Scaling Blockchain Consensus via a Robust Shared Mempool

Fangyu Gai\textsuperscript{1,1}, Jianyu Niu\textsuperscript{2,1}, Ivan Beschastnikh\textsuperscript{3}, Chen Feng\textsuperscript{1}, Sheng Wang\textsuperscript{4}
\textsuperscript{1}\{fangyu.gai, chen.feng\}@ubc.ca, \textsuperscript{2}niujy@sustech.edu.cn, \textsuperscript{3}bestchai@cs.ubc.ca, \textsuperscript{4}sh.wang@alibaba-inc.com

University of British Columbia (\textsuperscript{1}Okanagan Campus, \textsuperscript{3}Vancouver Campus)\textsuperscript{2}
Southern University of Science and Technology, \textsuperscript{4}Alibaba Group

Abstract—Leader-based Byzantine fault-tolerant (BFT) consensus protocols used by permissioned blockchains have limited scalability and robustness. To alleviate the leader bottleneck in BFT consensus, we introduce Stratus, a robust shared mempool protocol that decouples transaction distribution from consensus. Our idea is to have replicas disseminate transactions in a distributed manner and have the leader only propose transaction ids. Stratus uses a provably available broadcast (PAB) protocol to ensure the availability of the referenced transactions. To deal with unbalanced load across replicas, Stratus adopts a distributed load balancing protocol.

We implemented and evaluated Stratus by integrating it with state-of-the-art BFT-based blockchain protocols. Our evaluation of these protocols in both LAN and WAN settings shows that Stratus-based protocols achieve 5× to 20× higher throughput than their native counterparts in a network with hundreds of replicas. In addition, the performance of Stratus degrades gracefully in the presence of network asynchrony, Byzantine attackers, and unbalanced workloads.

Index Terms—Blockchain, Byzantine fault-tolerance, leader bottleneck, shared mempool.

I. INTRODUCTION

The emergence of blockchain technology has revived interest in Byzantine fault-tolerant (BFT) systems \cite{1, 2, 3, 4, 5}. Unlike traditional distributed databases, BFT systems (or blockchains) provide data provenance and allow federated data processing in untrusted and hostile environments \cite{6, 7}. This enables a rich set of decentralized applications, in e.g., finance \cite{8}, gaming \cite{9}, healthcare \cite{10}, and social media \cite{11}. Many companies and researchers are seeking to build enterprise-grade blockchain systems \cite{12, 13, 14, 15} to provide Internet-scale decentralized services \cite{16}.

The core of a blockchain system is the BFT consensus protocol, which allows distrusting parties to replicate and order a sequence of transactions. Many BFT consensus protocols \cite{17, 18, 19, 20} adopted by permissioned blockchains follow the classic leader-based design of PBFT \cite{21}: only the leader node determines the order to avoid conflicts. We call such protocols leader-based BFT protocols, or LBF.

In the normal case (Byzantine-free), an LBFT consensus instance roughly consists of a proposing phase and a commit phase. In the proposing phase, the leader pulls transactions from its local transaction pool (or mempool), forms a proposal, and broadcasts the proposal to the other replicas. On receiving a proposal, replicas verify the proposal content before entering the commit phase. In the commit phase, the leader coordinates multiple rounds of message exchanges to ensure that all correct replicas commit the same proposal at the same position. If the leader behaves in a detectable Byzantine manner, a view-change sub-protocol will be triggered to replace the leader with one of the replicas.

A key scalability challenge for LBFT is the leader bottleneck. Since the proposing and commit phases are both handled by the leader, adding replicas increases the load on the leader and reduces performance. For example, in a LAN environment, the throughput of LBFT protocols drops from 120K tps (transaction per second) with 4 replicas to 20K tps with 64 replicas, while the transaction latency surged from 9 milliseconds to 3 seconds \cite{22}. This has also been documented by other work \cite{23, 24, 25}.

Prior work has focused on increasing LBFT performance by improving the commit phase, e.g., reducing message complexity \cite{19}, truncating communication rounds \cite{26}, and enhancing tolerance to Byzantine faults \cite{27, 28}. Recent works \cite{23, 25} reveal that a more significant factor limiting LBFT’s scalability lies in the proposing phase, in which a proposal with batched transaction data (e.g., 10 MB) is disseminated by the single leader node, whereas messages exchanged in the commit phase (e.g., signatures, hashes) are much smaller (e.g., 100 B). Formal analysis in the Appendix of our technical report \cite{29} shows that reducing the message complexity of the commit phase cannot address this scalability issue.

More broadly, previous works to address the leader bottleneck have proposed horizontal scaling or sharing the blockchain into shards that concurrently run consensus \cite{30, 31, 32, 2}. These approaches require a large network to ensure safety \cite{33} and demand meticulous coordination for cross-shard transactions. By contrast, vertical scaling approaches employ hierarchical schemes to send out messages and collect votes \cite{34, 35}. Unfortunately, this increases latency and requires complex re-configuration to deal with faults.

In this paper, we follow neither of the above strategies. Instead, we introduce the shared mempool (SMP) abstraction, which decouples transaction distribution from consensus, leaving consensus with the job of ordering transaction ids. SMP allows every replica to accept and disseminate client transactions so that the leader only needs to order transaction ids. Applying SMP reaps the following benefits. First, SMP
reduces the proposal size and increases throughput. Second, SMP decouples the transaction synchronization from ordering so that non-leader replicas can help with transaction distribution. Lastly, SMP can be integrated into existing systems without changing the consensus core.

SMP has been used to improve scalability [36], [23], [25], but prior work has passed over two challenges. Challenge 1: ensuring the availability of transactions referenced in a proposal. When a replica receives a proposal, its local mempool may not contain all the referenced transactions. These missing transactions prevent consensus from entering the commit phase, which may cause frequent view-changes (Section VII-C). Challenge 2: dealing with unbalanced load across replicas. SMP distributes the load from the leader and lets each replica disseminate transactions. But, real workloads are highly skewed [37], overwhelming some replicas and leaving others under-utilized (Section VII-D). Existing SMP protocols ignore this and assume that each client sends transactions to a uniformly random replica [25], [23], [36], but this assumption does not hold in practical deployments [38], [39], [40], [41].

We address these challenges with Stratus, an SMP implementation that scales leader-based blockchains to hundreds of nodes. Stratus introduces a provably available broadcast (PAB) primitive to ensure the availability of transactions referenced in a proposal. With PAB, consensus can safely enter the commit phase and not block on missing transactions. To deal with unbalanced workloads, Stratus uses a distributed load-balancing (DLB) co-designed with PAB. DLB dynamically estimates a replica’s workload and capacity so that overloaded replicas can forward their excess load to under-utilized replicas. To summarize, we make the following contributions:

- We introduce and study a shared mempool abstraction that decouples network-based synchronization from ordering for leader-based BFT protocols. To the best of our knowledge, we are the first to study this abstraction explicitly.
- To ensure the availability of transactions, we introduce a broadcast primitive called PAB, which allows replicas to process proposals without waiting for transaction data.
- To balance load across replicas, we introduce a distributed load-balancing protocol co-designed with PAB, which allows busy replicas to transfer their excess load to under-utilized replicas.
- We implemented Stratus and integrated it with HotStuff [19], Streamlet [20], and PBFT [21]. We show that Stratus-based protocols substantially outperform the native protocols in throughput, reaching up to 5× and 20× in typical LANs and WANs with 128 replicas. Under unbalanced workloads, Stratus achieves up to 10× more throughput.

II. RELATED WORK

One classic approach that relieves the load on the leader is horizontal scaling, or sharding [31], [2], [30]. However, using sharding in BFT consensus requires inter-shard and intra-shard consensus, which adds extra complexity to the system. An alternative, vertical scaling technique has been used in PigPaxos [34], which replaced direct communication between a Paxos leader and replicas with relay-based message flow.

Recently, many scalable designs have been proposed to bypass the leader bottleneck. Algorand [15] can scale up to tens of thousands of replicas using Verifiable Random Functions (VRFs) [42] and a novel Byzantine agreement protocol called BA×. For each consensus instance, a committee is randomly selected via VRFs to reach consensus on the next set of transactions. Some protocols such as HoneyBadger [43] and Dumbo [44] adopt a leader-less design in which all the replicas contribute to a proposal. They are targeting on consensus problems under asynchronous networks, while our proposal is for partially synchronous networks. Multi-leader BFT protocols [24], [45], [46] have multiple consensus instances run concurrently, each led by a different leader. Multi-leader BFT protocols such as MirBFT [46] and RCC [24] use multiple consensus instances that are run concurrently by different leaders. These protocols follow a monolithic approach and introduce mechanisms in the view-change procedure to deal with the ordering across different instances and during failures. These additions render a BFT system more error-prone and inefficient in recovery. Stratus-enabled protocols are agnostic to the view-change since Stratus does not modify the consensus core.

Several proposals address the leader bottleneck in BFT, and we compare these in Table I. Tendermint uses gossip to shed the load from the leader. Specifically, a block proposal is divided into several parts and each part is gossiped into the network. Replicas reconstruct the whole block after receiving all parts of the block. The most recent work, Kauri [35], follows the vertically scaling approach by arranging nodes in a tree to propagate transactions and collect votes. It leverages a pipelining technique and a novel re-configuration strategy to overcome the disadvantages of using a tree structure. However, Kauri’s fast re-configuration requires a large fan-out parameter (that is at least larger than the number of expected faulty replicas), which constrains its ability to load balance. In general, tree-based approaches increase latency and require complex re-configuration strategies to deal with faults.

To our knowledge, S-Paxos [36] is the first consensus protocol to use a shared Mempool (SMP) to resolve the leader bottleneck. S-Paxos is not designed for Byzantine failures. Leopard [25] and Narwhal [23] utilize SMP to separate transaction dissemination from consensus and are most similar to our work. Leopard modifies the consensus core of PBFT to allow different consensus instances to execute in parallel, since transactions may not be received in the order that proposals are proposed. However, Leopard does not guarantee that the referenced transactions in a proposal will be available. It also does not scale well when the load across replicas is unbalanced. Narwhal [23] is a DAG-based Mempool protocol. It employs reliable broadcast (RB) [47] to reliably disseminate transactions and uses a DAG to establish a causal relationship among blocks. Narwhal can make progress even if the consensus protocol is stuck. However, RB incurs quadratic message complexity and Narwhal only scales well
when the nodes running the Mempool and nodes running the consensus are located on separate machines. Our work differs from prior systems by contributing (1) an efficient and resilient broadcast primitive, along with (2) a co-designed load balancing mechanism to handle uneven workloads.

III. SHARED MEMPOOL OVERVIEW

We propose a shared mempool (SMP) abstraction that decouples transaction dissemination from consensus to replace the original mempool in leader-based BFT protocols. This decoupling idea enables us to use off-the-shelf consensus protocols rather than designing a scalable protocol from scratch.

A. System Model

We consider two roles in the BFT protocol: leader and replica. A replica can become a leader replica via view-changes or leader-rotation. We inherit the Byzantine threat model and communication model from general BFT protocols [21], [19]. In particular, there are $N \geq 3f + 1$ replicas in the network and at most $f$ replicas are Byzantine. The network is partially synchronous, whereby a known bound $\Delta$ on message transmission holds after some unknown Global Stabilization Time (GST) [48].

We consider external clients that issue transactions to the system. We assume that each transaction has a unique ID and that every client knows about all the replicas (e.g., their IP addresses). We also assume that each replica knows, or can learn, the leader for the current view. Clients can select replicas based on network delay measurements, a random hash function, or another preference. Byzantine replicas can censor transactions, however, so a client may need to switch to another replica (using a timeout mechanism) until a correct replica is found. We assume that messages sent in our system are cryptographically signed and authenticated. The adversary cannot break these signatures.

We further assume that clients send each transaction to exactly one replica, but they are free to choose the replica for each transaction. Byzantine clients can perform a duplicate attack by sending identical transactions to multiple replicas. We consider these attacks out of scope. In future work we plan to defend against these attacks using the bucket and transaction partitioning mechanism from MirBFT [46].

B. Abstraction

A mempool protocol is a built-in component in a consensus protocol, running at every replica. The mempool uses the ReceiveTx(tx) primitive to receive transactions from clients and store them in memory (or to disk, if necessary). If a replica becomes the leader, it calls the MakeProposal() primitive to pull transactions from the mempool and constructs a proposal for the subsequent consensus process. In most existing cryptocurrencies and permissioned blockchains [49], [13], [18], the MakeProposal() primitive generates a full proposal that includes all the transaction data. As such, the leader bears the responsibility for transaction distribution and consensus coordination, leading to the leader bottleneck.

To relieve the leader’s burden of distributing transaction data, we propose a shared mempool (SMP) abstraction, which has been used in the previous works [50], [25], [23], but has not been systematically studied. The SMP abstraction enables the transaction data to be first disseminated among replicas, and then small-sized proposals containing only transaction ids are produced by the leader for replication. In addition, transaction data can be broadcast in batches with a unique id for each batch. This further reduces the proposal size. The SMP abstraction requires the following properties:

- **SMP-Inclusion**: If a transaction is received and verified by a correct replica, then it is eventually included in a proposal.
- **SMP-Stability**: If a transaction is included in a proposal by a correct leader, then every correct replica eventually receives the transaction.

The above two liveness properties ensure that a valid transaction is eventually replicated among correct replicas. Particularly, SMP-Inclusion ensures that every valid transaction is eventually proposed while SMP-Stability, first mentioned in [36], ensures that every proposed transaction is eventually available at all the correct replicas. The second property makes SMP non-trivial to implement in a Byzantine environment; we elaborate on this in Section III-E. We should note that a BFT consensus protocol needs to ensure that all the correct replicas maintain the same history of transaction, or safety. Using SMP does not change the order of committed transactions. Thus, the safety of the consensus protocol is always maintained.

C. Primitives and Workflow

The implementation of the SMP abstraction modifies the two primitives ReceiveTx(tx) and MakeProposal() used in the traditional Mempool and adds two new primitives ShareTx(tx) and FillProposal(p) as follows:

- **ReceiveTx(tx)** is used to receive an incoming tx from a client or replica, and stores it in memory (or disk if necessary).
- **ShareTx(tx)** is used to distribute tx to other replicas.
- **MakeProposal()** is used by the leader to pull transactions from the local mempool and construct a proposal with their ids.
- **FillProposal(p)** is used when receiving a new proposal $p$. It pulls transactions from the local mempool according to the transaction ids in $p$ and fills it into a full proposal. It returns missing transactions if there are any.

Next, we show how these primitives work in an order-execute (OE) model, where transactions are first ordered through a consensus engine (using leader-based BFT consensus protocols) and then sent to an executor for execution. We
argue that while for simplicity our description hinges on an OE model, the principles could also be used in an execute-order-validate (EOV) model that is adopted by Hyperledger [3].

We use two primitives from the consensus engine, which are Propose(p) and Commit(p). The leader replica uses Propose(p) to broadcast a new proposal p and Commit(p) to commit p when the order of p is agreed on across the replicas (i.e., total ordering). As illustrated in Figure 1, the transaction processing in state machine replication using SMP consists of the following steps:

1. Upon receiving a new transaction \( tx \) from the network, a replica calls ReceiveTx(tx) to add \( tx \) to the mempool, and 2. disseminates \( tx \) by calling ShareTx(tx) if \( tx \) is from a client (avoiding re-sharing if \( tx \) is from a replica).
2. Once the replica becomes the leader, it obtains a proposal (with transaction ids) \( p \) by calling MakeProposal(), and 4. proposes it via Propose(p).
5. Upon receipt of a proposal \( p \), a non-leader replica calls FillProposal(p) to reconstruct \( p \) (pulling referenced transaction from the mempool), which is sent to the consensus engine to continue the consensus process.
6. The consensus engine calls Commit(p) to send committed proposals to the executor for execution.

**D. Data Structure**

**Microblock.** Transactions are collected from clients and batched into microblocks for dissemination. This is to amortize the verification cost. Recall that we assume a client only sends a request to a single replica, which makes the microblocks sent from a replica disjoint from others. Each microblock has a unique id calculated from the transaction ids it contains.

**Proposal.** The MakeProposal() primitive generates a proposal that consists of an id list of the microblocks and some metadata (e.g., the hash of the previous block, root hash of the microblocks).

**Block.** A block is obtained by calling the FillProposal(p) primitive. If all the microblocks referenced in a proposal \( p \) can be found in the local mempool, we call it a full block, or a full proposal. Otherwise, we call it a partial block/proposal. A block contains all the data included in the relevant proposal and a list of microblocks.

**E. Challenges and Solutions**

Here we discuss two challenges and corresponding solutions in implementing our SMP protocol.

**Problem-I: missing transactions lead to bottlenecks.** Using best-effort broadcast [51] to implement ShareTx(tx) cannot ensure SMP-Stability since some referenced transactions (i.e., microblocks) in a proposal might never be received due to Byzantine behavior [25]. Even in a Byzantine-free case, it is possible that a proposal arrives earlier than some of the referenced transactions. We call these transactions missing transactions. Figure 2 illustrates an example in which a Byzantine broadcaster \( (R_5) \) only shares a transaction \((tx_1)\) with the leader \((R_1)\), not the other replicas. Therefore, when \( R_1 \) includes \( tx_1 \) in a proposal, \( tx_1 \) will be missing at the receiving replicas. On the one hand, missing transactions block the consensus instance because the integrity of a proposal depends on the availability of the referenced transactions, which is essential to the security of a blockchain. This could cause frequent view-changes which significantly affect performance, as we will show in Section VII-C. On the other hand, to ensure SMP-Stability, replicas have to proactively fetch missing transactions from the leader. This, however, creates a new bottleneck. It is also difficult for the leader to distinguish between legitimate and malicious transaction requests.

A natural solution to address the above challenge is to use reliable broadcast (RB) [23] to implement ShareTx(tx). However, Byzantine reliable broadcast has quadratic message complexity and needs three communication rounds (round trip delay) [51], which is not suitable for large-scale systems. We observe that some properties of reliable broadcast are not needed by SMP since they can be provided by the consensus protocol itself (i.e., consistency and totality). This enlightens us to seek for a lighter broadcast primitive.

**Solution-I: provably available broadcast.** We resolve this problem by introducing a provably available broadcast (PAB) primitive to ensure the availability of transactions referenced in a proposal with negligible overhead. PAB provides an API to generate an availability proof with at least \( f + 1 \) signatures. Since at most \( f \) signatures are Byzantine, the availability proof guarantees that at least one correct replica (excluding the sender) has the message. This guarantees that the message can be eventually fetched from at least one correct replica. As such, by using PAB in Stratus, if a proposal contains valid available proofs for each referenced transaction, it can be passed directly to the commit phase without waiting for the transaction contents to arrive. Therefore, missing transactions can be fetched using background bandwidth without blocking the consensus.

**Problem-II: unbalanced workload/bandwidth distribution.** In deploying a BFT system across datacenters, it is difficult to ensure that all the nodes have identical resources. Even if all the nodes have similar resources, it is unrealistic to assume that they will have a balanced workload in time and space. This is because clients are unevenly distributed across regions and tend to use a preferred replica (nearest or most trusted). In
Fig. 2: In a system with SMP, consisting of 5 replicas in which $R_2$ is Byzantine and $R_1$ is the current leader.

these cases, replicas with a low ratio of workload to bandwidth become bottlenecks.

To address the heterogeneity in workload/bandwidth, one popular approach is gossip [52], [53], [15]: the broadcaster randomly picks some of its peers and sends them the message, and the receivers repeat this process until all the nodes receive the message with high probability. Despite their scalability, gossip protocols have a long tail-latency (the time required for the last node to receive the message) and high redundancy.

**Solution-II: distributed load balancing.** We address the challenge by introducing a distributed load-balancing (DLB) protocol that is co-designed with PAB. DLB works locally at each replica and dynamically estimates a replica’s local workloads and capacities so that overloaded replicas can forward their excess load (microblocks) to under-utilized replicas (proxies). A proxy can disseminate a certain microblock on behalf of the original sender and prove that a microblock is successfully distributed by submitting available proof to the sender. If the proof is not submitted in time, the sender picks another under-utilized replica and repeats the process.

**IV. TRANSACTION DISSEMINATION**

We now introduce a new broadcast primitive called provably available broadcast (PAB) for transaction dissemination, which mitigates the impact of missing transactions (Problem-I). Every replica in Stratus runs PAB to distribute microblocks and collect availability proofs (threshold signatures). When a replica becomes the leader, it pulls microblock ids as well as corresponding proofs into a proposal. This ensures that every receiving replica will have an availability proof for all the referenced microblocks in a valid proposal. These proofs resolve Problem I (Section III-E) by providing **PAB-Provable Availability.** This ensures that a replica will eventually receive all the referenced microblocks and it does not need to wait for missing microblocks to arrive.

Broadcasting microblocks and collecting proofs is a distributed process that is not on the critical path of consensus. As a result, they will not increase latency. In fact, we found that PAB significantly improves throughput and latency (Figure 5).

**A. Provably Available Broadcast**

In PAB, the sending replica, or sender, $s$ broadcasts a message $m$, collects acknowledgements of receiving the message $m$ from other replicas, and produces a succinct proof $\sigma$ (realized via threshold signature [54]) over $m$, showing that $m$ is available to at least one correct replica, say $r$. Eventually, other replicas that do not receive $m$ from $s$ retrieves $m$ from $r$. Formally, PAB satisfies the following properties:

**PAB-Integrity:** If a correct replica delivers a message $m$ from sender $s$, and $s$ is correct, then $m$ was previously broadcast by $s$.

**PAB-Validity:** If a correct sender broadcasts a message $m$, then every correct replica eventually delivers $m$.

**PAB-Provable Availability:** If a correct replica $r$ receives a valid proof $\sigma$ over $m$, then $r$ eventually delivers $m$.

We divide the algorithm into two phases, the push phase and the recovery phase. The communication pattern is illustrated in Figure 3. We use angle brackets to denote messages and events and assume that messages are signed by their senders. In the push phase, the sender broadcasts a message $m$ and each receiver (including the sender) sends a PAB-Ack message $\langle$PAB-Ack$m, id|\rangle$ back to the sender. As long as the sender receives at least a quorum of $q = f + 1$ PAB-Ack messages (including the sender) from distinct receivers, it produces a succinct proof $\sigma$ (realized via threshold signature), showing that $m$ has been delivered by at least one correct replica. The recovery phase begins right after $\sigma$ is generated, and the sender broadcasts the proof message $\langle$PAB-Proof$|id, \sigma|\rangle$. If some replica $r$ receives a valid PAB-Proof without receiving $m$, $r$ fetches $m$ from other replicas in a repeated manner.

Algorithm 1 shows the push phase, which consists of two rounds of message exchanges. In the first round, the broadcaster disseminates $m$ via Broadcast() when the PAB-BROADCAST event is triggered. Note that a replica triggers PAB-BROADCAST only if $m$ is received from a client to avoid re-sharing (Line 8). We use $C$ to denote the client set and $R$ to denote the replica set. In the second round, every replica that receives $m$ acts as a witness by sending the sender a PAB-Ack message over $m, id$ (including the signature). If the sender receives at least $q$ PAB-Ack messages for $m$ from distinct replicas, it generates a proof $\sigma$ from associated signatures via threshold-sign() and triggers a PAB-AVA event. The value of $q$ will be introduced shortly.
Algorithm 1 PAB with message \( m \) at \( R_i \) (push phase)

1: Local Variables:
2: \( S \leftarrow \{\} \) \( \triangleright \) signature set over \( m.id \)
3: \( q \leftarrow f + 1 \) \( \triangleright \) quorum value adjustable between \([f + 1, 2f + 1]\)

4: upon event (PAB-Broadcast\([m] \)) do Broadcast((PAB-Msg\([m, R_i]\))
5: upon receipt (PAB-Msg\([m, s]\)) for the first time do \( \triangleright \) \( s \in C \cup R \)
6: Store\([m] \) \( \triangleright \) for future request
7: if \( s \in C \) then trigger (PAB-Broadcast\([m]\))
8: else Send\([s, (PAB-Ack\([m.id, R_i]\))] \)

10: upon receipt (PAB-Ack\([id, s_j]\)) do \( \triangleright \) if \( R_i \) is the sender
11: \( S \leftarrow S \cup s_j \)
12: if \( |S| \geq q \) then \( \triangleright \) satisfies the quorum condition
13: \( \sigma \leftarrow \text{threshold-sign}(S) \)
14: trigger (PAB-Ava\([id, \sigma]\))

The recovery phase serves as a backup in case the Byzantine senders only send messages to a subset of replicas or if messages are delayed due to network asynchrony. The pseudocode of the recovery phase is presented in Algorithm 2. The sender broadcasts the proof \( \sigma \) of \( m \) on event PAB-Ava. After verifying \( \sigma \), the replica that has not received the content of \( m \) invokes the PAB-Fetch() procedure, which sends PAB-Request messages to a subset of replicas that are randomly picked from signers of \( \sigma \) (excluding replicas that have been requested). The function random\([0,1]\) returns a random real number between 0 and 1. The configurable parameter \( \alpha \) denotes the probability that a replica is requested. If the message is not fetched in \( \delta \) time, the PAB-Fetch() procedure will be invoked again and the timer will be reset.

Although we use \( q = f + 1 \) as the stability parameter in the previous description of PAB, the threshold is adjustable between \( f + 1 \) and \( 2f + 1 \) without hurting PAB’s properties. The upper bound is \( 2f + 1 \) because there are \( N \geq 3f + 1 \) replicas in total, where up to \( f \) of them are Byzantine. In fact, \( q \) captures a trade-off between the efficiency of the push and recovery phases. A larger \( q \) value improves the recovery phase since it increases the chance of fetching the message from a correct replica. But, a larger \( q \) increases latency, since it requires that the replica waits for more acks in the push phase.

B. Using PAB in Stratus

Now we discuss how we use PAB in our Stratus Mempool and how it is integrated with a leader-based BFT protocol. Recall Figure 1 that shows the interactions between the shared mempool and the consensus engine in the Propose phase. Specifically, (i) the leader makes a proposal by calling MakeProposal(), and (ii) upon a replica receiving a new proposal \( p \), it fills \( p \) by calling FillProposal\([p]\). Here we present the implementations of the MakeProposal() and FillProposal\([p]\) procedures as well as the logic for handling an incoming proposal in Algorithm 3. The consensus events and messages are denoted with CE.

Since transactions are batched into microblocks for dissemination, we use microblocks (i.e., \( mb \)) instead of transactions in our description. The consensus protocol subscribes PAB-Deliver events and PAB-Proof messages from the under-
lying PAB protocol and modifies the handlers, in which we use \( mbMap \), \( pMap \), and \( avaQue \) for bookkeeping. Specifically, \( mbMap \) stores microblocks upon the PAB-DELIVER event (Line 9). Upon the receipt of PAB-Proof messages, the microblock id is pushed into the queue \( avaQue \) (Line 8) and the relevant proof \( \sigma \) is recorded in \( pMap \) (Line 7).

We assume the consensus protocol proceeds in views, and each view has a designated leader. A new view is initiated by a CE-NEWVIEW event. Once a replica becomes the leader for the current view, it attempts to invoke the MakeProposal() procedure, which pulls microblocks (only ids) from the front of \( avaQue \) and piggybacks associated proofs. It stops pulling when the number of contained microblocks has reached \( BLOCKSIZE \), or there are no microblocks left in \( avaQue \). The reason why the proposal needs to include all the associated available proofs of each referenced transaction is to show that the availability of each referenced microblock is guaranteed. We argue that the inevitable overhead is negligible provided that the microblock is large.

On the receipt of an incoming proposal \( p \), the replica verifies every proof included in \( p.payload \) and triggers a CE-VIEWCHANGE event if the verification is not passed, attempting to replace the current leader. If the verification is passed, a \( \langle CE-ENTER\text{COMMIT}|p \rangle \) event is triggered and the processing of \( p \) enters the commit phase (Line 26). Next, the replica invokes the FillProposal(\( p \)) procedure to pull the content of microblocks associated with \( p.payload \) from the mempool. The PAB-Fetch(id) procedure (Algorithm 2) is invoked when missing microblocks are found. The thread waits until all the requested microblocks are delivered. Note that this thread is independent of the thread handling consensus events. Therefore, waiting for requested microblocks will not block consensus. After a full block is constructed, the replica triggers a \( \langle CE-FULL|\text{block} \rangle \) event, indicating that the block is ready for execution.

In Stratus, the transactions in a microblock are executed if and only if all transactions in the previous microblocks are received and executed. Since missing transactions are fetched according to their unique ids, consistency is ensured. Therefore, using Stratus in any case will not compromise the correctness analysis is in our technical report [29].

V. LOAD BALANCING

We now discuss Stratus’ load balancing. Recall that replicas disseminate transactions in a distributed manner. But, due to network heterogeneity and workload imbalance (Problem II), performance will be bottlenecked by overloaded replicas. Furthermore, a replica’s workload and its resources may vary over time. Therefore, a load balancing protocol that can adapt to a replica’s workload and capacity is necessary.

In our design, busy replicas will forward excess load to less busy replicas that we term proxies. The challenges are (i) how to determine whether a replica is busy, (ii) how to decide which replica should receive excess loads, and (iii) how to deal with Byzantine proxies that refuse to disseminate the received load.

Our load balancing protocol works as follows. A local workload estimator monitors the replica to determine if it is busy or unbusy. We discuss work estimation in Section V-B. Next, a busy replica forwards newly generated microblocks to a proxy. The proxy initiates a PAB instance with a forwarded microblock and is responsible for the push phase. When the push phase completes, the proxy sends the PAB-Proof message of the microblock to the original replica, which continues the recovery phase. In addition, we adopt a banList to avoid Byzantine proxies. Next, we discuss how a busy replica forwards excess load.

A. Load Forwarding

Before forwarding excess load, a busy replica needs to know which replicas are unbusy. A naive approach is to ask other replicas for their load status. However, this requires all-to-all communications and is not scalable. Instead, we use the well-known Power-of-d-choices (Pod) algorithm [57], [58], [59]. A busy replica randomly samples load status from \( d \) replicas, and forwards its excess load to the least loaded replica (the proxy). Here, \( d \) is usually much smaller than the number of replicas \( N \). Our evaluation shows that \( d = 3 \) is sufficient for a network with hundreds of nodes and unbalanced workloads (see Section VII-D). Note that the choice of \( d \) is independent of \( f \); we discuss how we handle Byzantine proxies later in this section. The randomness in Pod ensures that the same proxy is unlikely to be re-sampled and overloaded.

Algorithm 4 depicts the LB-ForwardLoad procedure and relevant handlers. Upon the generation of a new microblock \( mb \), the replica first checks whether it is busy (see Section V-B). If so, it invokes the LB-ForwardLoad(\( mb \)) procedure to forward \( mb \) to the proxy; otherwise, it broadcasts \( mb \) using PAB by itself. To select a proxy, a replica samples load status from \( d \) random replicas (excluding itself) within a timeout of \( \tau \) (Line 10). Upon receiving a workload query, a replica obtains its current load status by calling the GetLoadStatus() (see Section V-B) and piggybacks it on the reply (Line 23-25). If the sender receives all the replies or times out, it picks the replica that replied with the smallest workload and sends \( mb \) to it. This proxy then initiates a PAB instance for \( mb \) and sends the PAB-Proof message back to the original sender when a valid proof over \( mb \) is generated. Note that if no replies are received before timeout, the sending replica initiates a PAB instance by itself (Line 13). Note that due to the decoupling design of Stratus, the overhead introduced by load forwarding has negligible impact on consensus. To prevent a malicious
Algorithm 4 The Load Forwarding procedure at replica \(R_i\)

1. Local Variables:
2. \(\text{samples} \leftarrow \{\}\) \(\triangleright\) stores sampled info for a microblock
3. \(\text{banList} \leftarrow \{\}\) \(\triangleright\) stores potentially Byzantine proxies

4. \textbf{upon event} \(\text{NEWMB}(mb)\) do
5. \hspace{1em} if \(\text{IsBusy}(mb)\) do \(\text{LB-ForwardLoad}(mb)\)
6. \hspace{1em} else trigger (PAB-BROADCAST\(mb)\)

7. \textbf{procedure} \(\text{LB-ForwardLoad}(mb)\) \(\triangleright\) if \(R_i\) is the busy sender
8. \hspace{1em} \text{starttimer}(\text{Sample}(\tau, mb, id))
9. \hspace{1em} \(K \leftarrow \text{SampleTargets}(d)\ \backslash \text{banList}\)
10. \hspace{1em} for all \(r \in K\) do \(\text{Send}(r, \langle \text{LB-Query}(mb, id, R_i)\rangle)\)
11. \hspace{1em} wait until \(|\text{samples}(mb, id)| = 0 \) or \(\tau\) timeout do
12. \hspace{1em} if \(|\text{samples}(mb, id)| = 0\) then
13. \hspace{2em} trigger (PAB-BROADCAST\(mb)\)
14. \hspace{1em} return
15. \hspace{1em} find \(r_p \in \text{samples}(mb, id)\) with the smallest \(w\)
16. \hspace{1em} \text{starttimer}(\text{Forward}(\tau', mb))
17. \hspace{1em} \(\text{banList}\!\!.\text{Append}(r_p)\) \(\triangleright\) every proxy is put in \text{banList}
18. \hspace{1em} \text{Send}(r_p, \langle \text{LB-ForwardLoad}(mb)\rangle)\) \(\triangleright\) send \(mb\) to the proxy
19. \hspace{1em} wait until \(\text{PAB-Proof}\) over \(mb\) is received or \(\tau'\) timeout do
20. \hspace{1em} if \(\tau'\) do \(\text{LB-ForwardLoad}(mb)\)
21. \hspace{1em} else \(\text{banList}\!\!.\text{Remove}(r_p)\) \(\triangleright\) \(R_p\) is removed from \text{banList}

22. \textbf{upon receipt} \(\langle \text{LB-ForwardLoad}(mb, r)\rangle\) do \(\triangleright\) if \(R_i\) is the proxy with \(mb\)
23. \hspace{1em} trigger (PAB-BROADCAST\(mb)\)

24. \textbf{upon receipt} \(\langle \text{LB-Query}(id, r)\rangle\) do \(\triangleright\) if \(R_i\) is sampled
25. \hspace{1em} \(w \leftarrow \text{GetLoadStatus}()\)
26. \hspace{1em} \text{Send}(r, \langle \text{LB-Info}([w, id, R_i])\rangle)

27. \textbf{upon receipt} \(\langle \text{LB-Info}(w, id, r)\rangle\) do \(\triangleright\) if \(R_i\) is busy
28. \hspace{1em} \(\text{samples}[id][R_i] \leftarrow w\)

29. \textbf{upon receipt} \(\langle \text{PAB-Proof}(id, \sigma)\rangle\) before \(\tau'\) timeout do
30. \hspace{1em} if \(\text{threshold-verify}(id, \sigma)\) is not true do return
31. \hspace{1em} trigger (PAB-AVA\(id, \sigma)\) \(\triangleright\) \(R_i\) takes over the recovery phase

32. \textbf{upon event} \(\langle \text{Reset}(\text{banList})\rangle\)
33. \hspace{1em} \(\text{banList} \leftarrow \{\}\) \(\triangleright\) clear \text{banList} periodically

Replica from sending a small batch to reduce the performance, every replica can set a minimum batch size for receiving a batch.

In Stratus, each replica randomly and independently chooses \(d\) replicas from the remaining \(N-1\) replicas. Since the workload of each replica changes quickly, the sampling happens for each microblock without blocking the forwarding process. Therefore, for each load balancing event of an overloaded replica \(A\), the probability that a specific replica (other than replica \(A\)) is chosen by replica \(A\) is \(d/(N-1)\). The probability that a replica is chosen by all replicas is very small. For example, when \(d = 3\) and \(N = 100\) the probability that a replica is chosen by more than 7 replicas is about 0.03. We omit the analysis details due to the page limit. Next, we discuss how we handle Byzantine behaviors during load forwarding.

Handling faults. A sampled Byzantine replica can pretend to be unoccupied by responding with a low busy level and censoring the forwarded microblocks. In this case, the SMP-Inclusion would be compromised: the transactions included in the censored microblock will not be proposed. We address this issue as follows. A replica \(r\) sets a timer before sending \(mb\) to a selected proxy \(p\) (Line 16). If \(r\) does not receive the available proof \(\sigma\) over \(mb\) before the timeout, \(r\) re-transmits \(mb\) by re-invoking the \(\text{LB-ForwardLoad}(mb)\) (Line 20). Here, the unique microblock ids prevent duplication. The above procedure repeats until a valid \(\sigma\) over \(mb\) is received. Then \(r\) continues the recovery phase of the PAB instance with \(mb\) by triggering the PAB-AVA event (Line 33).

To prevent Byzantine replicas from being sampled again, we use a \text{banList} to store proxies that have not finished the push phase of a previous PAB instance. That is, before a busy sender sends a microblock \(mb\) to a proxy, the proxy is added to the \text{banList} (Line 17). For future sampling, the replicas in the \text{banList} are excluded. As long as the sender receives a valid proof message for \(mb\) from the proxy before a timeout, the proxy will be removed from the \text{banList} (Line 21). The \text{banList} is periodically cleared by a timer to avoid replicas from being banned forever (Line 33). Note that more advanced \text{banList} mechanisms can be used based on proxies’ behavior [60] and we consider to include them in our future work.

B. Workload Estimation

Our workload estimator runs locally on an on-going basis and is responsible for estimating load status. Specifically, it determines: (i) whether the replica is overloaded, and (ii) how much the replica is overloaded, which correspond to the two functions in Algorithm 4, \text{IsBusy()} and \text{GetLoadStatus()}, respectively. To evaluate replicas’ load status, two ingredients need to be considered: workload and capacity. As well, the estimated results must be comparable across replicas in a heterogeneous network.

To address these challenges, we use stable time \((ST)\) to estimate a replica’s load status. The stable time of a microblock is measured from when the sender broadcasts the microblock until the time that the microblock becomes stable (receiving \(f+1\) acks). To estimate \(ST\) of a replica, the replica calculates the \(ST\) of each microblock if it is the sender and takes the \(n\)-th (e.g., \(n = 95\)) percentile of the \(ST\) values in a window of the latest stable microblocks. The estimated \(ST\) of a replica is updated when a new microblock becomes stable. The window size is configurable and we use 100 as the default size.

Our approach is based on two observations. First, the variability in network delay in a private network is small [61]. Second, network delay increases sharply when a node is overloaded. The above observations are based on our measurements. A selection of these is shown in Figure 4. Figure 4a is a heat map of measured delays between two regions.
(Virginia and Singapore) in Alibaba Cloud over 24 hours. Figure 4b exhibits the round-trip delay distribution during 1 minute starting the 12th hour in the measurements. We omit measurements of other pair of datacenters in this paper. Our results demonstrate that the inter-datacenter network delays across different regions are stable and predictable based on recent measurement data. Thus, under a constant workload, the calculated ST should be at around a constant number $\alpha$ with an error of $\epsilon$. If the estimated ST is larger than $\alpha + \epsilon$ by a parameter of $\beta$, a replica is considered busy (return true in the IsBusy function). Additionally, the value of ST reflects the degree to which a replica is loaded: the smaller the ST, the more resources a replica has for disseminating microblocks. Therefore, we use the ST as the return value of the function GetLoadStatus. Note that the GetLoadStatus returns a NULL value if the calling replica is busy. Also note that due to network topology, the ST value does not faithfully reflect the load status across replicas. For example, some replicas may have a smaller ST because they are closer to a quorum of other replicas. In this case, forwarding excess load to these replicas also benefits the system. For overloaded replicas with large ST values, the topology has a negligible impact. In case the network is unstable, we can also estimate the load status by monitoring the queue length of the network interface card. We save that for future work.

VI. IMPLEMENTATION

We prototyped Stratus$^2$ in Go with Bamboo$^3$, which is an open source project for prototyping, evaluating, and benchmarking BFT protocols. Bamboo provides validated implementations of state-of-the-art BFT protocols such as PBFT$^4$ [21], HotStuff$^5$ [19], and Streamlet$^6$ [20]. Bamboo also supplies common functionalities that a BFT replication protocol needs. In our implementation, we replaced the mempool in Bamboo with Stratus shared mempool. Because of Stratus$^7$ well-designed interfaces, the consensus core is minimally modified. We used HotStuff$^8$’s Pacemaker for view change, though Stratus is agnostic to the view-change mechanism. Similar to [19], [62], we use ECDSA to implement the quorum proofs in PAB instead of threshold signature. This is because the computation efficiency of ECDSA$^4$ is better than Boldyreva$^8$’s threshold signature $[62]$. Overall, our implementation added about 1,300 lines of Go to Bamboo.

Optimizations. Since microblocks consume the most bandwidth, we need to reserve sufficient resources for consensus messages to ensure progress. For this, we adopt two optimizations. First, we prioritize the transmission and processing of consensus messages. Second, we use a token-based limiter to limit the sending rate of data messages: every data message (i.e., microblock) needs a token to be sent out, and tokens are refilled at a configurable rate. This ensures that the network resources will not be overtaken by data messages. The above optimizations are specially designed for Stratus and are only used in Stratus-based implementations. We did not use these optimizations in non-Stratus protocols in our evaluation since they may negatively affect those protocols.

VII. EVALUATION

Our evaluation answers the following questions.

- Q1: how does Stratus perform as compared to the alternative Shared Mempool implementations with a varying number of replicas? (Section VII-B)
- Q2: how do missing transactions caused by network asynchrony and Byzantine replicas affect the protocols’ performance? (Section VII-C)
- Q3: how does unbalanced load affect protocols’ throughput? (Section VII-D)

A. Setup

Testbeds. We conducted our experiments on Alibaba Cloud with ecs.s6-c1m2.xlarge instances$^5$. Each instance has 4vGPUs and 8GB memory and runs Ubuntu server 20.04. We ran each replica on a single ECS instance. We performed protocol evaluations in LANs and WANs to simulate national and regional deployments, respectively [35]. LANs and WANs are typical deployments of permissioned blockchains and permissionless blockchains that run a BFT-based PoS consensus protocol [12], [14]. In LAN deployments, a replica has up to 3 Gbit/s of bandwidth and inter-replica RTT of less than 10 ms. For WAN deployments, we use NetEm$^6$ to simulate a WAN environment with 100 ms inter-replica RTT and 100 Mbit/s replica bandwidth.

Workload. Clients are run on 4 instances with the same specifications. Each client concurrently sends multiple transactions to different replicas. Bamboo’s benchmark provides an in-memory key-value store backed by the protocol under evaluation. Each transaction is issued as a simple key-value set operation submitted to a single replica. Since our focus is on the performance of the consensus protocol with the mempool, we do not involve application-specific verification (including signatures) and execution (including disk IO operations) of transactions in our evaluation. We measure both throughput and latency on the server side. The latency is measured between the moment a transaction is first received by a replica and the moment the block containing it is committed. We avoid end-to-end measurements to exclude the impact of the

| Acronym | Protocol description |
|---------|----------------------|
| N-HS | HotStuff without a shared mempool |
| N-PBFT | Native PBFT without a shared mempool |
| SMP-HS | HotStuff integrated with a simple shared mempool |
| SMP-HS-G | SMP-HS with gossip instead of broadcast |
| SMP-HS-Even | SMP-HS with an even workload across replicas |
| S-HS | HotStuff integrated with Stratus (this paper) |
| S-PBFT | PBFT integrated with Stratus (this paper) |
| Narwhal | HotStuff based shared mempool |
| MirBFT | PBFT based multi-leader protocol |

$^2$Available at https://github.com/gitferry/bamboo-stratus
$^3$Available at https://github.com/gitferry/bamboo
$^4$We trivially concatenate $f + 1$ ECDSA signatures

*Authorized licensed use limited to: The University of British Columbia Library. Downloaded on October 02,2024 at 21:45:35 UTC from IEEE Xplore. Restrictions apply.*
network delay between a replica and a client. Each data point is obtained when the measurement is stabilized (sampled data do not vary by more than 1%) and is an average over 3 runs. In our experiments, workloads are evenly distributed across replicas except for the last set of experiments (Section VII-D), in which we create skewed load to evaluate load balancing.

**Protocols.** We evaluate the performance of a wide range of protocols (Table II). We use native HotStuff and PBFT with the original mempool as the baseline, denoted as (N-HS and N-PBFT, respectively). All of our implementations of HotStuff are based on the Chained-HotStuff (three-chain) version from the original paper [19], in which pipelining is used and leaders are rotated for each proposal. Our implementation of PBFT shares the same chained blockchain structure as Chained-HotStuff for a fair comparison. We also compare against a version of HotStuff with a basic shared mempool with best-effort broadcast and fetching (denoted as SMP-HS). Finally, we equip HotStuff and PBFT with our Stratus Mempool, denoted as S-HS and S-PBFT, respectively. We also implemented a gossip-based shared mempool (distributing microblocks via gossip), denoted by SMP-HS-G, to evaluate load balancing and compare it with S-HS. All protocols are implemented using the same Bamboo code base for a fair comparison. The sampling parameter $d$ is set to 1 by default. This is because $d = 1$ allows the busy sender to randomly pick exactly one replica without comparing workload status between others. When we gradually increase $d$, the chance of selecting a less busy replica increases significantly. However, increasing $d$ also incurs overhead. In our experiments (Section VII-D) we show that $d = 3$ exhibits the best performance.

We also compare against Narwhal\footnote{Available at https://github.com/facebookresearch/narwhal/}, which uses a shared mempool with reliable broadcast. Narwhal is based on HotStuff and splits functionality between workers and primaries, responsible for transaction dissemination and consensus, respectively. To fairly compare Narwhal with Stratus, we let each primary have one worker and locate both in one VM instance. As another baseline, we compare our protocols with MirBFT \footnote{Available at https://github.com/hyperledger-labs/mirbft/tree/research}, a state-of-the-art multi-leader protocol. All replicas act as leaders in an epoch for fair comparison.

**B. Scalability**

We evaluate the scalability of the protocols by increasing the number of replicas from 16 to 400. We use N-HS, N-PBFT, SMP-HS, S-PBFT, Narwhal, and MirBFT for comparison and run experiments in both LANs and WANs. We gradually increase the workload until the system is saturated, i.e., the workload exceeds the maximum system throughput, resulting in sharply increasing delay.

We use a batch size of 256KB and 128-byte transaction payload, which gives 2,000 transactions per batch, for Stratus-based protocols throughout our experiments. We find that proposal size (number of microblock ids included in a proposal) does not have an obvious impact on performance as long as we choose a proper batch size (number of transactions in a microblock). Therefore, we do not constrain proposal size. For every protocol we use a microblock/proposal size settings that maximizes the protocol’s performance. We omit experimental results that explore these settings due to space constraints.

Figure 5 depicts the throughput and latency of the protocols with an increasing number of replicas in LANs and WANs. We can see that protocols using the shared mempool (SMP-HS, S-HS, S-PBFT, and Narwhal) or relying on multiple leaders outperform the native HotStuff and Streamlet (N-HS and N-PBFT) in throughput in all experiments. Previous works [19], [25], [23] have also shown that the throughput/latency of N-HS decreases/increases sharply as the number of replicas increases, and meaningful results can no longer be observed beyond 256 nodes. Although Narwhal outperforms N-HS due to the use of a shared mempool, it does not scale well since it employs the heavy reliable broadcast primitive. As shown in [23], Narwhal achieves better scalability only when each primary has multiple workers that are located in different machines. MirBFT has higher throughput than S-HS when there are fewer than 16 replicas. This is because Stratus imposes a higher message overhead than PBFT. However, MirBFT’s performance drops faster than S-HS because of higher message complexity. MirBFT is comparable to S-PBFT because they have the same message complexity. The gap between them is due to implementation differences.

SMP-HS and S-HS show a slightly higher latency than N-HS when the network size is small ($< 16$ in LANs and $< 32$ in WANs). This is due to batching. They outperform the other two protocols in both throughput and latency when the network size is beyond 64 and show flatter lines in throughput as the network size increases. The throughput of SMP-HS and S-HS achieve $5 \times$ throughput when $N = 128$ as compared to N-HS, and this gap grows with network size. Finally, SMP-HS and S-HS have similar performance, which indicates that the use...
of PAB incurs negligible overhead, which is amortized by a large batch size.

**TABLE III:** Outbound bandwidth consumption comparison (MB/s) with \( N = 64\) replicas. The bandwidth of each replica is throttled to 100 MB/s. The results are collected when the network is saturated.

| Role/Messages | N-HS | SMP-HS | S-HS (this paper) |
|---------------|------|--------|-------------------|
| Leader        |      |        |                   |
| Proposals     | 75.4 | 4.3    | 9.8               |
| Microblocks   | N/A  | 50.5   | 50.5              |
| SUM           | 75.4 | 55.2   | 60.1              |
| Non-leader    |      |        |                   |
| Microblocks   | N/A  | 50.4   | 50.3              |
| Votes         | 0.5  | 2.3    | 2.4               |
| Acks          | N/A  | N/A    | 4.7               |
| SUM           | 0.5  | 22.9   | 27.4              |

**Bandwidth consumption.** We evaluate the outbound bandwidth usage at the leader and the non-leader replica in N-HS, SMP-HS, and S-HS. We present the results in Table III. We can see that the communication bottleneck in N-HS is at the leader, while the bandwidth of non-leader replicas is underutilized. In SMP-HS and S-HS, the bandwidth consumption between leader replicas and non-leader replicas are more even, and the leader bottleneck is therefore alleviated. We observe that S-HS adds around 10% overhead on top of SMP-HS due to the use of PAB. Next, we show that this overhead is worthwhile as it provides availability insurance. We also observe that around 40% of bandwidth remains unused. This is because chain-based protocols are bounded by latency: each proposal goes through two rounds of communication (one-to-all-to-one). We consider out-of-order processing of proposals for better network utilization as important future work.

**C. Impact of Missing Transactions**

Recall that in **Problem-I** (Section III-E), a basic shared mempool with best-effort broadcast is subject to missing transactions. In the next set of experiments, we evaluate the throughput of SMP-HS and S-HS under a period of network asynchrony and Byzantine attacks.

**Network asynchrony.** During network asynchrony, a proposal is likely to arrive before some of referenced transactions (i.e., missing transactions), which negatively impacts performance. The point of this experiment is to show that Stratus-based protocols can make progress during view-changes and are more resilient to network asynchrony.

We ran an experiment in a WAN setting, during which we induce a period of network fluctuation via NetEm. The fluctuation lasts for 10 s, during which network delays between replicas fluctuate between 100 ms and 300 ms for each message (i.e., 200 ms base with 100 ms uniform jitter). We set the view-change timer to be 1000 ms. We keep the transaction rate at 25KTx/s without saturating the network.

We ran the experiment 10 times and each run lasts 30 seconds. We show the results in Figure 6. During the fluctuation, the throughput of SMP-HS drops to zero. This is because missing transactions are fetched from the leader, which causes congestion at the leader. As a result, view-changes are triggered, during which no progress is made. When the network fluctuation is over, SMP-HS slowly recovers by processing the accumulated proposals. On the other hand, S-HS makes progress at the speed of the network and no view-changes are triggered. This is due to the **PAB-Provable Availability** property: no missing transactions need to be fetched on the critical consensus path.

**Byzantine senders.** The attacker’s goal in this scenario is to overwhelm the leader with many missing microblocks.

The strategies for each protocol are described as follows. In SMP-HS, Byzantine replicas only send microblocks to the leader (Figure 2). In S-HS, Byzantine replicas have to send microblocks to the leader and to at least \( f \) replicas to get proofs. Otherwise, their microblocks will not be included in a proposal (consider the leader is correct). In this experiment, we consider two different quorum parameters for PAB (see Section VIII), \( f + 1 \) and \( 2f + 1 \) (denoted by S-HS-f and S-HS-2f, respectively). These variants will explain the tradeoff between throughput and latency. We ran this experiment in a LAN setting with \( N = 100 \) and \( N = 200 \) replicas (including the leader). The number of Byzantine replicas ranged from 0 to 30 (\( N = 100 \)) and 0 to 60 (\( N = 200 \)).

Figure 7 plots the results. As the number of Byzantine replicas increases, the throughput/latency of SMP-HS decreases/increases sharply. This is because replicas have to fetch missing microblocks from the leader before processing a proposal. We also observe a slight drop in throughput of S-HS. The reason is that only background bandwidth is used to deal with missing microblocks. The latency of S-HS remains flat since the consensus will never be blocked by missing microblocks as long as the leader provides correct proofs. In addition, we notice that Byzantine behavior has more impact on larger deployments. With \( N = 200 \) replicas, the perfor-
mance of SMP-HS decreases significantly. The throughput is almost zero when the number of Byzantine replicas is 60 and the latency surges when there are more than 20 Byzantine replicas. Finally, S-HS-2f has better throughput than S-HS-f at the cost of higher latency as the number of Byzantine replicas increases. The reason is that with a larger quorum size, fewer microblocks need to be fetched. However, a replica needs to wait for more acks to generate available proofs.

D. Impact of Unbalanced Workload

Previous work [38], [39], [40], [41] has observed that node degrees in large-scale blockchains have a power-law distribution. As a result, most clients send transactions to a few popular nodes, leading to unbalanced workload (Problem II in Section III-E). In this experiment, we vary the ratio of workload to bandwidth by using identical bandwidth for each replica but skewed workloads across replicas. We use two Zipfian parameters [64], Zipf1 ($s = 1.01, v = 1$) and Zipf10 ($s = 1.01, v = 10$), to simulate a highly skewed workload and a lightly skewed workload, respectively. We show the workload distributions in Figure 8. For example, when $s = 1.01$ and there are 100 replicas, 10% of the replicas will receive over 85% of the load.

We evaluate load-balancing in Stratus using the above distributions in a WAN setting. Stratus samples $d$ replicas to select the least loaded one as the proxy, we consider $d = 1, 2, 3$, denoted by S-HS-d1, S-HS-d2, and S-HS-d3, respectively. We also use SMP-HS-G, HotStuff with a gossip-based shared mempool for comparison. We set the gossip fan-out parameter to 3.

Figure 9 shows protocols’ throughput. We can see that S-HS-dX outperforms SMP-HS and SMP-HS-G in all experiments. S-HS-dX achieves $5 \times (N = 100)$ to $10 \times (N = 400)$ throughput with Zipf1 as compared with SMP-HS. SMP-HS-G does not scale well under a lightly skewed workload (Zipf10) due to the message redundancy. We also observe that S-HS-dX achieves the best performance when $d = 3$, while the gap between different $d$ values is not significant.

Fig. 8: Workload distribution with different network sizes and Zipfian parameters.

Fig. 9: Throughput with different workload distribution.

if a referenced microblock is missing. The microblocks with missing proofs will be discarded after a timeout.

Now consider a Byzantine leader that includes microblocks without availability proofs into a proposal. This will trigger a view-change, which will replace the leader. In some PoS blockchains [12], [13], such leaders are also slashed.

**Attacks on load balancing.** A Byzantine sender can try to congest the network by sending identical microblocks to multiple proxies. To mitigate this attack, we propose a simple solution. When a busy replica $r'$ decides on $r^*$ as the proxy, it forwards the microblock $mb$ to $r^*$ along with a message $\rho$ that contains $r^*$’s signature over $mb$id concatenated with $r^*$’s identity. Then, $r'$ broadcast $mb$ along with $\rho$ using PAB. This allows other replicas to check if a microblock by the same sender is broadcast by different proxies. Once detected, a replica can reject microblocks from this sender or report this behavior by sending evidence to the other replicas. If the proxy fails to complete PAB, the original sender either broadcasts the microblock by itself or waits for a timeout to garbage collect the microblock.

A malicious replica can pretend to be busy and forward its load to other replicas. This can be addressed with an incentive mechanism: a replica that produced the availability proof for a microblock using PAB is rewarded. This information is verifiable because the availability proofs for each microblock are in the proposal and will be recorded on the blockchain if the proposal is committed. In addition, to prevent a malicious sender from overloading a proxy, the proxy can set a limit on its buffer, and reject extra load.

IX. CONCLUSION AND FUTURE WORK

We presented a shared mempool abstraction that solves the leader bottleneck of leader-based BFT protocols. We designed Stratus, a novel shared mempool protocol to address two challenges: missing transactions and unbalanced workloads. Stratus overcomes these with an efficient provably available broadcast (PAB) and a load balancing protocol. For example, HotStuff’s throughput is $5 \times$ to $20 \times$ higher than native HotStuff. In our future work, we plan to extend Stratus to multi-leader BFT protocols.

ACKNOWLEDGEMENTS

This work was partly done during Fangyu Gai’s research internship at Alibaba DAMO Academy. Ivan Beschastnikh was supported by the NSERC Discovery Grant RGPIN-2020-05203. Chen Feng was supported by the NSERC Discovery Grant RGPIN-2016-05310 and CREATE grant 528125-2019.
REFERENCES

[1] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. Falcondb: Blockchain-based collaborative database. In Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14–19, 2020, pages 637–652. ACM, 2020.

[2] Muhammad El-Hindi, Martin Heyden, Carsten Binnig, Ravi Ramamurthy, Avind Arasu, and Donald Kossmann. Blockchaingrid - towards a shared data directory on blockchains. In Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, pages 1905–1908. ACM, 2019.

[3] Elii Androulaki, Artur Barger, Vira Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Davy, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Murallahdaran, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoval Stathakopoulou, Marko Vukolic, Sharan Weed Coco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018, pages 30:1–30:15. ACM, 2018.

[4] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. Slimchain: Scaling blockchain transactions through off-chains storage and parallel processing. Proc. VLDB Endow., 14(11), 2021.

[5] Yehonatan Buchnik and Roy Friedman. Fireledger: A high throughput blockchain consensus protocol. Proc. VLDB Endow., 13(9):1525–1539, 2020.

[6] Pingcheng Ruan, Tien Tuan Anh Dinh, Dumitrel Loghin, Meihui Zhang, Gang Chen, Qian Lin, and Beng Chin Ooi. Blockchains vs. distributed databases: Dichotomy and fusion. In SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, pages 1504–1517. ACM, 2021.

[7] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. AFT ’20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020, pages 1–11. ACM, 2020.

[8] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999, pages 173–186. USENIX Association, 1999.

[9] Fangyu Gai, Ali Farahbaksh, Jianyu Niu, Chen Feng, Ivan Beschastnikh, and Hao Duun. Dissecting the performance of chained-BFT. In 44th IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Virtual, pages 190–200. IEEE, 2021.

[10] George Danezis, Eleftherios Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tulip: A dag-based mempool and efficient BFT consensus. CoRR, abs/2105.11827, 2021.

[11] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. RCC: resilient concurrent consensus for high-throughput secure transaction processing. In 37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021, pages 1392–1403. IEEE, 2021.

[12] Kexin Hu, Kaiwen Guo, Guang Tang, Zhenfeng Zhang, Hao Cheng, and Zhiyao Zhao. Don’t count on one to carry the ball: Scaling BFT without sacrificing efficiency. CoRR, abs/2106.08114, 2021.

[13] Ramakrishna Kotla, Lorenzo Alvissi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Byz: speculative Byzantine fault tolerance. In Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007, pages 45–58. ACM, 2007.

[14] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. Scalable Byzantine consensus via hardware-assisted secret sharing. IEEE Trans. Computers, 68(1):139–151, 2019.

[15] Zhiyun Xiong, Dahlia Malkhi, Kartik Nayak, and Ling Ren. Strengthened fault tolerance in Byzantine fault tolerant replication. CoRR, abs/2101.03715, 2021.

[16] Fangyu Gai, Jianyu Niu, Ivan Beschastnikh, Chen Feng, and Sheng Wang. Devouring the leader bottleneck in BFT consensus. CoRR, abs/2203.05158, 2022.

[17] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Limas Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omnledger: A secure, scale-out, decentralized ledger via sharding. In 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA, pages 583–598. IEEE Computer Society, 2018.

[18] Mohammad Javad Amiri, Divyakant Agrawal, and Amir El Abbadi. Sharper: Sharding permissioned blockchains over network clusters. In SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, pages 76–88. ACM, 2021.

[19] Juaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In Jay R. Lorch and Minlan Yu, editors, 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019, pages 95–112. USENIX Association, 2019.

[20] Bernardo David, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. Gearbox: An efficient uc sharded ledger leveraging the safety-liveness dichotomy. Cryptology ePrint Archive, Report 2021/211, 2021. https://ia.cr/2021/211.

[21] Aleksey Charapko, Alidin Alijijang, and Murat Demirbas. Pigpaxos: Offloading the leader for high throughput state machine replication. In SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, pages 235–247. ACM, 2021.

[22] Ray Neiheiser, Miguel Matos, and Luis E. T. Rodrigues. Kauri: Scalable BFT consensus with pipelined tree-based dissemination and aggregation. In SOSP ’21: ACM SIGOPS 26th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021, pages 1–17. ACM, 2021.

[23] Bao Lu and Edmund L Wong. Zyzzyva: speculative Byzantine fault tolerance. In Proceedings of the 21st ACM Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021, pages 637–652. ACM, 2020.

[24] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. Sax: Offloading the leader for high throughput state machine replication. In IEEE 31st Symposium on Reliable Distributed Systems, SRDS 2012, Irvine, CA, USA, October 8-11, 2012, pages 111–120. IEEE Computer Society, 2012.

[25] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Ray Neiheiser, Miguel Matos, and Luís E. T. Rodrigues. Don’t count on one to carry the ball: Scaling BFT without sacrificing efficiency. CoRR, abs/2106.08114, 2021.
