Straggler Resilient Practical BFT via Blind Agreement

Mohamed Yassine Boukhari
EPFL

Dr. Gauthier Voron
Project Supervisor
Distributed Computing Lab - EPFL

Abstract

With the escalating prevalence of malicious attacks and software failures, Byzantine fault-tolerant BFT state machine replication has emerged as a crucial technique to ensure the integrity, consistency, and availability of services. Despite numerous efforts to enhance their performance, BFT technologies remain vulnerable to machine and network heterogeneity which can impact the efficiency and reliability of such protocols, as they are designed under the assumption of uniformity in the underlying infrastructure and limit their usability in real world systems. We propose a variation of the Practical Byzantine Fault Tolerance algorithm that is resilient to nodes with limited network bandwidth. The key insight is that not all replicas need to receive the complete client request to participate in the consensus protocol. In our system, slower replicas rely on the decisions made by the faster, non-faulty replicas to achieve the required two-thirds majority consensus. We implemented the algorithm in Golang, demonstrating a stable throughput in the presence of stragglers.

1 Introduction

The rapid development of distributed systems has spurred extensive research on Byzantine fault-tolerant (BFT) consensus algorithms. BFT state machine replication protocols aim to enable a set of replicas to reach consensus on a sequence of pending requests, and endure arbitrary failures from a subset of these replicas while ensuring two basic security properties: safety, where honest replicas' outputs are consistent, and liveness, where honest inputs are eventually output by honest replicas.

Though these protocols carefully address such correctness properties, their authors often assume a homogeneous underlying infrastructure and neglect the effects of system heterogeneity. Consequently, they fail to address a major bottleneck in distributed computation: the significant performance deterioration due to slow servers, known as stragglers. Stragglers can arise from slow hardware, adverse network conditions, or other factors, leading to overall system inefficiency and reduced performance.

Since the introduction of the canonical *PBFT* [3], great efforts have been made to improve the performance of such protocols, but they mainly aimed to achieve more robust and optimised implementations of an already existing protocol such as *BFT-SMaRt* [2], reduce the total communication complexity (*Zyzzyva* [8] - *Hotstuff* [13] - *SBFT* [7]) or reduce the number of active replicas in the absence of faults [11], but none has treated the problem of having a subset of stragglers inside the system.

Existing BFT protocols operate in iterations, each consisting of two distinct phases: a broadcast phase, in which one or all of the nodes broadcast a batch of requests to the others and an agreement phase, in which the nodes vote for requests to execute. From a communications standpoint, the broadcast phase is bandwidth-intensive while the agreement phase typically comprises of multiple rounds of short messages.

The high bandwidth demand during the broadcast phase can severely strain nodes with limited network capacity. These stragglers, slow down the overall process, as the protocol cannot proceed until at least $2f+1$ honest nodes have downloaded the broadcasted request.

We aimed to confirm the previous hypothesis by measuring the throughput *BFT-SMaRt* which is a production grade leader based BFT system under different network conditions and topologies 1 when a demanding workload is applied. Our goal was to find which links become problematic when they have low bandwidth and thus allow us to understand better how to optimise the message exchange pattern in order to mitigate the effects of stragglers.

Our findings 2 show there is no drop in throughput when the link between the straggler and the leader of the consensus is not limited in terms on bandwidth but the links between the other replicas and the straggler are. However, once the bandwidth between the leaser and the straggler drops the throughout of the whole system drops significantly. This confirms the previous speculations. In this paper, we present SR-BFT, a new approach to build leader based BFT protocols that signif-

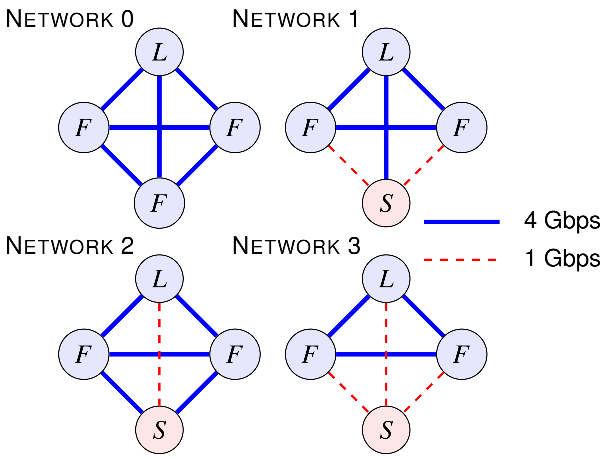

Figure 1: Studied Network Topologies

icantly improves performance in the presence of bandwidth variability. The key idea behind this is to broadcast the request only to the subset of nodes that have sufficient bandwidth and make the rest of the replicas agree on the small digest of the request which eliminates the requirement of high bandwidth between the leader and the subset of the slowest followers. We make this scheme possible by forcing the stragglers to wait for a weak quorum certificate before voting for the acceptance of a request. This guarantees the data availability of requests: when a straggler node accepts a commitment blindly into the log, it must know that the request is available in the network and can be retrieved at a later time by from at least one honest node in the network.

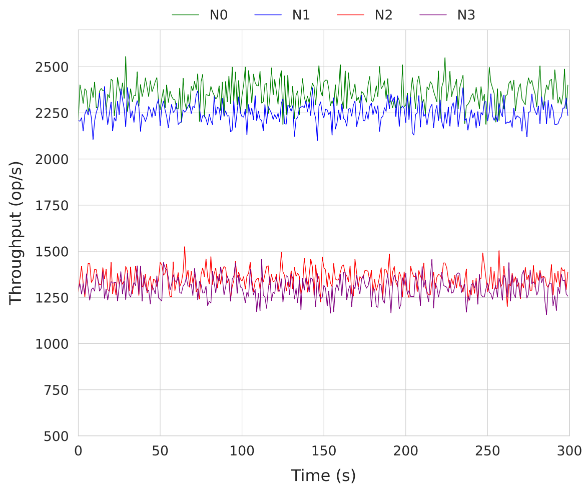

Figure 2: Throughput of *BFT-SMaRt* for topologies in 1

Contributions

We make the following contributions:

- **BFT Behavior under Bandwidth Variability:** We present an analysis of the performance of the BFT-SMaRt system under varying bandwidth of network links. Our findings highlight the pivotal role of the link between the leader and the straggler and provide insights into the effects of bandwidth constraints on BFT protocols.
- **Framework for BFT Prototyping and Simulation:** We introduce a framework designed for the prototyping and simulation of BFT systems. It enables fast development of Proofs of Concepts for new protocols and facilitates the simulation of diverse network conditions for testing and benchmarking purposes.
- **New SR-BFT Protocol:** We propose a novel BFT protocol that demonstrates stable throughput under the presence of stragglers compared to the already established BFT algorithms. We implemented it and conducted various experiments to evaluate its performance.
- **Laying the Bricks for a More Complete System:** We outline the foundational steps toward building a more comprehensive straggler resilient BFT system.

2 Related Work

BFT Consensus

Byzantine Fault Tolerance [9] is a generic approach that tolerates arbitrary failures, can be categorized into three types: synchronous BFT, partially synchronous BFT, and asynchronous BFT. Synchronous BFT protocols assume a known upper bound on message delivery times. Partially synchronous BFT protocols assume that after an unknown period of time, the system eventually behaves synchronously. We focus on Partially synchronous BFT.

PBFT provided the first practical solution to the Byzantine fault tolerance problem . PBFT employs an optimal bound of $n \geq 3f+1$ replicas, where up to f replicas can be controlled by Byzantine adversaries. It uses encrypted messages to prevent spoofing and replay attacks and to detect corrupted messages. PBFT follows a leader-based paradigm, guaranteeing safety in an asynchronous model and liveness in a partially synchronous model.

Committee Sampling in BFT

Pando [10] is a BFT protocol that selects a small committee for consensus and conveys the results to all replicas. it also decouples block transmission from agreement on the order, and optimizes both the communication and computational cost of the three underlying building blocks of BFT: transmission, consensus, and state transfer.

Separating Agreement from Execution

Previous works have aimed to reduce the costs of BFT protocols by separating different phases of the process. For instance, OMADA [5] and ZZ [11] separate the agreement phase from the execution phase, while DispersedLedger [12] decouples agreement from block retrieval.

3 SR-BFT Protocol

3.1 System Model

We consider a system consisting of a fixed set of $n = 3f + 1$ replicas, indexed by $i \in [n]$, where at most f replicas are Byzantine and other replicas are honest. Out of the remaining $2f + 1$ honest replicas, we assume that up to f replicas can be stragglers.

Network communication among replicas is point-to-point, authenticated, and reliable: an honest replica receives a message from another replica if and only if the latter sent that to the former.

The view Leader in each instance of the consensus predetermines the set of fast and straggler replicas. We suppose that we have a set of persistent stragglers in the system that can be determined statically.

3.2 Description

SR-BFT aims to reduce the communication overhead in BFT protocols, particularly focusing on minimizing the number of packets exchanged between the leader and straggler replicas on links with limited network bandwidth. This is achieved by leveraging two core insights.

First, the primary bottleneck in the communication pattern is the transmission of PRE-PREPARE messages as in most implementations, the client request is included in the message which increases significantly its size. Second, in a system of f faulty nodes and $3f + 1$ total nodes, a strong quorum of $2f + 1$ matching responses is needed to reach a consensus, but a weak quorum of $f + 1$ is enough to verify the legitimacy of a request. If $f + 1$ replicas accept a leader's proposal, at least one honest replica is contained in the quorum and the proposal is legitimate.

SR-BFT leverages the second insight to enable a set of replicas to vote blindly on a leader's proposal, without having to see and verify the request of the client. Blind Agreement is when a replica votes in favor of a proposal if it knows that $f + 1$ replicas have already accepted it. By relying on Blind Agreement, up to f stragglers do not need to receive the full PRE-PREPARE in order to participate in the consensus which allows them to conserve bandwidth that can then be utilized more effectively for other parts of the protocol.

3.2.1 Normal Case Operation

The consensus in replication consists of an array of linearizable consensus instances for each client request. In each consensus instance, replicas make *predicates* for the primary's instructions in each phase based on quorums that include votes from $2f + 1$ different replicas.

A consensus starts when a client invokes an operation to a leader, which starts the first phase in replication, *the request phase*. This is handled exactly as in [3] so we don't mention the exact behavior in this report.

After receiving a request, the leader starts the consensus among replicas with *the pre-prepare phase* in which the primary assigns a unique sequence number to the request. Then, it assembles the current view number, sequence number, and digest of the request message into a PRE-PREPARE message; the PRE-PREPARE message also includes the original request received from the client creating the FULL-PRE-PREPARE and omits it in the case of LIGHT-PRE-PREPARE. The leader multicasts the FULL-PRE-PREPARE to the fast replicas and LIGHT-PRE-PREPARE to the stragglers.

Next, the process enters *the prepare phase*. After receiving the PRE-PREPARE message, fast backups ① check that they are in the same view as the primary, ② confirm that the sequence number has not been assigned to other requests, and ③ compute the digest of the request to ensure the digests match. If the message is valid, backups proceed by broadcasting FULL-PREPARE messages to all replicas. On the other hand, upon receiving LIGHT-PRE-PREPARE message from the primary, stragglers proceed to do steps ①-② then wait until they receive $f + 1$ FULL-PREPARE messages from the other replicas. If this condition is satisfied, the straggler broadcasts a BLIND-PREPARE message.

Then, a replica collects PREPARE messages from the other backups. If the replica receives $2f$ PREPARE messages from different backups, the backup makes a predicate that the request is *prepared*. The predicate is supported by a *quorum certificate* consisting of $2f + 1$ agreements (including itself) of the same order from different replicas. Since there are at most f faulty replicas among the total of $n = 3f + 1$ replicas, the combination of pre-prepare and prepare phases ensures that non-faulty replicas have reached an agreement on the order of the request in the same view.

In *the commit phase*, replicas broadcast a COMMIT message. Similar to the previous phase, request is mark as *committed* when $2f$ COMMIT messages have been received from different replicas.

Finally, when predicate *committed* is valid, non stragglers start *the reply phase* and sends the client a REPLY message indicating that the consensus for committing the proposed request is reached. The client waits for a weak certificate to confirm the result. That is, it waits for $f + 1$ replies from different replicas.

3.2.2 Checkpointing and View changes

Non-stragglers periodically construct checkpoints of the application state to facilitate garbage collection of their pending request logs. In contrast, stragglers do not have access to

Message types	Message Structure
REQUEST	$\langle$ `op,timestamp,clientId` $\rangle$
FULL-PRE-PREPARE	$\langle\langle$ `view,seqNumber,digestOfRequest` $\rangle$, `request` $\rangle$
LIGHT-PRE-PREPARE	$\langle\langle$ `view,seqNumber,digestOfRequest` $\rangle\rangle$
FULL-PREPARE	$\langle$ `view,seqNumber,digestOfRequest,serverId` $\rangle$
BLIND-PREPARE	$\langle$ `view,seqNumber,digestOfRequest,serverId` $\rangle$
FULL-COMMIT	$\langle$ `view,seqNumber,digestOfRequest,serverId` $\rangle$
BLIND-COMMIT	$\langle$ `view,seqNumber,digestOfRequest,serverId` $\rangle$
REPLY	$\langle$ `view,timestamp,clientId,serverId,resultOfOp` $\rangle$

Table 1: The messaging formats in SR-BFT.

client requests and, therefore, cannot participate in the execution part of the protocol. Instead, stragglers rely on the state transfer protocol to retrieve the state when the number of fast replicas drops below $f+1$ and their participation in the execution becomes required. We preserve the same mechanism for state transfer as in [2] and leave the adaptation and optimisation of the protocol to future iterations of the project.

The view-change protocol provides liveness by allowing the system to make progress when the primary fails. View changes are triggered by timeouts that prevent backups from waiting indefinitely for requests to execute. We don't make any major modifications to the view change procedure and follow the same principles as in [3].

3.3 Correctness

We Prove the correctness of our algorithm in the case of $3f+1$ replicas where f are faulty and $2f+1$ are honest. Out of the $2f+1$ honest nodes, f are stragglers.

3.3.1 Safety

"The algorithm provides safety if all non-faulty replicas agree on the sequence numbers of requests that commit locally"

***Claim 1*:** If $f+1$ replicas, including the leader, decide on a proposal, then the outcome is honest and can't be overturned in this round of consensus.
The above claim implies that two non-faulty replicas agree on the sequence number of requests that commit locally in the same view at the two replicas.
The view-change protocol ensures that non-faulty replicas also agree on the sequence number of requests that commit locally in different views at different replicas. The proof is identical to the proof in [3].

3.3.2 Liveness

"The liveness property ensures that all requests from correct clients are eventually executed"

Honest Leader we assume an honest leader operates in the system. This allows us to state that the leader broadcasts the FULL/LIGHT-PRE-PREPARE messages to all the replicas in the system. In the optimistic case, the weak quorum of $f+1$ required by the stragglers to blindly vote agree to a proposal is composed only of honest replicas that will proceed to commit and execute the request and eventually deliver the response to the client. In the worst case scenario, f replicas out the $f+1$ behave maliciously and will not commit and deliver a valid response. We argue that under eventual synchrony, the remaining honest replicas will eventually receive and accept the proposal as per ***Claim 1*** and execute it.

Faulty Leader In the case that the current leader is faulty, the liveness property is guaranteed by the view-change mechanism. We don't make any changes to the view change mechanism and we claim that liveness is preserved based on the properties of PBFT [3]. We mention that stragglers consider a request as executed if they receive $f+1$ FULL-COMMITS for that request.

4 Implementation

4.1 KOLLAPS

Kollaps [6] is a network simulation tool agnostic of the application language and transport protocol that scales to thousands of processes and is accurate when compared against a bare-metal deployment or state-of-the-art approaches that emulate the full state of the network. It allows the definition of network typologies with easy customization of end-to-end properties (e.g., latency, jitter, bandwidth, packet loss) and network elements (nodes, links, routers, switches) while also supporting dynamic events such as node crashes or link removals. It supports applications running on bare metal or inside containers instantiated from Docker images.
Kollaps offers a Deployment Generator that integrates seamlessly with container images and orchestrators like Docker Swarm and Kubernetes, ensuring consistent performance and ease of deployment. In addition, it takes away the complexity of orchestration of the running processes which allows for fast and reliable simulations.

4.2 PAXIBFT

PaxiBFT [1] is a prototyping and evaluation framework for BFT protocols written in GOLANG. it provides a leveled playground for protocol evaluation and comparison. The protocols are implemented using common building blocks for networking, message handling, quorums, etc... and the networking interface encapsulates a message passing model, exposes basic APIs for a variety of message exchange patterns, and transparently supports TCP, UDP, and simulated connection with Go channels. PaxiBFT also includes benchmarking support to evaluate the protocols in terms of their performance, availability, scalability, and consistency.

In PaxiBFT, all BFT protocols can be implemented by coding the protocols' phases, functions, and message types. this is done by filling in three GO modules.

4.3 EASY BFT PRO SIM

We combined both KOLLAPS and PAXIBFT into a system for easy prototyping and benchmarking of BFT protocols while allowing for complex simulation scenarios on heterogeneous network conditions. We provide a set of scripts that automates the building and deployment of the docker containers executing the BFT service. This allows for rapid and easy end-to-end deployment of simulations, covering every step from the generation of necessary system configuration files, such as host IP addresses, to the retrieval of benchmarking logs from the executing containers. An overview of the framework is detailed in 3.

4.4 SR-BFT

We implemented a prototype of our straggler resilient protocol in GOLANG relying on PAXIBFT. We redefined the following modules.

- *messages.go* Protocol messages definitions as explained in 1.
- *replica.go* Interface between the client and the replica and defines the REQUEST handling functions.
- *srbft.go* Describes the core of the consensus protocol and represents the ideas explained in section 3.

5 Performance Evaluation

5.1 Experimental Setup

In this section, we evaluate our SR-BFT prototype under various workloads and network conditions, focusing on graceful execution scenarios. We compared its performance against a range of BFT protocols, including PBFT, HotStuff, Streamlet, and Tendermint, using the implementations provided by the authors of PAXIBFT. Our experiments were conducted natively on Ubuntu 22.04 (Kernel 6.1) with an AMD Ryzen 7 5800H processor featuring Radeon Graphics (16 cores). Each experimental result is averaged over three runs, with each run lasting until the measurements stabilized. We used four replicas, with one faulty node and one straggler. The experiments aim to evaluate the following aspacts:

- Throughput under varying network conditions
- Latency measurement in the presence of stragglers
- The effect of request size

By default, throughput is measured from the server's side, and latency is measured from the client's side.

5.2 Evaluation Results

Throughput under bandwidth variability

This experiment uses our framework to study the variations in throughput of different BFT systems in the presence of stragglers. The capacity of normal links is fixed at *10 Mbps*, and we progressively decrease the bandwidth of the stragglers until it reaches zero. We use a request size of 512 bytes and disable request batching. As shown in Figure 4, both *PBFT* and *HotStuff* are highly sensitive to bandwidth decreases, with their throughputs collapsing when the capacity of the straggler links drops below 80% of the regular links. *Streamlet* and *Tendermint* display similar trends, but the impact is less severe compared to the other two protocols. On the other hand, *SR-BFT* reaches its peak performance with only 30% of the regular link bandwidth and maintains stable throughput even when bandwidth decreases to a reasonable threshold. Although *HotStuff* outperforms *SR-BFT* in homogeneous network scenarios, our proposal is specifically designed to operate efficiently in networks with a high presence of stragglers.

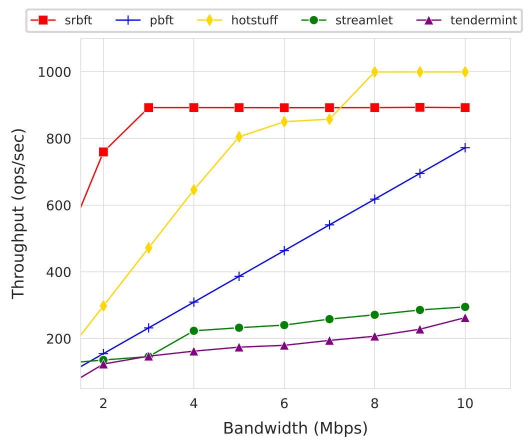

Figure 4: Throughput comparison under bandwidth variability

Latency in the presence of stragglers

In this experiment, we examine the latency under various throughputs for *SR-BFT* and *PBFT* in scenarios where stragglers have a bandwidth of 10% and 20% of the nominal links. As depicted in Figure 5, *SR-BFT* supports significantly higher throughputs at low latency compared to *PBFT* before reaching

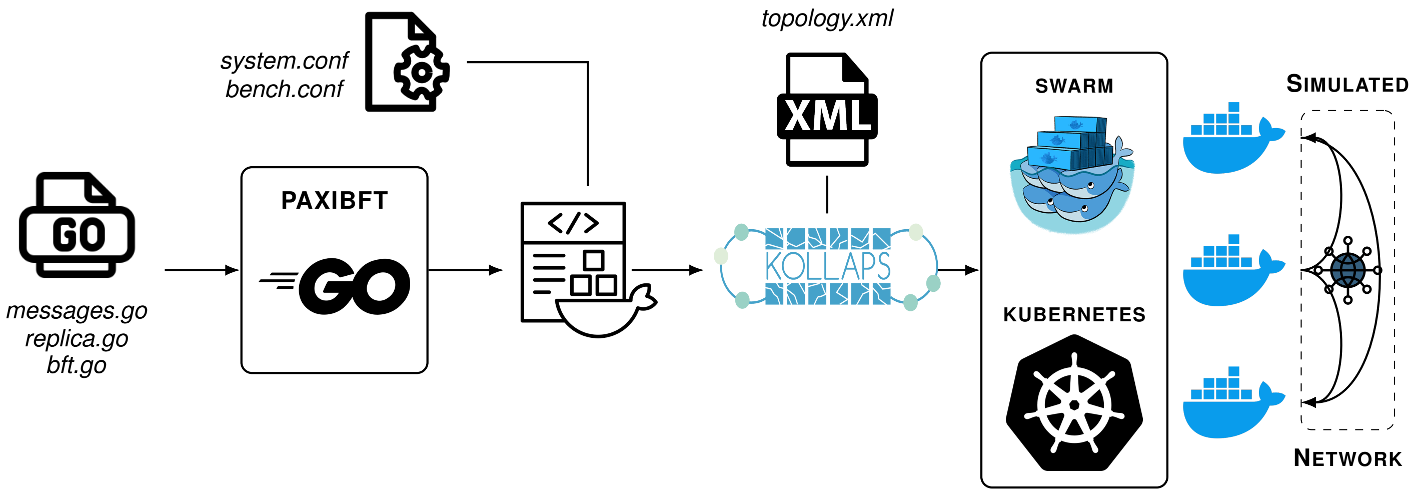

Figure 3: EASY BFT PRO SIM

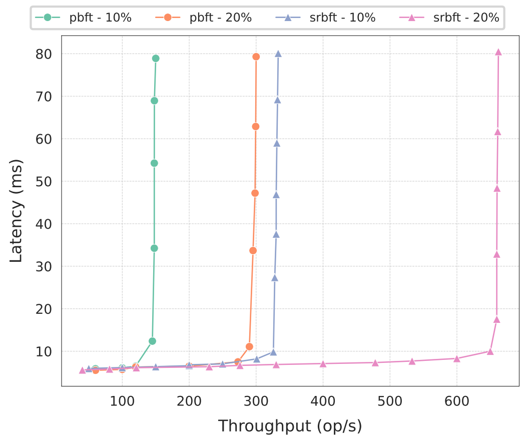

Figure 5: Throughput vs mean latency

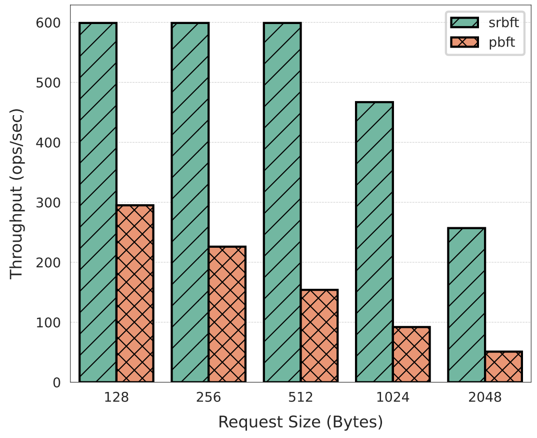

Figure 6: Throughput as a function of request size

a throughput threshold, beyond which the latency increases dramatically for both systems.

Throughput under bandwidth variability

We mentioned in the previous sections that the main bottleneck in leader based BFT systems is the broadcasting of the requests from the leader to the replicas. To further investigate this, we studied the effects of increasing the size of the requests on the performance of our system. Results are shown in figure 6.

6 Lessons Learned

While this work focuses on addressing the immediate challenges posed by stragglers in Byzantine Fault Tolerant (BFT) protocols, we delegate other foundational aspects of straggler-resilient BFT systems to future research. However, we mention them here for completeness.

A reputation-based leader rotation mechanism is crucial to effectively rotate leadership away from stragglers, as stragglers can significantly degrade system performance or even cause the entire system to halt. Passive protocols blindly rotate leadership among servers on a predefined schedule, potentially selecting unavailable or slow servers as leaders. PrestigeBFT [14] for example proposes an active view-change protocol using reputation mechanisms that calculate a server's potential correctness based on historic behavior.

Periodic state transfers are also essential to mitigate service downtime when a straggler must be included in the weak quorum. For example, in BFT-SMaRt, if a lagging replica is required for the consensus to advance, it can cause the system's throughput to plummet to zero while waiting for the slow replica to catch up. To address this, a more adaptable proactive recovery scheme needs to be designed.

Additionally, incorporating more bandwidth-friendly state transfer methods [4] will further enhance the system's robustness and efficiency. These improvements are vital for ensuring the reliability and performance of BFT systems in the presence of stragglers.

References

[1] ALQAHTANI, S., AND DEMIRBAS, M. Bottlenecks in blockchain

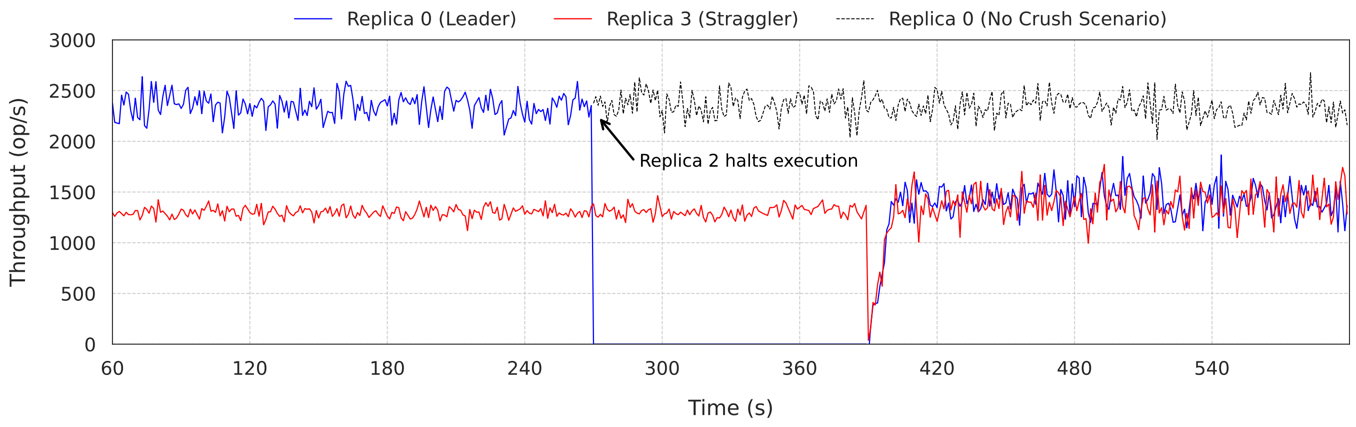

Figure 7: Throughput evolution of BFT-SMaRt across time and events

consensus protocols. In *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)* (Aug. 2021), IEEE.

[2] BESSANI, A., SOUSA, J., AND ALCHIERI, E. E. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2014), pp. 355–362.

[3] CASTRO, M. Practical byzantine fault tolerance.

[4] CHIBA, T., OHMURA, R., AND NAKAMURA, J. A state transfer method that adapts to network bandwidth variations in geographic state machine replication, 2021.

[5] EISCHER, M., AND DISTLER, T. Scalable byzantine fault tolerance on heterogeneous servers. In *2017 13th European Dependable Computing Conference (EDCC)* (2017), pp. 34–41.

[6] GOUVEIA, P., NEVES, J., SEGARRA, C., LIECHTI, L., ISSA, S., SCHIAVONI, V., AND MATOS, M. Kollaps: Decentralized and dynamic topology emulation, 2020.

[7] GUETA, G. G., ABRAHAM, I., GROSSMAN, S., MALKHI, D., PINKAS, B., REITER, M. K., SEREDINSCHI, D.-A., TAMIR, O., AND TOMESCU, A. Sbft: a scalable and decentralized trust infrastructure, 2019.

[8] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst. 27*, 4 (jan 2010).

[9] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4*, 3 (1982), 382–401.

[10] WANG, X., WANG, H., ZHANG, H., AND DUAN, S. Pando: Extremely scalable bft based on committee sampling. Cryptology ePrint Archive, Paper 2024/664, 2024. https://eprint.iacr.org/2024/664.

[11] WOOD, T., SINGH, R., VENKATARAMANI, A., SHENOY, P., AND CECCHET, E. Zz and the art of practical bft execution. pp. 123–138.

[12] YANG, L., PARK, S. J., ALIZADEH, M., KANNAN, S., AND TSE, D. DispersedLedger: High-Throughput byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (Renton, WA, Apr. 2022), USENIX Association, pp. 493–512.

[13] YIN, M., MALKHI, D., REITER, M. K., GUETA, G. G., AND ABRAHAM, I. Hotstuff: Bft consensus in the lens of blockchain, 2019.

[14] ZHANG, G., PAN, F., TIJANIC, S., AND JACOBSEN, H.-A. Prestigebft: Revolutionizing view changes in bft consensus algorithms with reputation mechanisms, 2023.