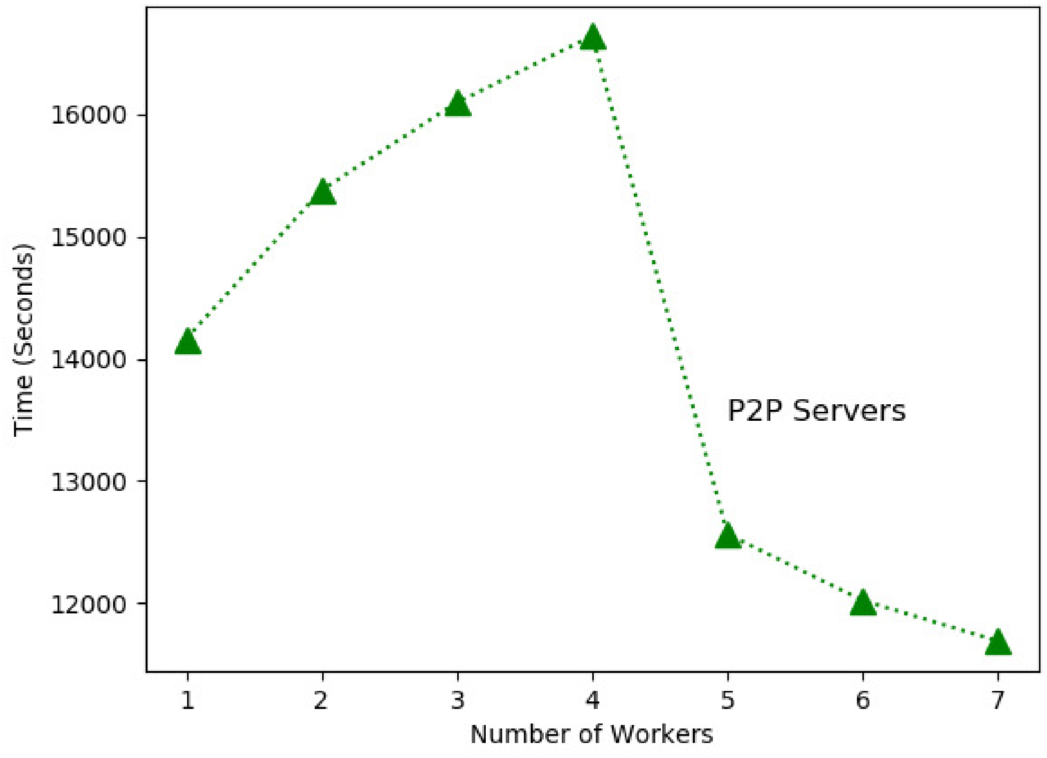

Figure 5.4: Measured training throughput of P2P system.

the gradients are computed after data loading and preprocessing. Both allreduce and MPI Allreduce are used to average the gradients to single value. The inter-GPU or inter-CPU communication and operation whether on a single network node or across network nodes in implementations of parallel and distributed deep learning training are built on top of MPI and can get benefits from all optimizations that related to MPI such as Open MPI [35]. The RA algorithm allows worker nodes to average gradients and send them to all nodes without the need for a PS. The aims of ring reduction operation are to reduce communication overhead that can cause by all-to-one or one-to-all collective communications [58]. Horovod also has less amount of code lines for distributed training and increased the scalability comparing to the well-known PS systems. The RA has used distributed data parallelism scheme. Every node in the system has a subset of the data $\frac{n}{w}$. The RA [53] is bandwidth-optimal. Technically, every node of w communicates with two of its peers $2 * (w - 1)$

times. During this communication, a node sends and receives chunks of the data buffer. Every node starts with a local value and ends up with an aggregate global result. In the first $(w - 1)$ iterations, each node in a ring topology sends gradients to its successor and receives gradients from its predecessor. Following by a reduction operation that adds up received values to the values in the node's buffer. In the second $(w - 1)$ iterations, every node has a sub-final block of data. Finally, all-gather operation transmits a final aggregated block to every other node. The bandwidth is optimal with enough buffer size for storing received messages [53]. RA scales independently of the number of nodes as we find in our experiments 5.7. However, RA is limited by the slowest directed communication link between neighbors and available network bandwidth. The latency in the experiment show near steady bandwidth usage that illustrated 5.6. RA overlaps the computation of gradients at lower layers in a deep neural network with the transmission of gradients at higher layers, which reducing training time.

$$T_{(cpu-ring)} = (epoch * (\frac{\frac{n}{b}}{w}) * (T_{processing} + update\_time)) \tag{5.5}$$

$$T_{(tcp-ring)} = \frac{(2 * (w-1) * \frac{W}{w})}{Bandwidth} \tag{5.6}$$

$$T_{(total)} = T_{(cpu-ring)} + T_{(tcp-ring)} \tag{5.7}$$

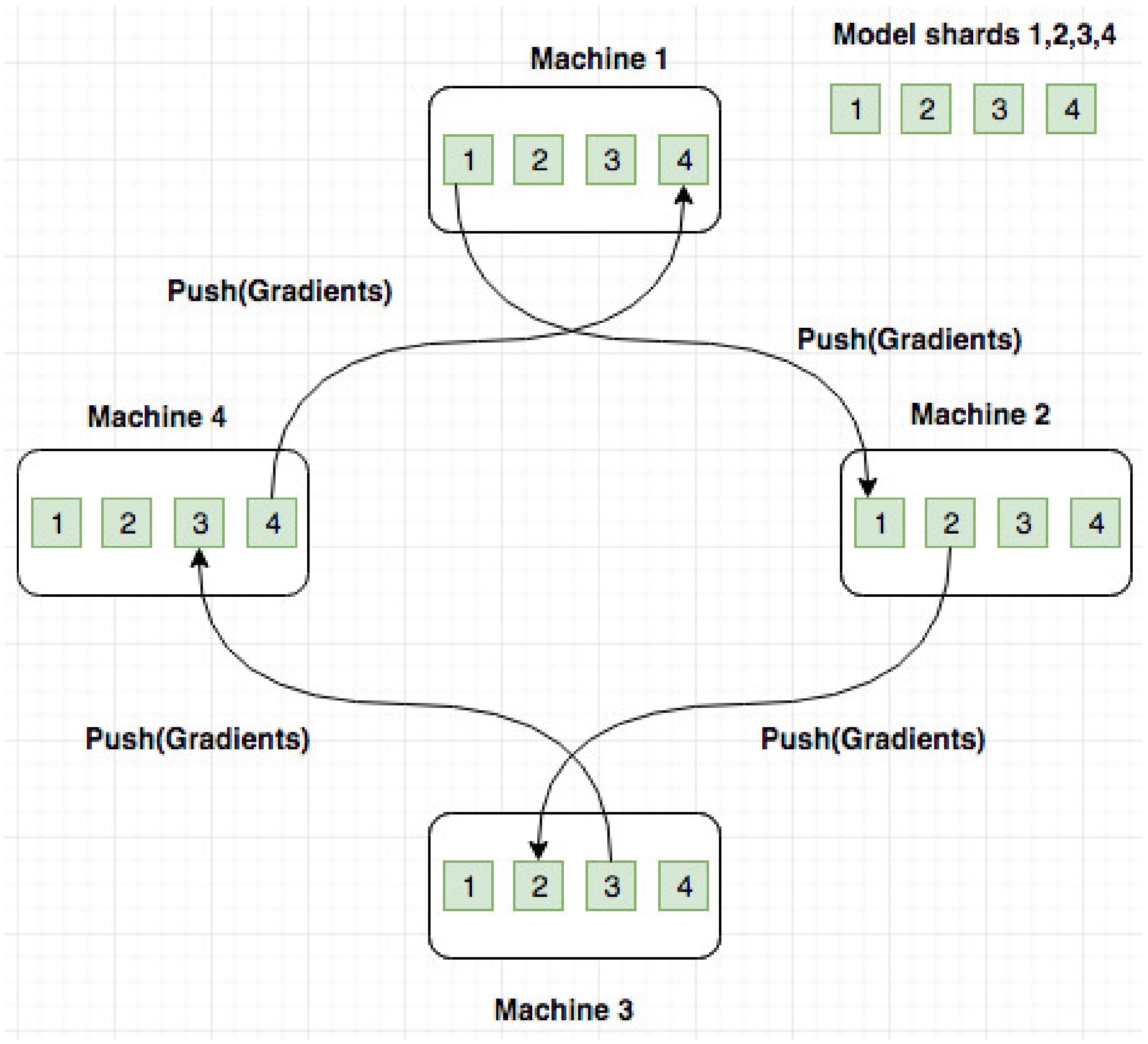

Figure 5.5: RA Architecture.

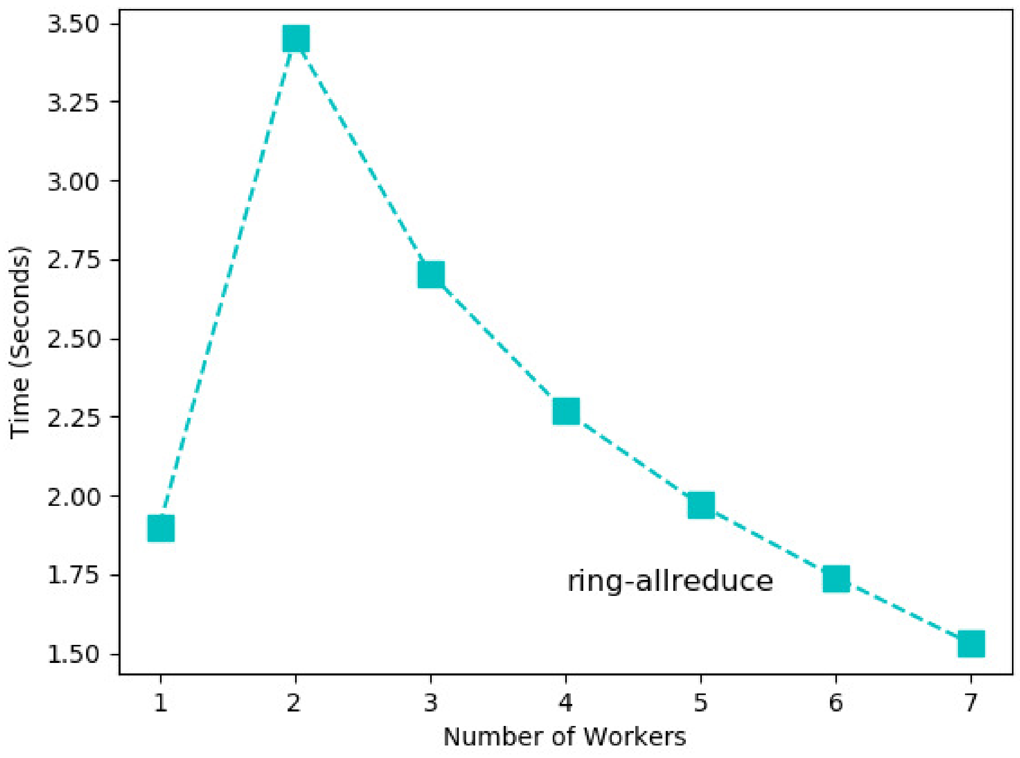

Figure 5.6: Epoch time of RA.

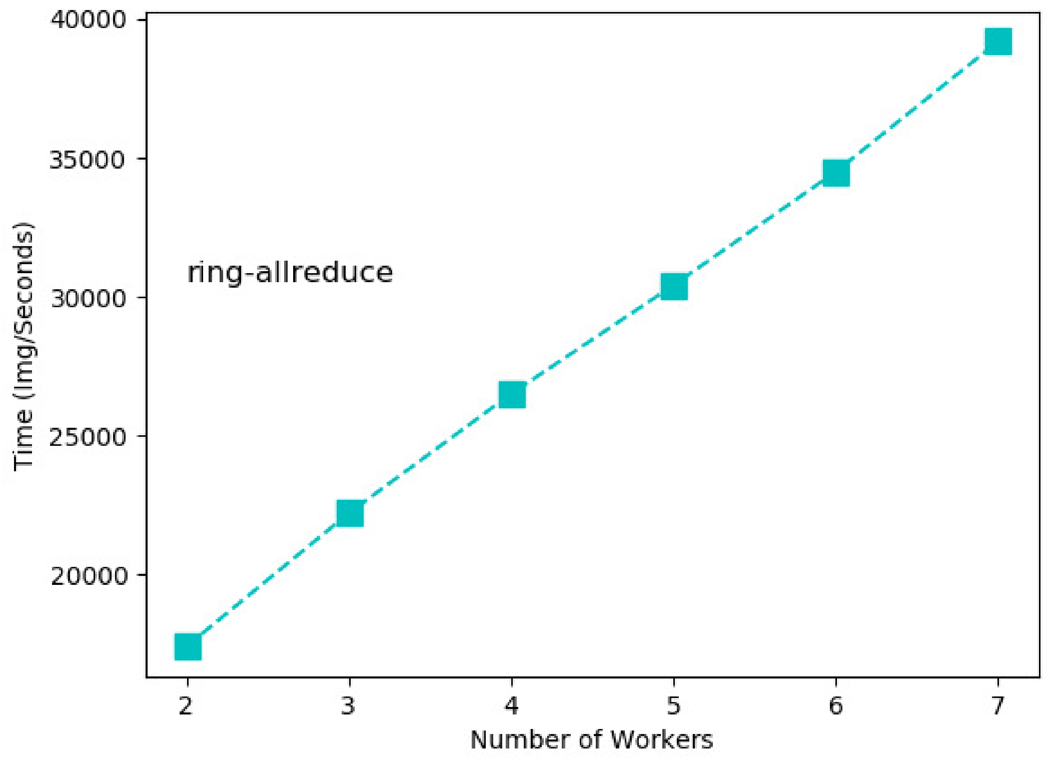

Figure 5.7: Measured training throughput of RA.

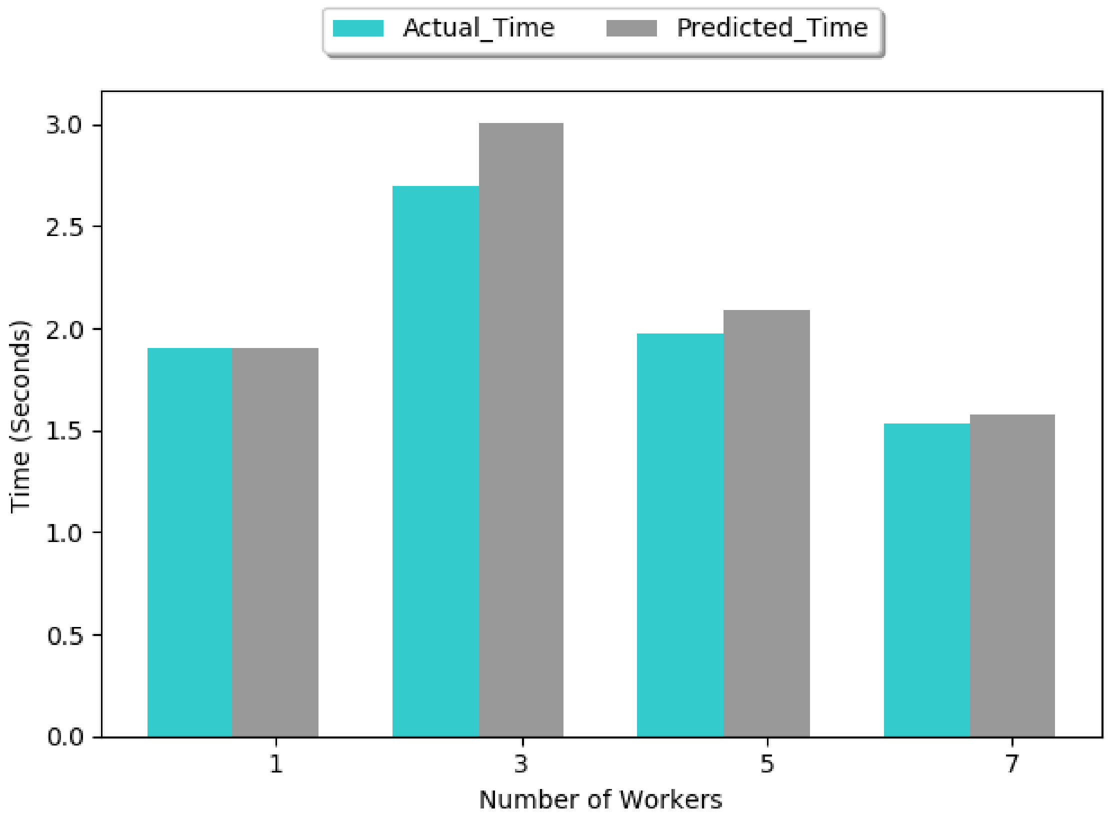

Figure 5.8: Estimated epoch time for RA.

5.3 Evaluation

5.3.1 Experimental Environment

Here, we run a set of experiments in distributed ML system introduced in Section 3.1. In order to provide a quantitative evaluation of P2P and RA (Horovod), we evaluated the performance of these system architectures with the same basic classification ML tasks. The system performance has two dimensions, latency metric and throughput metric. All of our experiments were conducted in an Amazon EC2 cloud computing platform using m4.xlarge instances. Each instance contains 4 vCPU powered by Intel Xeon E5-2676 v3 processor and 16GiB RAM. We use the MNIST database of handwritten digits [45] as our dataset. The MNIST dataset contains 60,000 training samples and 10,000 test samples of handwritten digits (0 to 9). Each digit is normalized and centered in a gray-scale (0 - 255) image with size 28 * 28. Each image consists of 784 pixels that represent the features of the digits. we deployed worker machines from one to seven machines to evaluate and quantify each system throughput and latency. All our ML classification task are written on top of TensorFlow version 1.11.0, an open-source dataflow software library originally release by Google.

5.3.2 Experimental Evaluation

We implemented multilayer neural networks with two hidden layers, and we chose the Softmax activation function as the output layer on three system architectures. We did not include in our study the neural network convergence because we believe that it depends on the neural network architecture and hyper-parameters and it has no dependence of distributed computing framework. For all experiments, we fixed the batch size (Batch size =100).

Peer-to-Peer. We have the number of servers equal to the number of clients as shown

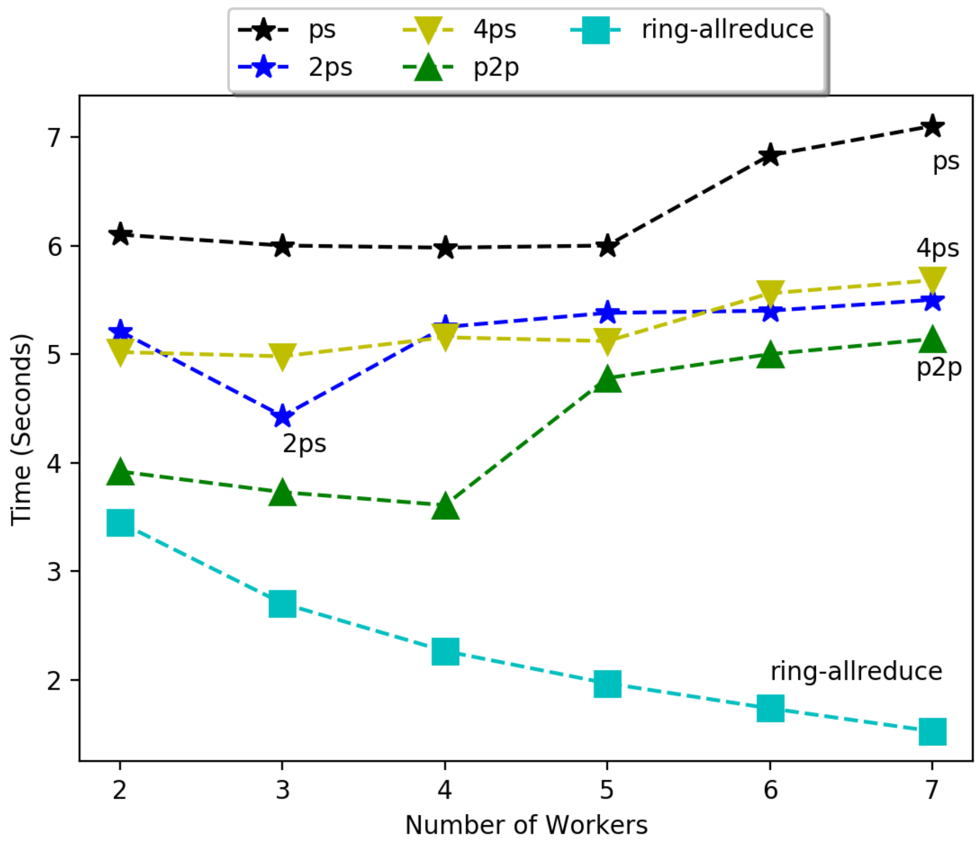

Figure 5.9: Latency Comparison.

in Figure 5.1. In our experiment, we have seven servers and seven clients co-allocated on seven machines. We have noticed some improvement in latency and throughput comparing to 1PS, 2PS, and 4PS as in Figures 5.9, and Figure 5.10. The reason is that the part of model was located on same machine where server located. Also, in this training, we do not need to pull the model from remote machines because it is already updated on the same machine.

Ring Allreduce. In Figures 5.6, we noticed that the epoch time reduced sub-linearly with number of machines due to many factors like bandwidth independence from number of nodes, computation and communication overlaps.

Ease of Development TensorFlow has low-level and high-level API in Python and C++. Back-end TensorFlow was written in C++ while front-end was written in wide language support such as Python and C++. In distributed training, developers have to write and de-

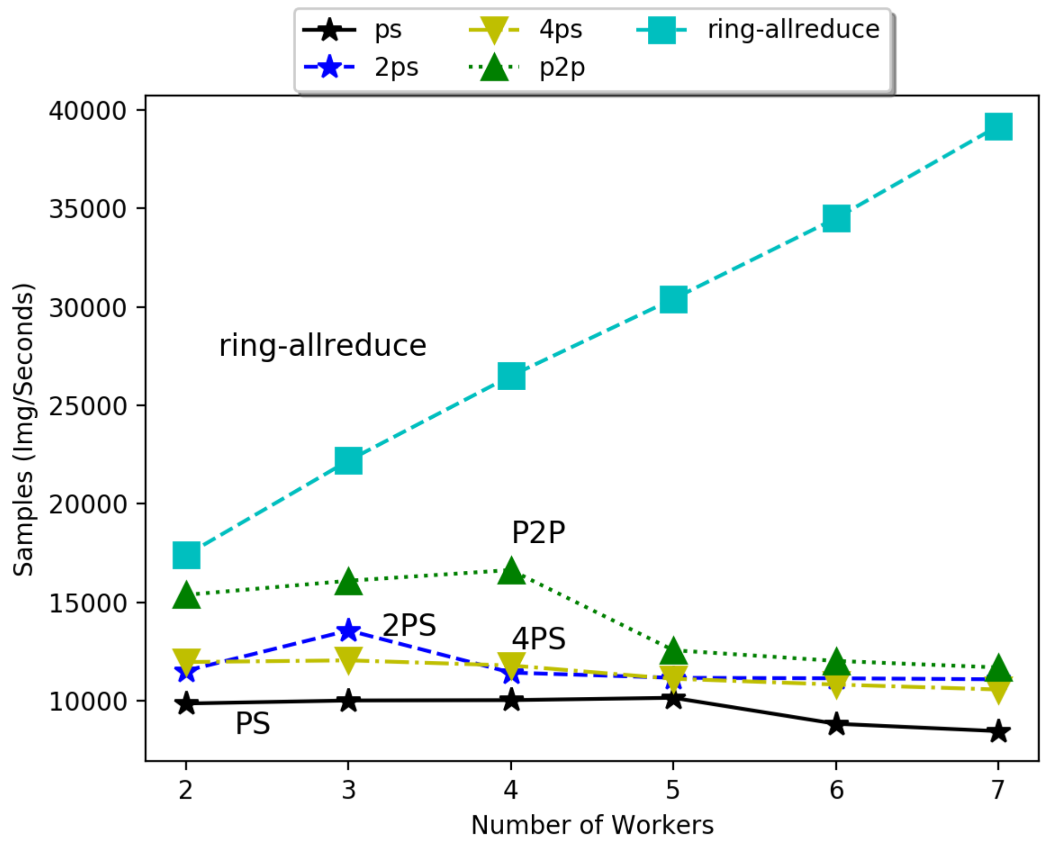

Figure 5.10: Throughput Comparison.

ploy the code on each machine or have to setup the cloud manager. Both ways need expert knowledge to setup and run the network training. In TensorFlow, there are many functions that are available on the framework but because of the updates and frequent new releases with new features deprecation make developers confused. The TensorFlow provides more APIs and primitives than any other ML framework. Debugging is hard but fortunately, we have computational graph visualizations (Tensorboard) that offer the visualization suite for tracking performance and network topology. The TensorFlow has large community support and widely used in many business and labs. Horovod API is different from TensorFlow in many ways such as simplicity of running distributed training. Developers write the code on one machine and that machine will communicate it to every other machine in the system. The Horovod APIs and primitives are not rich comparing to TensorFlow. Horovod has no clear debugging tool and it uses TensorFlow Tensorboard. Horovod has a lack of support and only few business and people are familiar with the Horovod library. In distributed training, I noticed that there is not one right answer for which architecture should be used. Using PS is good when developers have less powerful and not reliable machines such as cluster of CPU. In TensorFlow, PS architecture is well supported and developers will have a large community for help in debugging and suggestions. On the other hand, Horovod is preferable if developers' environments have fast devices such as GPU with strong communication link.

5.4 Related Work

5.4.1 RingBFT

RingBFT [55] proposed a solution for cross-shard transactions to improve system performance. RingBFT is built based on ring topology and assumes that the PBFT protocol is running in each shard. In the case of a single-shard, the client request(C) is submitted to shard A. The primary receives the request, and runs PBFT with all replicas in Shard A. In case of a cross-shard where the client request(C) is going to modify A,B, and D, the C submits to shard A that has the lowest id. The primary receives the request, and runs PBFT with all replicas in Shard A. Then, each replica in A forwards the result to the next shard(B). Notice, every replica in every shard has the same id in another shard. Upon receiving the forward messages in shard B, each replica of B will broadcast forward the messages to all replicas of B. After receiving the first prepare message, each replica starts a timeout for remote view-change. If every node receives f+1 prepare messages, then it will stop remote view-change timeout. Otherwise, each replica of B will send to each replica of A that has the same id. If replicas in A receives f+1 messages from replicas in B, the replicas in A will do local view change and forward the messages again. Finally, the last shard D, will forward the messages to A which is considered the primary shard for the client's transaction and A will reply to the client. If every shard execution depends on the previous shard, we need another rotation on the ring to carry dependencies and to execute locally.

5.5 Conclusion

In this work, we presented a comparative analysis of communication performance (latency and throughput) for two distributed system architectures. We found in P2P that the throughput fails to increase linearly due to network congestion. We also found that RA achieves better performance due to the efficient use of network bandwidth and overlapping computation and communication.

We hope our study would help the practitioners for selecting the system architecture and deployment parameters for their training systems. Our study can also pave the way for future work to estimate the scalability of distributed DNNs training. A promising direction for future work is to study the trade-offs between network congestion and extra computation. The research question here from the distributed system perspective is to identify which architectures and design elements can facilitate exploring and exploiting these trade-offs.

Chapter 6

Conclusion

This chapter summarizes our contributions and presents two potential future research topics that are related to the accomplished works.

6.1 Summary

We took a two-pronged approach for solving system bottlenecks. First, we studied prominent Byzantine fault tolerant(BFT) consensus protocols. We proposed an evaluation framework for BFT protocols and evaluated its performance. Then, we provided a theoretical analysis of complexity of these consensus protocols. Second, to address this scalability bottleneck, we propose using communication pipelining and communication aggregation techniques. We show that these techniques have significant effect on improving system performance. We also investigate suitable communication topologies; we propose a ring topology because it provides constant communication costs between replicas and allows the use of piggybacking technique.

6.2 Future Work

6.2.1 Improving Scalability by Enabling Local Communication

One direction for future work is to enable hierarchical communications in order to scale the protocol to more nodes. At each round, a global coordinator selects a local coordinator in each region to maintain the configuration blockchain and to restrict the communication within each region. The global coordinator will reach consensus with local coordinators on sharding the space between regions. Local coordinator will further shard the space among all local leaders. Then, each region will run BigBFT locally among nodes to order blocks.

6.2.2 Client as a Coordinator

We are planning to enable client node to act as a system coordinator to further improve BFT scalability by reducing the interactions between replicas. The client can assign the request's sequence number, choose a set of trusted replicas to form a quorum, and learn the status of the request from the chain if committed. For instance, the client may choose a quorum of nodes including well-known validators, as shown in Stellar protocol [47], with the high stake values to commit its blocks. Client may use blockchain to check recent honest validators that are able to commit and add to the chain. For more flexibility, the client might develop some set of policies to choose its quorum such as choosing common nodes between recent last two committed blocks. However, the system should have a security mechanism to prevent a malicious client from violating the safety of the system.

Chapter 7

List of Publications

- **Salem Alqahtani**, Murat Demirbas. BigBFT: A Multileader Byzantine Fault Tolerance Protocol for High Throughput. International Performance Computing and Communications Conference (IPCCC), 2021.

- **Salem Alqahtani**, Murat Demirbas. Bottlenecks in Blockchain Consensus Protocols. IEEE International Conference on Omni-Layer Intelligent Systems (COINS), 2021.

- **Salem Alqahtani**, Murat Demirbas. Performance Analysis and Comparison of Distributed Machine Learning Systems. The 28th International Conference on Computer Communication and Networks (ICCCN), 2019.

- Kuo Zhang, **Salem Alqahtani**, Murat Demirbas. ¡b¿A Comparison of Distributed Machine Learning Platforms. The 26th International Conference on Computer Communication and Networks (ICCCN), 2017.

Reference

[1] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. "Hot-stuff the linear, optimal-resilience, one-message BFT devil". In: *CoRR, abs/1803.05069* (2018).

[2] Ittai Abraham, Dahlia Malkhi, et al. "The blockchain consensus layer and BFT". In: *Bulletin of EATCS* 3.123 (2017).

[3] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. "Dissecting the performance of strongly-consistent replication protocols". In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1696–1710.

[4] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. *Paxi Framework*. `https://github.com/ailidani/paxi`. 2018.

[5] Salem Alqahtani. *PaxiBFT*. `https://github.com/salemmohammed/PaxiBFT`. 2019.

[6] Salem Alqahtani and Murat Demirbas. "Bottlenecks in Blockchain Consensus Protocols". In: *arXiv preprint arXiv:2103.04234* (2021).

[7] Yackolley Amoussou-Guenou et al. "Dissecting tendermint". In: *International Conference on Networked Systems*. Springer. 2019, pp. 166–182.

[8] Zeta Avarikioti et al. "FnF-BFT: Exploring Performance Limits of BFT Protocols". In: *arXiv preprint arXiv:2009.02235* (2020).

[9] baidu-research. *baidu-research/tensorflow-allreduce*. Aug. 2017. URL: `https://github.com/baidu-research/tensorflow-allreduce`.

[10] Shehar Bano et al. "SoK: Consensus in the age of blockchains". In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 2019, pp. 183–198.

[11] Catalonia-Spain Barcelona. “Mencius: building efficient replicated state machines for WANs”. In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. 2008.

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

[13] Christian Berger and Hans P Reiser. “Scaling Byzantine consensus: A broad analysis”. In: *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. 2018, pp. 13–18.

[14] Dan Boneh et al. “Aggregate and verifiably encrypted signatures from bilinear maps”. In: *International conference on the theory and applications of cryptographic techniques*. Springer. 2003, pp. 416–432.

[15] Ethan Buchman. “Tendermint: Byzantine fault tolerance in the age of blockchains”. PhD thesis. 2016.

[16] Ethan Buchman, Jae Kwon, and Zarko Milosevic. “The latest gossip on BFT consensus”. In: *arXiv preprint arXiv:1807.04938* (2018).

[17] Sean Busbey. *ycsb*. `https://github.com/brianfrankcooper/YCSB`.

[18] *Business Data Platform Statista*. `https://www.statista.com/statistics/863917/number-crypto-coins-tokens/`. Accessed: 2021-10-12.

[19] Vitalik Buterin and Virgil Griffith. “Casper the friendly finality gadget”. In: *arXiv preprint arXiv:1710.09437* (2017).

[20] Christian Cachin and Marko Vukolić. “Blockchain consensus protocols in the wild”. In: *arXiv preprint arXiv:1707.01873* (2017).

[21] Fran Casino, Thomas K Dasaklis, and Constantinos Patsakis. “A systematic literature review of blockchain-based applications: Current status, classification and open issues”. In: *Telematics and informatics* 36 (2019), pp. 55–81.

[22] Miguel Castro, Barbara Liskov, et al. “Practical Byzantine fault tolerance”. In: *OSDI*. Vol. 99. 1999. 1999, pp. 173–186.

[23] Miguel Castro, Barbara Liskov, et al. “Practical byzantine fault tolerance”. In: *OSDI*. Vol. 99. 1999. 1999, pp. 173–186.

[24] Benjamin Y Chan and Elaine Shi. *Streamlet: Textbook Streamlined Blockchains*. Cryptology ePrint Archive, Report 2020/088. `https://eprint.iacr.org/2020/088`. 2020.

[25] Benjamin Y Chan and Elaine Shi. “Streamlet: Textbook Streamlined Blockchains.” In: (2020).

[26] T-H Hubert Chan, Rafael Pass, and Elaine Shi. “PaLa: A Simple Partially Synchronous Blockchain.” In: *IACR Cryptol. ePrint Arch.* 2018 (2018), p. 981.

[27] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. “PigPaxos: Devouring the communication bottlenecks in distributed consensus”. In: *arXiv preprint arXiv:2003.07760* (2020).

[28] Konstantinos Christidis and Michael Devetsikiotis. “Blockchains and smart contracts for the internet of things”. In: *Ieee Access* 4 (2016), pp. 2292–2303.

[29] *Cosmos price Coinbase*. `https://www.coinbase.com/price/cosmos`. Accessed: 2021-10-12.

[30] *Diem Facebook Crypto*. `https://www.diem.com/en-us/`. Accessed: 2021-10-12.

[31] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. “Consensus in the presence of partial synchrony”. In: *Journal of the ACM (JACM)* 35.2 (1988), pp. 288–323.

[32] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. “Consensus in the presence of partial synchrony”. In: *Journal of the ACM (JACM)* 35.2 (1988), pp. 288–323.

[33] Facebook. *Libra Framework*. `https://github.com/libra/libra`. 2018.

[34] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. “Impossibility of distributed consensus with one faulty process”. In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[35] Edgar Gabriel et al. “Open MPI: Goals, concept, and design of a next generation MPI implementation”. In: *In Proceedings, 11th European PVM/MPI Users' Group Meeting*. 2004, pp. 97–104.

[36] Guy Golan Gueta et al. “SBFT: A scalable and decentralized trust infrastructure”. In: *arXiv e-prints* (2018), arXiv–1804.

[37] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. “Rcc: Resilient concurrent consensus for high-throughput secure transaction processing”. In: *Int. Conf. on Data Engineering (ICDE). IEEE*. 2021.

[38] Kexin Hu et al. “Don’t Count on One to Carry the Ball: Scaling BFT without Sacrificing Efficiency”. In: *arXiv preprint arXiv:2106.08114* (2021).

[39] Ethan Buchman Jae Kwon. *My Research Software*. Version 2.0.4. Dec. 2021. URL: `https://github.com/cosmos/cosmos/blob/master/WHITEPAPER.md`.

[40] Jae Kwon and Ethan Buchman. “Cosmos: a network of distributed ledgers (2016)”. In: *URL https://cosmos. network/whitepaper* (2016).

[41] Leslie Lamport. *Specifying systems*. Vol. 388. Addison-Wesley Boston, 2002.

[42] Leslie Lamport. “The part-time parliament”. In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 277–317.

[43] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine generals problem”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

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

[45] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: (2010). URL: `http://yann.lecun.com/exdb/mnist/`.

[46] Hao Li et al. “Malt: distributed data-parallelism for existing ml applications”. In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 3.

[47] David Mazieres. “The stellar consensus protocol: A federated model for internet-level consensus”. In: ().

[48] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. “Providing authentication and integrity in outsourced databases using Merkle hash trees”. In: ().

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

[50] Moni Naor and Avishai Wool. “The load, capacity, and availability of quorum systems”. In: *SIAM Journal on Computing* 27.2 (1998), pp. 423–447.

[51] Giang-Truong Nguyen and Kyungbaek Kim. “A Survey about Consensus Algorithms Used in Blockchain.” In: *Journal of Information processing systems* 14.1 (2018).

[52] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 305–319.

[53] Pitch Patarasuk and Xin Yuan. “Bandwidth optimal all-reduce algorithms for clusters of workstations”. In: *Journal of Parallel and Distributed Computing* 69.2 (2009), pp. 117–124.

[54] Marshall Pease, Robert Shostak, and Leslie Lamport. “Reaching agreement in the presence of faults”. In: *Journal of the ACM (JACM)* 27.2 (1980), pp. 228–234.

[55] Sajjad Rahnama et al. “RingBFT: Resilient Consensus over Sharded Ring Topology”. In: *arXiv preprint arXiv:2107.13047* (2021).

[56] Fred B Schneider. “Implementing fault-tolerant services using the state machine approach: A tutorial”. In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319.

[57] Fred B Schneider. “The state machine approach: A tutorial”. In: *Fault-tolerant distributed computing*. Springer, 1990, pp. 18–41.

[58] Alex Sergeev. *An Uber Journey in Distributed Deep Learning*. Youtube. 2015. URL: `https://www.youtube.com/watch?v=SphfeTl70MI`.

[59] Alexander Sergeev and Mike Del Balso. “Horovod: fast and easy distributed deep learning in TensorFlow”. In: *arXiv preprint arXiv:1802.05799* (2018).

[60] Elaine Shi. “Streamlined blockchains: A simple and elegant approach (a tutorial and survey)”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2019, pp. 3–17.

[61] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolić. “Mir-bft: High-throughput BFT for blockchains”. In: *arXiv preprint arXiv:1906.05552* (2019).

[62] Marko 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.

[63] Abdul Wahab and Waqas Mehmood. “Survey of Consensus Protocols”. In: *arXiv preprint arXiv:1810.03357* (2018).

[64] Yin Yang. “Linbft: Linear-communication byzantine fault tolerance for public blockchains”. In: *arXiv preprint arXiv:1807.01829* (2018).

[65] Maofan Yin et al. “HotStuff: BFT consensus with linearity and responsiveness”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. ACM. 2019, pp. 347–356.

[66] Maofan Yin et al. “HotStuff: BFT consensus with linearity and responsiveness”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. ACM. 2019, pp. 347–356.

[67] Kuo Zhang, Salem Alqahtani, and Murat Demirbas. “A Comparison of Distributed Machine Learning Platforms”. In: *Computer Communication and Networks (ICCCN), 2017 26th International Conference on*. IEEE. 2017, pp. 1–9.