Scaling Blockchain Consensus via a Robust Shared Mempool

Fangyu Gai[1,†,§], Jianyu Niu[2,†], Ivan Beschastnikh[3], Chen Feng[1], Sheng Wang[4]
University of British Columbia ([1]Okanagan Campus, [3]Vancouver Campus)
[2]Southern University of Science and Technology [4]Alibaba Group
[1]{fangyu.gai, chen.feng}@ubc.ca [2]niujy@sustech.edu.cn [3]bestchai@cs.ubc.ca [4]sh.wang@alibaba-inc.com

ABSTRACT

There is a resurgence of interest in Byzantine fault-tolerant (BFT) systems due to blockchains. However, leader-based 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 to 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 that is co-designed with PAB.

We implemented and evaluated Stratus by integrating it with state-of-the-art BFT-based blockchain protocols and evaluated these protocols in both LAN and WAN settings. Our results show that Stratus-based protocols achieve up to $5-20\times$ more throughput than their native counterparts. In addition, the performance of Stratus degrades gracefully in the presence of network asynchrony, Byzantine attackers, and unbalanced workloads. Our design provides easy-to-use APIs so that other BFT systems suffering from leader bottlenecks can use Stratus.

PVLDB Reference Format:
Fangyu Gai[1,†,§], Jianyu Niu[2,†], Ivan Beschastnikh[3], Chen Feng[1], Sheng Wang[4]. Scaling Blockchain Consensus via a Robust Shared Mempool. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/gitferry/bamboo-stratus.

1 INTRODUCTION

The emergence of blockchain technology has revived interest in Byzantine fault-tolerant (BFT) data processing and database systems [4, 11, 26, 53, 64]. Unlike traditional distributed databases, BFT systems (or blockchains) provide data provenance and allow federated data processing in an untrusted and hostile environments [54, 58]. This enables a rich set of decentralized applications in various fields, e.g., finance [37], gaming [41], healthcare [31], and social media [1]. Many companies and researchers are seeking to build enterprise-grade blockchain systems [29, 42, 51, 59] to provide Internet-scale decentralized services [43].

The technical core of a blockchain system is the BFT consensus protocol, which allows distrusting parties to replicate and continuously order a sequence of transactions. Many BFT consensus protocols [10, 16, 30, 66] adopted by permissioned blockchains follow the classic leader-based design of PBFT [15]: only the leader node determines the order to avoid possible conflicts. We call such protocols leader-based BFT protocols, or LBFT.

In a 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 make sure that all the correct replicas commit the same proposal at the same position. If the leader is faulty or behaves in a detectable Byzantine manner, a view-change sub-protocol will be triggered to replace the malicious 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 more nodes increases the load on the leader and reduces performance. Empirical results show that in a LAN environment, as the network size grows (up to 64 replicas), the throughput of LBFT protocols drops from $120K$ tps (transaction per second) to $20K$ tps, while the transaction latency surges from 9 milliseconds to 3 seconds [27]. This has also been documented by other work [20, 34, 36].

Research in this direction has focused on increasing LBFT performance by improving the *commit phase*, e.g., reducing message complexity [66], truncating communication rounds [40], and enhancing tolerance to Byzantine faults [44, 63]. Recent work [20, 36] reveals that a more significant factor limiting LBFT's scalability lies in the *proposing phase*, in which a proposal with batched transaction data (e.g., $10MB$) is disseminated by the single leader node, whereas messages exchanged in the commit phase (e.g., signatures, hashes) are much smaller (e.g., $100B$). Our formal analysis in Appendix A.1 shows that reducing message complexity of the commit phase cannot address this scalability issue.

More broadly, previous work to address the leader bottleneck has proposed *horizontal scaling* or sharding to split the blockchain into shards that concurrently run consensus and validate transactions [3, 26, 39, 61]. These approaches requires a large network to ensure safety [21] and demand meticulous coordination for cross-shard transactions. By contrast, *vertical scaling* approaches

†These authors have contributed equally to this work. Corresponding author: JN.
§Work partly done during a research internship at Alibaba DAMO Academy.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

employ hierarchical schemes to send out messages and collect votes [17, 50]. Unfortunately, this increases latency and requires complex re-configuration to deal with faults.

In this paper, we follow neither of the above approaches. 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 changes to the consensus core.

Using SMP for scaling is not new [8, 20, 36], but there are two challenges that have not been properly addressed. One challenge is 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. This can happen because of network asynchrony (i.e., transactions are still in flight) and Byzantine attacks (i.e., transactions do not exist). These missing transactions prevent a consensus instance from entering the commit phase. The replica can fetch and wait for the missing transactions to arrive, or it can start a view-change (to replace the current leader). *Reliable broadcast* (RB) [9, 20] can resolve this issue by reliably disseminating transaction data, but RB has quadratic message overhead and multiple rounds of communication, which does not scale.

A second challenge is dealing with unbalanced load across replicas. SMP distributes the load of the leader by letting each replica disseminate transactions. However, real-world workloads are highly skewed [19]. This overwhelms popular replicas while leaving other replicas underutilized, resulting in low throughput and long tail latencies. Existing SMP protocols ignore this issue by assuming that each client sends transactions to a uniformly random replica [8, 20, 36], but this assumption does not hold in deployed blockchain systems [22, 23, 48, 62].

To address the above challenges, we propose Stratus, a robust SMP implementation that scales permissioned blockchains to hundreds of nodes. Stratus introduces a *provably available broadcast* (PAB) primitive to ensure the availability of transactions referenced in a proposal with negligible overhead. By using PAB, the leader generates a succinct proof for the availability of each transaction and piggybacks these proofs in proposal messages. This allows the consensus protocol to safely enter the commit phase of a proposal without waiting for the missing microblocks to be received. To deal with unbalanced workloads, Stratus uses a distributed load-balancing (DLB) protocol that is co-designed with PAB. DLB dynamically estimates a replica's local workloads and capacities so that overloaded replicas can forward their excess load to under-utilized replicas.

To summarize, this paper makes 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 the primitives and properties of this abstraction.
- To ensure the availability of transactions, we introduce an efficient broadcast primitive called PAB, which enables replicas to process a proposal without waiting for transaction data to arrive.
- To balance load across replicas, we utilize 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 state-of-the-art BFT consensus protocols, HotStuff [66] and Streamlet [16]. Our results show that Stratus-HotStuff/Streamlet substantially outperforms the native protocols in throughput, reaching up to 5× and 20× in typical LANs and WANs with 128 replicas. Compared with a straw-man SMP protocol, the throughput of Stratus degrades gracefully (by 20% to 30%) with 30% malicious replicas while the system throughput of its counterpart could drop to zero. Under unbalanced workloads, Stratus achieves up to 10× more throughput.

2 RELATED WORK

Table 1: Comparison of existing protocols addressing the leader bottleneck.

Protocol	**Approach**	**Availability guarantee**	**Load balance**	**Message complexity**
Tendermint [10]	Gossip	✓	✓	$O(n^2)$
Kauri [50]	Tree	✓	✗	$O(n)$
Leopard [36]	SMP	✗	✗	$O(n)$
Narwhal [20]	SMP	✓	✗	$O(n^2)$
Stratus	SMP	✓	✓	$O(n)$

One classic approach that can relieve the load on the leader is *horizontal scaling*, or sharding [3, 26, 39]. Sharding is widely used in distributed databases, which horizontally assigns participants into multiple shards, each of which is managed by a subset of the nodes and handles a share of workloads. However, using sharding in BFT consensus requires inter-shard and intra-shard consensus, which add another level of complexity to the system. An alternative, *vertical scaling* technique has been used in PigPaxos [17], 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 [29] can scale up to tens of thousands of replicas due to the usage of Verifiable Random Functions (VRFs) [46] and a novel Byzantine agreement protocol called BA★. For each consensus instance, a committee is a randomly selected via VRFs to reach consensus on the next set of transaction. Some protocols such as HoneyBadger [45] and Dumbo [33] adopt a leader-less design in which all the participants contribute to a proposal. Multi-leader BFT protocols [5, 34, 57] propose concurrent consensus in which multiple consensus instances are run concurrently, each led by a different leader. These proposals improve the scalability by circumventing the leader bottleneck rather than resolving it directly and they are orthogonal to our approach.

Several recent proposals address the leader bottleneck in BFT and we compare these in Table 1. Tendermint uses gossip to shed 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 [50], 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 reconfiguration strategy to overcome the disadvantages of using a tree structure. Similar to Kauri, Hermes [38] also use a tree structure. Tree-based approaches, however, significantly increase latency and require complex re-configuration strategies to deal with faults.

Leopard [36] and Narwhal [20] 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 same order of proposals being proposed. However, Leopard uses a basic implementation of SMP and does not guarantee that the referenced transactions in a proposal will be available. In addition, it does not scale well when the load across replicas is unbalanced. Narwhal [20] is a DAG-based Mempool protocol that is independent from the consensus protocol. It employs reliable broadcast (RB) [9] to reliably disseminate transactions and uses a DAG to establish 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 in separate machines.

3 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.

3.1 System Model

We consider two roles in the BFT protocol: leader and replica. A replica can become a leader via view-changes or leader-rotation. Therefore, we also call these leader replica and non-leader replica, respectively. We inherit the same Byzantine threat model and communication model from general BFT protocols [15, 66]. In particular, there are $N = 3f + 1$ replicas in the network and at most f replicas behave arbitrarily. The network is partially synchronous, whereby a known bound Δ on message transmission holds after some unknown Global Stabilization Time (GST) [25].

We consider external clients that issue transactions to the system and the identities of clients can be verified independently by every replica. We assume that each transaction has a unique id to avoid duplication. We also assume that every client knows the connection information of all the replicas (e.g., IP addresses) and sends its request to only one replica. This makes the received transactions of a replica disjoint from others and achieves the best possible performance since there is no request duplication among replicas. The selection of replicas can be based on network delay measurements (picking the closest replica) or using a random hash function. Byzantine replicas can censor transactions, however, so a client can always switch to another replica (using a timeout mechanism) until an honest replica is found. Finally, we assume all the messages sent in our system are cryptographically signed and authenticated and the adversary cannot break these signatures. Byzantine clients may perform *request duplication* attack by sending transactions to multiple replicas. Resolving this attack is discussed in Mir-BFT [57], which is orthogonal to our paper.

3.2 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 stores them to 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 [10, 28, 51], 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. See our analysis in Appendix A.1.

To relieve the leader's burden of distributing transaction data, we propose a *shared mempool* (SMP) abstraction, which has been used in previous work [20, 36, 52], 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 many applications, a transaction payload ranges from hundreds of bytes to several KBs, while the typical size of an id is tens of bytes (e.g., 32 bytes in SHA256). In addition, transaction data can be broadcast in batches, which further reduces the proposal size. In other words, SMP distributes the job of dissemination to the other replicas, so that the leader's workload is reduced. See our analysis in Appendix A.2. 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 properties are both liveness properties, which 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 [8], 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 3.5. Also note that using SMP does not hurt the safety of the consensus protocol.

3.3 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).* This primitive is used to receive an incoming tx from a client or replica, and stores it in memory (or disk if necessary).

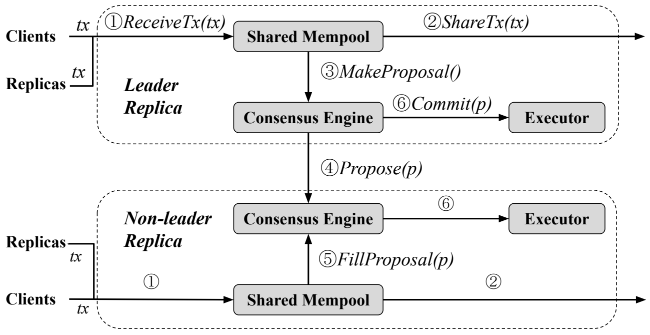


Figure 1: The processing of transactions in state machine replication using SMP.

- *ShareTx(tx)*. This primitive is used to distribute tx to other replicas.
- *MakeProposal()*. This primitive is used by the leader to pull transactions from the local mempool and construct a proposal with their ids.
- *FillProposal(p)*. This primitive 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 [4], where transactions are first executed and endorsed by several nodes, and then ordered by consensus, followed by serial validation and commit [55].

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 into 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).
- ③ 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.

3.4 Data Structure

In Stratus, transactions are constructed using three data structures: *microblock*, *block*, and *proposal*.

Microblock. Transactions are collected from clients and batched in microblocks for dissemination. This is to amortize the cost of verification. The total bytes in a microblock is no more than `BatchBytes` and a microblock is padded if not full. To avoid confusion, we use microblocks and transactions interchangeably throughout the paper. For example, the *ShareTx(tx)* primitive broadcasts a microblock instead of a single transaction in practice. 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 which consists of an id list of the microblocks and some metadata (e.g., 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*. Otherwise, we call it a *partial block*. A block contains all the data included in the relevant proposal and a list of microblocks.

3.5 Challenges and Solutions

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

Problem-I: missing transactions lead to a new bottleneck. Simply using best-effort broadcast [12] 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 [36] or network asynchrony. Figure 2 illustrates an example of this. In this Figure, a Byzantine broadcaster (R_5) only shares a transaction (tx_1) with the leader (R_1), not the other replicas. Therefore, when $R1$ 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. On the other hand, to ensure **SMP-Stability** of SMP, replicas have to proactively fetch missing transactions from the leader. This, however, creates a new bottleneck. Note that network asynchrony also causes missing transactions. And, it is difficult for the leader to distinguish between transaction requests caused by network asynchrony or those caused by malicious behavior.

A natural solution to address the above challenge is to use reliable broadcast (RB) [20] to implement *ShareTx(tx)*. The totality property of reliable broadcast ensures the **SMP-Stability** requirement of SMP in the sense that if some transaction is delivered by a correct leader, every correct replica eventually delivers the transaction. However, Byzantine reliable broadcast has quadratic message complexity and needs three communication rounds (round trip delay) [12], which is not suitable to large-scale systems. Note 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).

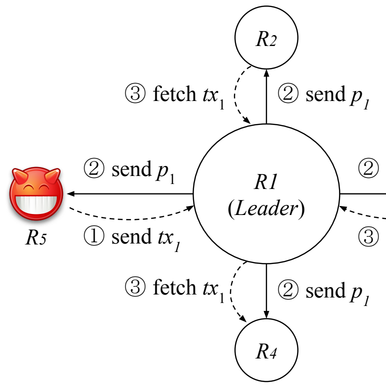


Figure 2: In a system with SMP, consisting of 5 replicas in which R_5 is Byzantine and R_1 is the current leader. ① R_5 only sends tx_1 to the leader (R_1). ② When R_1 broadcasts a proposal p_1 that includes the id of tx_1, the receiving replicas' Mempool does not contain tx_1. As such, when tx_1 is missing in p_1, ③ replicas have to fetch it from R_1 to ensure SMP-Stability, which, leads back to the leader bottleneck.

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 *available proof* of a sending message as long as at least one honest replica (excluding the sender) is holding the message. This proof ensures that the associated message can be eventually fetched from an honest replica. As such, by using PAB in Stratus, if a proposal contains valid available proofs for each referenced transactions, it can be passed directly to the commit phase without waiting for the transaction contents to arrive. Missing transactions can be fetched using background bandwidth without blocking the consensus.

Problem-II: unbalanced workload/bandwidth distribution. In deploying a BFT system across different datacenters, it is difficult to ensure that all the nodes have identical resources like bandwidth. 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 they tend to communicate with a preferred replica (nearest or most trusted). In these cases, replicas with lower ratio of workload to bandwidth become bottlenecks.

To address the heterogeneity in workload/bandwidth, a popular approach is to have replicas cooperatively disseminate transactions. For example, organizing replicas into a tree topology [39, 50], with the broadcaster placed at the root, helps to offload bandwidth usage to other nodes and deal with uneven node resources. Unfortunately, such schemes increase the delay since a message takes multiple hops to reach the destination. In addition, structured hierarchies require reconfiguration in the presence of faults, which is not straightforward in a Byzantine environment. An alternative approach is to use gossip [7, 14, 29], in which the broadcaster randomly pick a certain number of its peers and send 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 is responsible to disseminate a certain microblock on behalf the original sender and the proxy proves that a microblock is successfully distributed by submitting an available proof to the sender. If the proof is not submitted in time, the sender picks another under-utilized replica and repeats the process.

4 TRANSACTION DISSEMINATION

In this section, we introduce a new broadcast primitive called *provably available broadcast* (PAB) for transaction dissemination. PAB mitigates the impact of missing transactions (**Problem-I**). By using PAB, the leader generates a succinct proof for the availability of each transaction and piggybacks these proofs in proposal messages. As a result, replicas do not need to wait for missing transactions as long as the associated proofs are valid. Missing transactions can be fetched using background bandwidth without blocking consensus.

4.1 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 σ (realized via threshold signature [13]) over m, showing that m is available by at least one honest replicas, say r. Eventually, other replicas that do not receive m from s retrieves m from r. We say a replica delivers a message m if it receives m for the first time (either from clients or other replicas) and a $\langle \texttt{PAB-Ack} | m.id \rangle$ event is triggered. We use angle brackets to denote message and events. For messages, we assume they are signed by their senders. 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 replica r receives a valid proof σ 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. In the *push* phase, the *sender* broadcasts a message m and each receiver (including the sender) sends an PAB-Ack message $\langle \texttt{PAB-Ack} | m.id \rangle$ 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 σ (realized via threshold signature), showing that m has been delivered by at least one honest replica. The *recovery* phase begins right after σ is generated, and the *sender* broadcasts the proof message $\langle \texttt{PAB-Proof} | id, \sigma \rangle$. If some replica r receives a valid PAB-Proof without receiving m, r fetches m from other replicas in a repeated

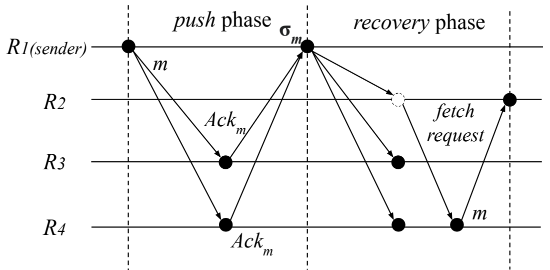


Figure 3: An example of the message flow in PAB with $N = 4$ replicas and $f = 1$. R_1 is the sender (Byzantine). R_2 did not receive m in the *push* phase due to the malicious $R1$ or network asynchrony. Thus, R_2 fetches m from R_4 (randomly picked) in the *recovery* phase.

Algorithm 1 PAB with message m at R_i (*push* phase)

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

4: upon event $\langle$PAB-Broadcast$|m\rangle$ do Broadcast($\langle$PAB-Msg$|m, R_i\rangle$)

5: upon receipt $\langle$PAB-Msg$|m, s\rangle$ for the first time do    ▷ $s \in C \cup R$
6:     Store($m$)                                        ▷ for future request
7:     trigger $\langle$PAB-Deliver$|m\rangle$
8:     if $s \in C$ then trigger $\langle$PAB-Broadcast$|m\rangle$
9:     else Send($s$, $\langle$PAB-Ack$|m.id, R_i\rangle$)

10: upon receipt $\langle$PAB-Ack$|id, s_j\rangle$ do              ▷ if $R_i$ is the sender
11:     $S \leftarrow S \cup s_j$
12:     if $|S| \geq q$ then                       ▷ satisfies the quorum condition
13:         $\sigma \leftarrow$ threshold-sign($S$)
14:         trigger $\langle$PAB-Ava$|id, \sigma\rangle$
```

manner. We should note that the quorum value q is adjustable between $f + 1$ and $2f + 1$, which is discussed in Section 4.3. We stick with $q = f + 1$ in the following descriptions.

Algorithm 1 shows the *push* phase of our implementation, 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 replica R_i triggers PAB-Broadcast only if m is received from clients C to avoid re-sharing (Line 8). In the second round, every replica that receives m acts as a witness by sending the sender an PAB-Ack message over $m.id$ (including the signature). If the sender receives at least q PAB-Ack messages of m from distinct replicas, it generates a proof σ from associated signatures via threshold-sign() and triggers a PAB-Ava event.

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 pseudo code of the *recovery* phase is presented in Algorithm 2. The sender broadcasts the proof σ of m on event PAB-Ava. After verifying σ, 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 σ (excluding replicas

Algorithm 2 PAB with message m at R_i (*recovery* phase)

```
1: Local Variables:
2: $signers \leftarrow \{\}$                                     ▷ signers of $m$
3: $requested \leftarrow \{\}$              ▷ replicas that have been requested

4: upon event $\langle$PAB-Ava$|id, \sigma\rangle$ do                    ▷ if $R_i$ is the sender
5:     Broadcast($\langle$PAB-Proof$|id, \sigma\rangle$)

6: upon receipt $\langle$PAB-Proof$|id, \sigma\rangle$ do
7:     if threshold-verify($id, \sigma$) is not true do return
8:     $signers \leftarrow \sigma.signers$
9:     if $m$ does not exist by checking $id$ do PAB-Fetch($id$)

10: procedure PAB-Fetch($id$)
11:     starttimer(Fetch, $\delta$, $id$)
12:     forall $r \in signers \setminus requested$ do
13:         if random([0, 1]) $> \alpha$ then
14:             $requested \leftarrow requested \cup r$
15:             Send($r$, $\langle$PAB-Request$|id, R_i\rangle$)
16:     wait until all requested messages are delivered, or $\delta$ timeout do
17:         if $\delta$ timeout do PAB-Fetch($id$)
```

that have been requested). The function random([0,1]) used by the algorithm returns a random real number ranging from 0 to 1. The configurable parameter α denotes the probability that a replica is requested. If the message is not fetched in δ time, the PAB-Fetch() procedure will be invoked again and the timer will be reset.

4.2 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 (by triggering $\langle$PAB-Broadcast$|mb\rangle$), 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 underlying 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 σ 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

Algorithm 3 *Propose* phase of view v at replica R_i

```
Local Variables:
mbMap ← {}                          ▷ maps microblock id to microblock
pMap ← {}                           ▷ maps microblock id to available proof
avaQue ← {}                         ▷ stores microblock id that is provably available

upon receipt ⟨PAB-Proof|id, σ⟩ do
    if threshold-verify(id, σ) is not true do return
    pMap[id] ← σ
    avaQue.Push(id)

upon event ⟨PAB-Deliver|mb⟩ do mbMap[mb.id] ← mb

upon event ⟨CE-NewView|v⟩ do
    if $R_i$ is the leader for view v then
        p ← MakeProposal(v)
        Broadcast(⟨CE-Propose|p, $R_i$⟩)

procedure MakeProposal(v)
    payload ← {}
    while(1)
        id ← avaQue.Pop()
        payload[id] ← pMap[id]
        if Len(payload) ≥ BlockSize or id =⊥ then
            break
    return newProposal(v, payload)

upon receipt ⟨CE-Propose|p, r⟩                    ▷ r is the current leader
    for id, σ ∈ p.payload do
        if threshold-verify(id, σ) is not true do
            trigger ⟨CE-ViewChange|$R_j$⟩
            return
    trigger ⟨CE-EnterCommit|p⟩
    FillProposal(p)

procedure FillProposal(p)
    block ← {p}
    forall id ∈ p.payload do
        if mb associated with id has not been delivered then
            PAB-Fetch(id)
    wait until every requested mb is delivered then
        forall id ∈ p.payload do
            block.Append(mbMap[id])
            avaQue.Remove(id)
            trigger ⟨CE-Full|block⟩
```

include all the associated available proofs of each referenced transaction is to show that the availability of each referenced microblock is guaranteed.

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 ⟨CE-EnterCommit|p⟩ 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. After a *full block* is constructed, the replica triggers a ⟨CE-Full|*block*⟩ event, indicating that the block is ready for execution.

As it is possible that a microblock id is committed before the corresponding content is received, the contained transactions cannot be immediately executed. Instead, these transactions (ids) are recorded and executed once the corresponding content is available. **The advantage of using PAB is its *provable availability*, which allows the consensus protocol to safely enter the commit phase of a proposal without waiting for the missing microblocks to be received.** In addition, the *recovery* phase proceeds concurrently with the consensus protocol (only background bandwidth is used) until the associated block is full for execution. Many optimizations [24, 56, 58] for improving the execution have been proposed and we hope to build on them in our future work.

4.3 Discussion

Adjustable quorums. Although we use $q = f + 1$ as the stability parameter in previous description, the threshold is adjustable between $f + 1$ and $2f + 1$ without hurting PAB's properties. The upper bound is $2f + 1$ because there are $N = 3f + 1$ 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.

Garbage collection. To ensure that transactions remain available in the system, replicas may have to keep the microblocks and relevant meta-data (e.g., acks) in case other replicas fetch them. To garbage-collect these messages, the consensus protocol should inform Stratus that a proposal is committed and the contained microblocks can then be garbage collected.

4.4 Correctness Analysis

Now we prove the correctness of PAB. Since the integrity and validity properties are simple to prove, here we only show that Algorithm 1 and Algorithm 2 satisfy **PAB-Provable Avalability**. Then we provide proofs that Stratus satisfies **SMP-Inclusion** and **SMP-Stability**.

Lemma 1 (**PAB-Provable Availability**). *If a proof σ over a message m is valid, then at least one correct replica holds m. In the recovery phase (Algorithm 2), the receiving replica r repeatedly invokes PAB-Fetch(id) and sends requests to randomly picked replicas. Eventually, a correct replica will respond and r will deliver m.*

Theorem 1. *Stratus ensures **PAB-Inclusion**.*

Proof. If a transaction tx is delivered and verified by a correct replica r (the sender), it will be eventually batched into a microblock mb and disseminated by PAB. Due to the *validity* property of PAB, mb will be eventually delivered by every correct replica, which sends acks over mb back to the sender. An available proof σ over mb will be generated and broadcast by the sender. Upon the receipt of σ, every correct replica pushes mb ($mb.id$) into $avaQue$. Therefore, mb (tx) will be eventually popped from $avaQue$ of a correct leader l and proposed by l. □

Theorem 2. *Stratus ensures **SMP-Stability**.*

Proof. If a transaction tx is included in a proposal by a correct leader, it means that tx is provably available (a valid proof σ over tx is valid). Due to the **PAB-Provable Availability** property of PAB, every correct replica eventually delivers tx. □

5 LOAD BALANCING

In this section, we discuss Stratus' load balancing strategy. 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 continually monitors the replica to determine if the replica is *busy* or *unbusy*. We discuss work estimation in Section 5.2. 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 replicas forward excess load.

5.1 Load Forwarding

Before forwarding excess load, replicas need to know which replicas have available resources. A naïve approach is to ask other replicas for their load status. However, this requires all-to-all communications and is not scalable. Therefore, we use the well-known Power-of-d-choices (Pod) algorithm [49, 60, 67] to avoid unnecessary message exchanges. In particular, a *busy* replica randomly samples workload information 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 7.4). The randomness in Pod ensures that the same proxy is unlikely to be re-sampled and overloaded.

Algorithm 4 depicts the implementation of the LB-ForwardLoad procedure. Upon the generation of a new microblock *mb*, the replica first checks whether it is *busy* (see Section 5.2). 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 workload information from d random replicas (excluding itself) within a timeout of τ (Line 10). Upon receiving a workload query, a replica obtains its current workload information by calling the GetLoadStatus() (see Section 5.2) and piggybacks it on the reply. 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 an PAB instance by itself (Line 13).

Algorithm 4 The Load Forwarding procedure at replica R_i

```
1:  Local Variables:
2:  $samples \leftarrow \{\}\{\}$                          ▷ stores sampled info for a microblock
3:  $banList \leftarrow \{\}$                              ▷ stores potentially Byzantine proxies

4:  upon event ⟨NewMB|$mb$⟩ do
5:      if IsBusy() do LB-ForwardLoad($mb$)
6:      else trigger ⟨PAB-Broadcast|$mb$⟩

7:  procedure LB-ForwardLoad($mb$)                    ▷ if $R_i$ is the busy sender
8:      starttimer(Sample, $\tau$, $mb.id$)
9:      $K \leftarrow$ SampleTargets($d$) $\setminus$ $banList$
10:     forall $r \in K$ do Send($r$, ⟨LB-Query|$md.id, R_i$⟩)
11:     wait until $|samples[md.id]| = d$ or $\tau$ timeout do
12:         if $|samples[md.id]| = 0$ then
13:             trigger ⟨PAB-Broadcast|$mb$⟩
14:             return
15:         find $r_p \in samples[md.id]$ with the smallest $w$
16:         starttimer(Forward, $\tau'$, $md$)
17:         $banList$.Append($r_p$)
18:         Send($r_p$, ⟨LB-Forward|$mb, R_i$⟩)            ▷ send $mb$ to the poxy
19:         wait until PAB-Proof over $mb$ is received or $\tau'$ timeout do
20:             if $\tau'$ do LB-ForwardLoad($mb$)

21: upon receipt ⟨LB-Forward|$mb, r$⟩ do           ▷ if $R_i$ is the proxy with $mb$
22:     trigger ⟨PAB-Broadcast|$mb$⟩

23: upon receipt ⟨LB-Query|$id, r$⟩ do                       ▷ if $R_i$ is sampled
24:     $w \leftarrow$ GetLoadStatus()
25:     Send($r$, ⟨LB-Info|$w, id, R_i$⟩)

26: upon receipt ⟨LB-Info|$w, id, r$⟩ do                        ▷ if $R_i$ is busy
27:     $samples[id][R_i] \leftarrow w$

28: upon receipt ⟨PAB-Proof|$id, \sigma$⟩ before $\tau'$ timeout do
29:     if threshold-verify($id, \sigma$) is not true do return
30:     trigger ⟨PAB-Ava|$id, \sigma$⟩        ▷ $R_i$ takes over the recovery phase
```

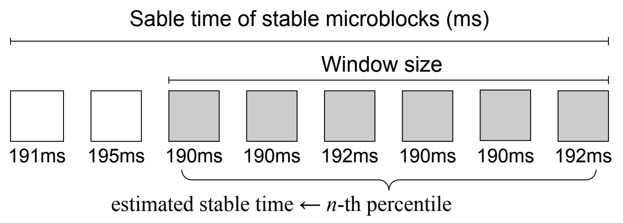
Handling faults. A sampled Byzantine replica can pretend to be unoccupied by responding with a low busy level and censor the forwarded microblocks. In this case, the **SMP-Inclusion** would be compromised: the transactions included in the censored microblock will not be proposed. We address this issue 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 σ over *mb* after this timeout, r re-transmits *mb* by re-invoking the LB-ForwardLoad(mb) (Line 20). Here the unique microblock ids prevent duplication. The above procedure repeats until a valid σ over *mb* is received. Then r continues the *recovery* phase of the PAB instance with *mb* by triggering the PAB-Ava event (Line 30).

To prevent Byzantine replicas from being sampled again, we use a *banList* to store proxies that have not finished the *push* phase of a PAB instance. That is, before a busy sender sends a microblock *mb* to a proxy, the proxy is added to the *banList* until the sender receives a valid proof message over *mb* before timeout. When sampling, the replicas in *banList* are excluded. Note that more advanced *banList* mechanism can be used based on proxies' behavior [18].

5.2 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, IsBusy() and GetLoadStatus(), respectively. To evaluate a replica's load status, two ingredients are necessary: workload and capacity. Although workload (e.g., disseminating microblocks) is easy to estimate, available resources (e.g., CPU and bandwidth) for the workload are difficult to estimate. 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. Figure 4 shows the estimation process. 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.



Sable time of stable microblocks (ms)
Window size
191ms
195ms
190ms
190ms
192ms
190ms
190ms
192ms
estimated stable time ← n-th percentile

Figure 4: The stable time (ST) of a replica is estimated by taking the n-th percentile of ST values over a window of latest stable microblocks. The window slides when new microblocks become stable.

Our approach is based on two observations. First, the variability in network delay in a private network is small. This was observed in [65] and we validated this in our own experiments (Appendix B). Second, network delay increases sharply when a node is overloaded. Thus, under a constant workload, the calculated ST should be at around a constant number α with error of ϵ. If the estimated ST is larger than $\alpha + \epsilon$ by a parameter of β, 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 are available at the replica for disseminating microblocks. Therefore, we directly 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 different 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 loads to these replicas also benefits our system. For overloaded replicas with large ST values, the topology has a negligible impact.

6 IMPLEMENTATION

We have implemented a prototype of Stratus[1] in the Go language using Bamboo [27][2], which is an open source project for prototyping, evaluating, and benchmarking BFT consensus and replication protocols. Bamboo readily provides validated implementations of state-of-the-art BFT protocols such as two-chain HotStuff, HotStuff [66], and Streamlet [16]. Bamboo also supplies many common functionalities that a BFT replication protocol needs, including network communication (point-to-point reliable communication via TCP), Pacemaker (for view-changes) [66], state machine of a key-value store, client API and Mempool. In our implementation we replaced the original mempool in Bamboo with our newly designed shared mempool. Because of Stratus' well-designed interfaces, the consensus core is minimally modified. Note that similar to [32, 66], we use ECDSA to implement the quorum proofs of PAB instead of threshold signature. This is because the computation efficiency of ECDSA[3] is superior to that of Boldyreva's threshold signature [32].

Overall, our implementation added about 1,300 lines of Golang code to the Bamboo codebase.

Optimizations. Since the transmission of microblocks consumes the most communication resources, we need to reserve sufficient resources for consensus messages to ensure the progress of the system. To achieve this, we adopt two optimizations in our implementation. First, we prioritize the transmission and processing of consensus messages. In particular, we process consensus messages (i.e., proposals and votes) and other messages (mainly microblocks) in two separate channels, the consensus channel and the data channel, respectively. And we give the consensus channel a higher priority: whenever a consensus message arrives, it is processed first. Second, we use a token-based limiter to limit the sending rate of data messages. Particularly, every data message (i.e., microblock) needs a token before being sent out and tokens are refilled in a configurable time. This ensures that the network resources will not be overtaken by data messages.

7 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 7.2)
- **Q2:** how does missing transactions caused by network asynchrony and replicas' Byzantine behavior affect protocols' performance? (Section 7.3)
- **Q3:** how does unbalanced load affect protocols' throughput? (Section 7.4)

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

7.1 Setup

Testbeds. We conducted our experiments on Alibaba Cloud ecs.s6-c1m2.xlarge instances[4]. Each instance has 4vGPUs and 8GB memory and runs Ubuntu server 20.04. We ran each replica on a single ECS instance. We perform protocol evaluations in LANs and WANs to simulate *national* and *regional* deployments, respectively [50]. In LAN deployments, each replica has a bandwidth of $1Gb/s$ and inter-replica roundtrip time (RTT) of less than $10ms$. For WAN deployments, we use NetEm [35] to simulate a WAN environment with $100ms$ inter-replica RTT and $100Mb/s$ replica bandwidth.

Workload. Clients are run on 4 instances with the same specifications. 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 network delay between a replica and a client. Each data point is obtained when the measurement is stabilized and is an average over 3 runs. In our experiments, workloads are evenly distributed across replicas except for the last set of experiments (Section 7.4), in which we create skewed load to evaluate load balancing.

Table 2: Summary of evaluated protocols.

Acronym	Protocol description
N-HS	Native HotStuff [66] without a shared mempool
N-SL	Native Streamlet [16] without a shared mempool
SMP-HS	HotStuff integrated with a simple shared mempool
SMP-HS-G	SMP-HS with gossip instead of broadcast
Narwhal [20]	HotStuff based shared mempool
S-HS	HotStuff integrated with **Stratus (this paper)**
S-SL	Streamlet integrated with **Stratus (this paper)**

Protocols. We evaluate the performance of a wide range of protocols (Table 2). We use native HotStuff and Streamlet with the original mempool as the baseline, denoted as (N-HS and N-SL, respectively). Further, we replace the original mempool of HotStuff with a basic shared mempool with best-effort broadcast and fetching, denoted as (SMP-HS). Finally, we equip HotStuff and Streamlet with our Stratus Mempool, denoted as (S-HS and S-SL, respectively). We also implemented a gossip-based shared mempool (distribute microblocks via gossip), denoted by SMP-HS-G, to evaluate load balancing and compare it with S-HS. All the above protocols are implemented using the same code base (i.e., Bamboo) for apple-to-apple comparison. The sampling parameter d is set to 1 by default unless we change it explicitly. In addition, we use Narwhal[5], which employs a shared mempool with reliable broadcast, for comparison. Narwhal is based on HotStuff and splits functionality between workers and primaries, which are responsible for txn dissemination and consensus, respectively. To fairly compare Narwhal with Stratus, we let each primary have one worker and locate both in one VM instance.

[4]https://www.alibabacloud.com/help/en/doc-detail/25378.htm
[5]Available at https://github.com/facebookresearch/narwhal/

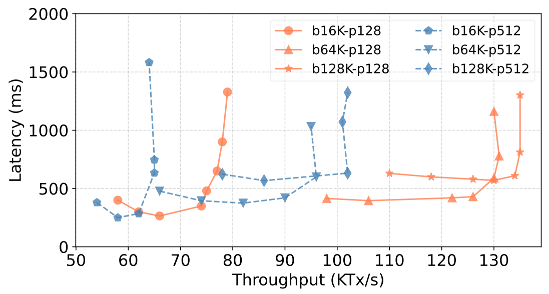


Figure 5: Throughput vs. latency with 128 replicas for S-HS. The batch size varies among $16KB$, $64KB$, and $128KB$. The transaction payload is either 128 or 512 bytes.

7.2 Scalability

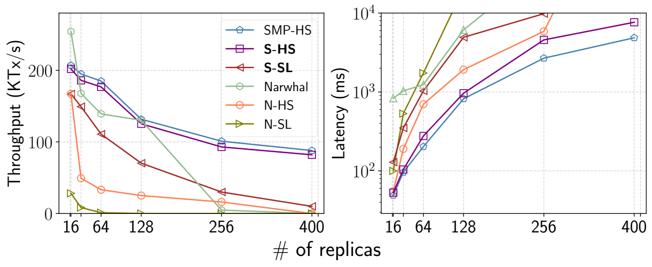
In the first set of experiments, we explore the impact of batch sizes and payload sizes on the proposed protocols. Then we vary the number of replicas from 16 to 400 to evaluate the scalability of the protocols in both LANs and WANs. These experiments are run in a common BFT setting in which less than one third of replicas remain silent. Since our focus is on normal-case performance, view changes are not triggered in these experiments unless clearly stated.

Throughput vs. latency. We deploy Stratus-based HotStuff (S-HS) in a LAN setting with 128 replicas. We vary the batch sizes from $16KB$ to $128KB$ and use the transaction payloads of 128 and 512 bytes (commonly used in blockchain systems [28, 51]). For instance, the batch size of $16KB$ with the transaction payload 128 bytes is denoted as $b16K - p128$. We gradually increase the workload until the protocol is saturated.

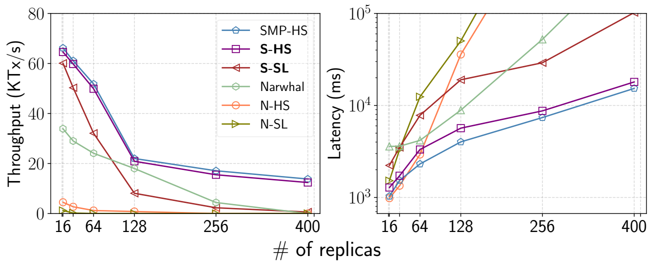
The results are depicted in Figure 5. We can first see that the dashed lines ($p128$) share a similar pattern to solid lines ($p512$). We can see that the throughput is higher when a larger batch size is used. This is because batching more transactions in a microblock amortizes the communication and verification cost. However, we observe that a larger batch size increases latency. This is because a replica must wait longer for transactions to fill a microblock. Further, the throughput gain of choosing a larger batch size is reduced as the batch size increases. We use the batch size of $128KB$ with the transaction payload 128 bytes in the rest of our experiments to balance throughput and latency.

Scalability. We evaluate the scalability of the protocols by increasing the number of replicas from 16 to 400. We use N-HS, N-SL, SMP-HS, S-SL, and Narwhal for comparison and run experiments in both LANs and WANs.

Figure 6 depicts throughput and latency of the protocols with an increasing number of replicas in LANs and WANs. We can



(a) LAN evaluation.



(b) WAN evaluation.

Figure 6: The throughput (left) and latency (right) of protocols in both LAN and WAN with increasing number of replicas. We use 128-byte payload and 128*KB* batch size.

see that protocols using the shared mempool (SMP-HS, S-HS, S-SL, and Narwhal) outperform the native HotStuff and Streamlet (N-HS and N-SL) in throughput in all experiments. Previous work [20, 36, 66] has 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 [20], Narwhal achieves better scalability only when each primary has multiple workers that are located in different machines.

SMP-HS and S-HS show a slightly larger 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 larger than 64 and show flatter lines in throughput as the network size increases. The throughput of SMP-HS and S-HS achieve 5× 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.

Bandwidth consumption. We further evaluate the outward bandwidth usage at the leader and the non-leader replica in N-HS, SMP-HS, and S-HS. We present the results in Table 3. 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, in which the leader bottleneck is significantly alleviated. We also observe that S-HS introduces around 10% overhead compared with SMP-HS. In the next section, we show this overhead is worthwhile to trade for availability insurance.

Table 3: Outward bandwidth consumption comparison (Mbps) with $N = 64$ replicas. The bandwidth of each replica is throttled to 100Mbps. The results are collected when the network is saturated.

Role/Messages		N-HS	SMP-HS	**S-HS (this paper)**
Leader	Proposals	75.4	4.7	9.8
	Microblocks	N/A	50.5	50.3
	SUM	**75.4**	**55.2**	**60.1**
Non-leader	Microblocks	N/A	50.4	50.3
	Votes	0.5	2.5	2.4
	Acks	N/A	N/A	4.7
	SUM	**0.5**	**52.9**	**57.4**

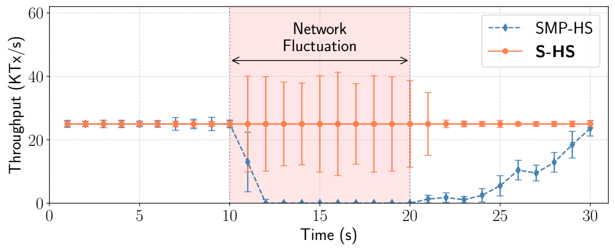


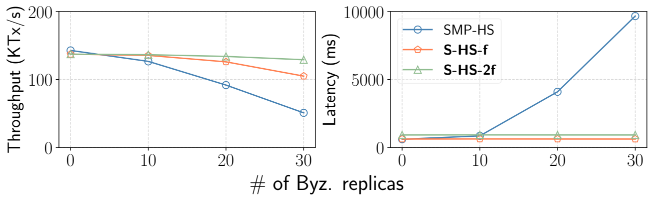
Figure 7: Extra network delay is injected at time 10s and lasts for 10s. The transaction rate is 25KTx/s. Results are averaged from 10 runs.

7.3 Impact of Missing Transactions

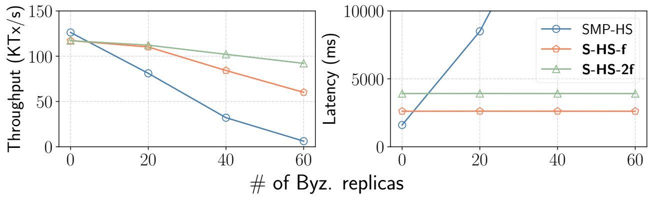
Recall that in **Problem-I** (Section 3.5), a basic shared mempool with best-effort broadcast is subject to network asynchrony and Byzantine senders that can cause missing transactions. In the next set of experiments, we evaluate the throughput of SMP-HS and S-HS in above situations.

Network asynchrony. Network asynchrony will cause transaction to be missing (Section 3.5) and negatively impact performance. We run an experiment in a WAN setting, during which we induce a period of network fluctuation via NetEm. The fluctuation lasts for 10s, during which network delays between replicas fluctuate between 100ms and 300ms for each messages (i.e., 200ms base with 100ms uniform jitter). We set the view-change timer to be 1,000ms. We keep the transaction rate at 25KTx/s without saturating the network.

We run the experiment 10 times and each run lasts 30s. The averaged throughput and variance of SMP-HS and S-HS over time are shown in Figure 7. 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.



(a) 100 total replicas with 0 to 30 Byz. ones.



(b) 200 total replicas with 0 to 60 Byz. ones.

Figure 8: Performance of SMP-HS and S-HS with different quorum parameters (S-HS-d1 and S-HS-d2) and increasing Byzantine replicas.

Byzantine senders. The goal of the attacker in this scenario is to induce many missing microblocks to overwhelm the leader. Note that it is difficult to identify which replicas are malicious (the sending replicas or the replicas that fetch missing transactions).

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 4.3), $f+1$ 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 8 plots the results. We can see that as the number of Byzantine replicas increases, the throughput/latency of SMP-HS decrease/increases sharply. This is because replicas have to fetch missing microblocks from the leader before processing a proposal. We also observe a slight drop of throughput in 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 the Byzantine behavior has more impact on larger deployments. With $N = 200$ replicas, the performance of SMP-HS decreases significantly. The throughput is almost zero when the number of Byzantine replicas is 60 and the latency surges when Byzantine replicas are more than 20. Finally, S-HS-2f has better throughput than S-HS-f at a cost of higher latency as the number of Byzantine replicas increases. The reason is that a larger quorum size forces Byzantine replicas to send to more honest replicas while more ack messages need to be collected.

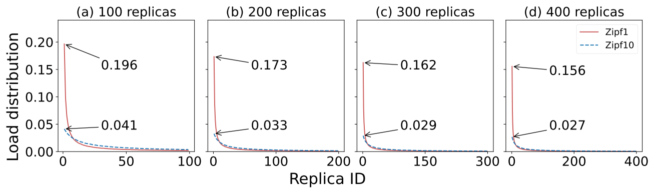


Figure 9: Workload distribution with different network sizes and Zipfian parameters.

7.4 Impact of Unbalanced Workload

Previous work has observed that node degrees in large-scale blockchain networks have a power-law distribution [22, 23, 48, 62]. As a result, most clients send transactions to a few popular nodes, leading to unbalanced workload across replicas. In this experiment, we use two Zipfian parameters[6], 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 9 for varying network sizes.

To evaluate load-balancing in Stratus, we run experiments using the above workload distributions in a WAN setting. Since our load-balancing protocol 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 implemented a gossip-based broadcast protocol, denoted by SMP-HS-G for comparison. We set the fanout parameter of gossip to 3.

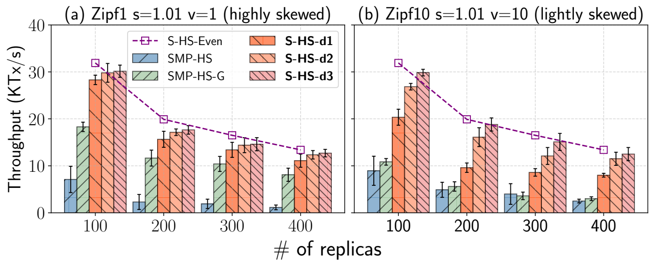


Figure 10: Throughput with different workload distribution.

Figure 10 shows the protocols' throughput. We can see that S-HS-dx outperforms SMP-HS and SMP-HS-G in all experiments. S-HS-dx achieves 5× ($N = 100$) to 10× ($N = 400$) 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 in highly skewed workloads.

8 CONCLUSION AND FUTURE WORK

We presented a shared mempool abstraction that resolves the leader bottleneck limitation of leader-based BFT protocols. We designed a novel shared mempool protocol called Stratus to address two challenges: missing transactions and unbalanced workload. Stratus overcomes these limitations by introducing an efficient provably

[6]Golang Zipfian generator. https://go.dev/src/math/rand/zipf.go.

available broadcast (PAB) and a load balancing protocol. Our experiments show that Stratus-HotStuff substantially outperforms native HotStuff, improving bandwidth by up to 5× and 20× in LAN and WAN environments with 128 replicas, respectively. With skewed workloads, Stratus-HotStuff achieves up to 10× more throughput than HotStuff with a basic shared mempool protocol. For our future work we plan to extend Stratus to multi-leader BFT protocols.

REFERENCES

[1] [n.d.]. Steemit. https://steemit.com/.

[2] Alibaba. [n.d.]. Alibaba cloud baas (blockchain-as-a-service). https://www.aliyun.com/product/baas,2018.

[3] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. SharPer: Sharding Permissioned Blockchains Over Network Clusters. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021.* ACM, 76–88.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018.* ACM, 30:1–30:15.

[5] Zeta Avarikioti, Lioba Heimbach, Roland Schmid, and Roger Wattenhofer. 2020. FnF-BFT: Exploring Performance Limits of BFT Protocols. *CoRR* abs/2009.02235 (2020).

[6] AWS. [n.d.]. Amazon quantum ledger database (qldb). https://aws.amazon.com/qldb,2018.

[7] Nicolae Berendea, Hugues Mercier, Emanuel Onica, and Etienne Rivière. 2020. Fair and Efficient Gossip in Hyperledger Fabric. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020, Singapore, November 29 - December 1, 2020.* IEEE, 190–200.

[8] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. 2012. S-Paxos: 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.* IEEE Computer Society, 111–120.

[9] Gabriel Bracha. 1987. Asynchronous Byzantine Agreement Protocols. *Inf. Comput.* 75, 2 (1987), 130–143.

[10] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. *CoRR* abs/1807.04938 (2018). arXiv:1807.04938 http://arxiv.org/abs/1807.04938

[11] Yehonatan Buchnik and Roy Friedman. 2020. FireLedger: A High Throughput Blockchain Consensus Protocol. *Proc. VLDB Endow.* 13, 9 (2020), 1525–1539.

[12] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media.

[13] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and Efficient Asynchronous Broadcast Protocols. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings (Lecture Notes in Computer Science)*, Joe Kilian (Ed.), Vol. 2139. Springer, 524–541.

[14] Daniel Cason, Enrique Fynn, Nenad Milosevic, Zarko Milosevic, Ethan Buchman, and Fernando Pedone. [n.d.]. The design, architecture and performance of the Tendermint Blockchain Network. In *40th International Symposium on Reliable Distributed Systems, SRDS 2021, Chicago, IL, USA, September 20-23, 2021.* IEEE, 23–33.

[15] Miguel Castro and Barbara Liskov. 1999. 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.* USENIX Association, 173–186.

[16] Benjamin Y. Chan and Elaine Shi. 2020. Streamlet: Textbook Streamlined Blockchains. In *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020.* ACM, 1–11.

[17] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2021. PigPaxos: Devouring the Communication Bottlenecks in Distributed Consensus. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021.* ACM, 235–247.

[18] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. [n.d.]. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proc. of USENIX NSDI 2009.*

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154.

[20] George Danezis, Eleftherios Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2021. Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus. *CoRR* abs/2105.11827 (2021). arXiv:2105.11827 https://arxiv.org/abs/2105.11827

[21] Bernardo David, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. 2021. GearBox: An Efficient UC Sharded Ledger Leveraging the Safety-Liveness Dichotomy. Cryptology ePrint Archive, Report 2021/211. https://ia.cr/2021/211.

[22] Christian Decker and Roger Wattenhofer. 2013. Information propagation in the Bitcoin network. In *13th IEEE International Conference on Peer-to-Peer Computing, IEEE P2P 2013, Trento, Italy, September 9-11, 2013, Proceedings.* IEEE, 1–10.

[23] Sergi Delgado-Segura, Surya Bakshi, Cristina Pérez-Solà, James Litton, Andrew Pachulski, Andrew Miller, and Bobby Bhattacharjee. 2019. TxProbe: Discovering Bitcoin's Network Topology Using Orphan Transactions. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers (Lecture Notes in Computer Science)*, Ian Goldberg and Tyler Moore (Eds.), Vol. 11598. Springer, 550–566.

[24] Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Adding Concurrency to Smart Contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, Elad Michael Schiller and Alexander A. Schwarzmann (Eds.). ACM, 303–312.

[25] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. 35, 2 (1988).

[26] Muhammad El-Hindi, Martin Heyden, Carsten Binnig, Ravi Ramamurthy, Arvind Arasu, and Donald Kossmann. 2019. BlockchainDB - Towards a Shared Database on Blockchains. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* ACM, 1905–1908.

[27] Fangyu Gai, Ali Farahbakhsh, Jianyu Niu, Chen Feng, Ivan Beschastnikh, and Hao Duan. 2021. Dissecting the Performance of Chained-BFT. In *41th IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Virtual.* IEEE, 190–200.

[28] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. [n.d.]. The Bitcoin Backbone Protocol: Analysis and Applications. In *Proc. of EUROCRYPT, 2015.*

[29] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017.* ACM, 51–68.

[30] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019.* IEEE, 568–580.

[31] Kristen N. Griggs, Olya Ossipova, Christopher P. Kohlios, Alessandro N. Baccarini, Emily A. Howson, and Thaier Hayajneh. 2018. Healthcare Blockchain System Using Smart Contracts for Secure Automated Remote Patient Monitoring. *J. Medical Syst.* 42, 7 (2018), 130:1–130:7.

[32] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Speeding Dumbo: Pushing Asynchronous BFT Closer to Practice. Cryptology ePrint Archive, Report 2022/027. https://ia.cr/2022/027.

[33] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2020. Dumbo: Faster Asynchronous BFT Protocols. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 803–818.

[34] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. 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.* IEEE, 1392–1403.

[35] Stephen Hemminger et al. 2005. Network emulation with NetEm. In *Linux conf au*, Vol. 5. Citeseer, 2005.

[36] Kexin Hu, Kaiwen Guo, Qiang Tang, Zhenfeng Zhang, Hao Cheng, and Zhiyang Zhao. 2021. Don't Count on One to Carry the Ball: Scaling BFT without Sacrifing Efficiency. *CoRR* abs/2106.08114 (2021). https://arxiv.org/abs/2106.08114

[37] IBM. [n.d.]. Blockchain for financial services. https://www.ibm.com/blockchain/industries/financial-services.

[38] Mohammad Mussadiq Jalalzai, Chen Feng, Costas Busch, Golden Richard III, and Jianyu Niu. 2021. The Hermes BFT for Blockchains. *IEEE Transactions on Dependable and Secure Computing* (2021), 1–1.

[39] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: 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.* IEEE Computer Society, 583–598.

[40] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. 2007. Zyzzyva: 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.* ACM, 45–58.

[41] Dapper Labs. [n.d.]. Crypto Kitties. https://www.cryptokitties.co/.

[42] Dapper Labs. [n.d.]. Flow Blockchain. https://www.onflow.org/.

[43] Mingyu Li, Jinhao Zhu, Tianxu Zhang, Cheng Tan, Yubin Xia, Sebastian Angel, and Haibo Chen. 2021. Bringing Decentralized Search to Decentralized Services. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021.* USENIX Association, 331–347.

[44] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. 2019. Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing. *IEEE Trans. Computers* 68, 1 (2019), 139–151.

[45] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew K. Miller. 2019. HoneyBadgerMPC and AsynchroMix: Practical Asynchronous MPC and its Application to Anonymous Communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019.* ACM, 887–903.

[46] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. 1999. Verifiable Random Functions. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA.* IEEE Computer Society, 120–130.

[47] Microsoft. [n.d.]. Microsoft azure blockchain service. https://azure.microsoft.com/services/blockchain-service,2018.

[48] Andrew K. Miller, James Litton, Andrew Pachulski, Neal Gupta, Dave Levin, Neil Spring, and Bobby Bhattacharjee. 2015. Discovering Bitcoin ' s Public Topology and Influential Nodes.

[49] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

[50] Ray Neiheiser, Miguel Matos, and Luís E. T. Rodrigues. 2021. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021.* ACM, 35–48.

[51] Novi. [n.d.]. DiemBFT. https://www.novi.com/.

[52] A. Pinar Ozisik, Gavin Andresen, Brian Neil Levine, Darren Tapp, George Bissias, and Sunny Katkuri. [n.d.]. Graphene: efficient interactive set reconciliation applied to blockchain propagation. In *Proc. of ACM SIGCOMM 2019.*

[53] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. 2020. 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.* ACM, 637–652.

[54] Pingcheng Ruan, Tien Tuan Anh Dinh, Dumitrel Loghin, Meihui Zhang, Gang Chen, Qian Lin, and Beng Chin Ooi. 2021. Blockchains vs. Distributed Databases: Dichotomy and Fusion. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021.* ACM, 1504–1517.

[55] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2020. A Transactional Perspective on Execute-order-validate Blockchains. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020.* ACM, 543–557.

[56] Alexander Spiegelman, Arik Rinberg, and Dahlia Malkhi. 2020. ACE: Abstract Consensus Encapsulation for Liveness Boosting of State Machine Replication. In *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference) (LIPIcs)*, Vol. 184. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:18.

[57] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2019. Mir-BFT: High-Throughput BFT for Blockchains. *CoRR* abs/1906.05552 (2019). arXiv:1906.05552 http://arxiv.org/abs/1906.05552

[58] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (transactions). In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021.* ACM, 1–17.

[59] Tendermint. [n.d.]. Tenderment Core. https://tendermint.com/.

[60] Nikita Dmitrievna Vvedenskaya, Roland L'vovich Dobrushin, and Fridrikh Izrailevich Karpelevich. 1996. Queueing system with selection of the shortest of two queues: An asymptotic approach. *Problemy Peredachi Informatsii* 32 (1996), 20–34.

[61] Jiaping Wang and Hao Wang. 2019. Monoxide: Scale out Blockchains with Asynchronous Consensus Zones. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 95–112.

[62] Taotao Wang, Chonghe Zhao, Qing Yang, Shengli Zhang, and Soung Chang Liew. 2021. Ethna: Analyzing the Underlying Peer-to-Peer Network of Ethereum Blockchain. *IEEE Trans. Netw. Sci. Eng.* 8, 3 (2021), 2131–2146.

[63] Zhuolun Xiang, Dahlia Malkhi, Kartik Nayak, and Ling Ren. 2021. Strengthened Fault Tolerance in Byzantine Fault Tolerant Replication. *CoRR* abs/2101.03715 (2021). arXiv:2101.03715 https://arxiv.org/abs/2101.03715

[64] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. 2021. SlimChain: Scaling Blockchain Transactions through Off-Chain Storage and Parallel Processing. *Proc. VLDB Endow.* 14, 11 (2021).

[65] Xinan Yan, Linguan Yang, and Bernard Wong. 2020. Domino: using network measurements to reduce state machine replication latency in WANs. In *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020.* ACM, 351–363.

[66] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. [n.d.]. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proc. of ACM PODC, 2019.*

[67] Lei Ying, R. Srikant, and Xiaohan Kang. 2015. The power of slightly more than one sample in randomized load balancing. In *2015 IEEE Conference on Computer Communications (INFOCOM).* 1131–1139.

A ANALYSIS

In this section, we first theoretically reveal the leader bottleneck of leader-based BFT protocols and then show how Stratus address the issue. In particular, we consider the ideal performance, i.e., all replicas are honest and the network is synchronous. We further assume that the ideal performance is limited by the available network capacity of each replica, denoted by C. For simplicity, we assume that transactions have the same size B (in bit unit). We use T_{max} to denote the maximum throughput, i.e., the number of proceeded transactions per second. We use W_l (resp. W_{nl}) to denote the workload of the leader (resp. a non-leader replica) for confirming a transaction. Furthermore, we have that

$$T_{max} = \min\left\{\frac{C}{W_l}, \frac{C}{W_{nl}}\right\}.$$

Since each replica has to receive and process the transaction once, we have $W_l, W_{nl} \geq B$. As a result, $T_{max} \leq C/B$. In other words, C/B is the upper bound of the maximum throughput of any BFT protocols.

A.1 Bottleneck of LBFT Protocols

In LBFT protocols, when making consensus of a transaction, the leader is in charge of disseminating it to other $n-1$ replicas, while each non-leader replica proceeds it from the leader. Hence, the workloads of proceeding the transaction for the leader and a non-leader replica are $W_l = B(n-1)$ and $W_{nl} = B$, respectively. Furthermore, we have

$$T_{max} = \min\left\{\frac{C}{B(n-1)}, \frac{C}{B}\right\} = \frac{C}{B(n-1)}.$$

The equation shows that with the increase of replicas, the maximum throughput of LBFT protocols will drops proportionally. Note that protocol overhead is not considered, which makes it easier to illustrate the unbalanced loads between the leader and non-leader replicas and to show the leader bottleneck.

Next, we take the most influential LBFT protocol, PBFT [15], as a concrete example to show more details of the leader bottleneck. As said in Section ??, in PBFT the agreement of a transaction involves three phases: the pre-prepare, prepare and commit phases. In particular, the leader first receives a transaction from a client, and then disseminates the transaction to all other $n-1$ replicas in pre-prepare phase. In prepare and commit phases, each replica involves in an broadcasting of their vote messages and receives all others' vote messages for reaching consensus.[7] Let σ denote the size of voting messages. The total workloads for the leader and a non-leader replica are $W_l = nB + 4(n-1)\sigma$ and $W_{nl} = B + 4(n-1)\sigma$,

[7]In implementation, the leader does not need to broadcast its votes in the prepare phase since the proposed transaction could represent the vote message.

respectively. Finally, we can derive the maximum throughput of PBFT as

$$T_{max} = \min\left\{\frac{C}{nB + 4(n-1)\sigma}, \frac{C}{B + 4(n-1)\sigma}\right\}. \quad (1)$$

The equations shows that both the dissemination of the transaction and vote messages limit the throughout.

Batching Processing. By Eq. (1), we can see that when processing a transaction in PBFT, each replica has to process $4(n-1)$ vote messages, which leads to high protocol overhead. To address this, multiple transactions can be batch into to a proposal (e.g., forming a block) to amortize the protocol overhead. For example, let K denote the size of a proposal, and the maximum throughput of PBFT when adopting batch strategy is

$$T_{max} = \frac{K}{B} \times \min\left\{\frac{C}{nK + 4(n-1)\sigma}, \frac{C}{K + 4(n-1)\sigma}\right\}.$$

When K is large (i.e., $K \gg \sigma$), we have $\frac{C}{nK+4(n-1)\sigma} \approx \frac{C}{nK}$ and $T_{max} = \frac{C}{nB}$. The shows that the maximum throughout drops with the increasing number of replicas, and the disseminating the proposal by the leader is still the bottleneck. In other words, batching strategy cannot address scalability issues of LBFT protocols. What is more, several state-of-the-art LBFT protocols such as HotStuff [66] achieve the linear message complexity by removing the $(n-1)$ factor from the $(n-1)\sigma$ overhead of non-leader replicas. However, this still cannot address the scalability issue since the proposal dissemination for the leader is still the dominating component.

A.2 Analysis of Using Shared Mempool

To address the leader bottleneck of LBFT protocols, our solution is to decouple the transaction dissemination with consensus algorithm, by which dissemination workloads can be balanced among all replicas, leading to better utilization of replicas' network capacities. In particular, to improve the efficiency of dissemination, transactions can be batched into microblocks, and replicas disseminates microblocks to each other. Each microblock is accompanied with a unique identifier. Later, after a microblock is synchronized among replicas, the leader only needs to propose an identifier of the microblock to make agreement. Since the unique mapping between identifiers and microblocks, ordered identifiers leads to a sequence of microblocks, which further determines a sequence of transactions.

Next, we show how the above decoupling idea can address the leader bottleneck. We use γ to denote the size of an identifier and η to denote the size of a microblock. Given a proposal with the same size K, it can include K/γ identifiers. Each identifier represents an microblock with η/B transactions. Hence, a proposal represents $\frac{K}{\gamma} \times \frac{\eta}{B}$ transactions. As said previously, the K/γ microblocks are disseminated by all non-leader replicas, so each non-leader replica has to disseminate $K/(\gamma(n-1))$ microblocks to all other replicas. Correspondingly, each replica (including the leader) can receive $K/(\gamma(n-1))$ microblocks from $n-1$ non-leader replicas. Hence, the workload for the leader is

$$W_l = (n-1)\frac{K\eta}{\gamma(n-1)} + (n-1)K = \frac{K\eta}{\gamma} + (n-1)K,$$

where $(n-1)K$ is the workload for disseminating the proposal. Similarly, the workloads for a non-leader replica is

$$W_l = n\frac{K\eta}{\gamma(n-1)} + (n-2)\frac{K\eta}{\gamma(n-1)} + K = \frac{2K\eta}{\gamma} + K,$$

where K is the workload for receiving a proposal from the leader. Finally, we can derive the maximum throughput as

$$T_{max} = \frac{K\eta}{\gamma B} \times \min\left\{\frac{C}{(K\eta)/\gamma + (n-1)K}, \frac{C}{(2K\eta)/\gamma + K}\right\}.$$

To make the throughout maximum, we can adjust η and γ to balance the workloads of the leader and non-leader replicas. This is $\frac{2K\eta}{\gamma} + K = \frac{K\eta}{\gamma} + (n-1)K$, and we have $\eta = (n-2)\gamma$. Finally, we can obtain the maximum throughput is $T_{max} = \frac{C(n-2)}{B(2n-3)}$. Particularly, when n is large, we have $T_{max} \approx \frac{C}{2B}$. The result is optimal since given a transaction, it has to be sent and received for n times (one for each replica), which leads to about $2nB$ workload, and the total network capacities of all replicas is nC. What is more, the maximum throughput $\frac{C}{2B}$ is not affected by the number of replicas, which makes the solution scalable.

Remark 1. *The above analysis focuses on replicas' network capacity, especially on the dissemination bottleneck of the leader. In reality, system performances are also limited by many factors such as replicas' storage and processing capacities. The analysis of these bottlenecks are orthogonal to this work and will be addressed in the future work.*

B INTER-DATACENTER DELAYS

We measured the network delays across 4 datacenters of Alibaba Cloud located in different regions, Singapore (SG), Sydney (SN), Virginia (VG), and London (LD). We use public IPs since private IPs are not applicable across regions in Alibaba Cloud. Each instance runs a client and a server [8]. A client issues a probing message every 10ms and records the sending and responding time to calculate the round-trip between a pair of datacenters.

The measurements lasted 24 hours and we choose the VG-SG pair (from VG to SG) as an example to exhibit the results. Figure **??** shows a heat map of the measured delays during 24 hours, and Figure **??** exhibits the round-trip delay distribution during 1 minute starting the 12th hour in the measurements. The above figures demonstrate that the inter-datacenter network delays across different regions are relatively stable and in most cases and highly predictable based on recent measurement data. Similar patterns are observed in other pairs of datacenters. The underlying reason of such stableness is that most cloud providers (e.g., Microsoft Azure [47], AWS [6], and Alibaba Cloud [2]) support private network peering, in which inter-datacenter traffic only traverses the provider's backbone infrastructure even though public IPs are used [65].

[8]We used the same software, a client, a server, and scripts, from Domino [65]