BFT-Blocks: The Case for Analyzing Networking in Byzantine Fault Tolerant Consensus

Richard von Seck, Filip Rezabek, Benedikt Jaeger, Sebastian Gallenmüller and Georg Carle
Chair of Network Architectures and Services, Technical University of Munich
{seck, rezabek, jaeger, gallenmu, carle}@net.in.tum.de

Abstract

Construction of robust and distributed systems is possible via the state-machine replication (SMR) approach. Architectures, featuring byzantine fault tolerance (BFT), offer additional robustness. Still, after more than 40 years of research, limitations on performance and scalability for practical systems remain. A large corpus of existing work improves on consensus complexity, performance and introduces a multitude of optimization techniques. The state-of-the-art is complex. On the other hand, many protocols designed for practical deployments are built on strong, common assumptions about underlying communication and authentication primitives. To fulfill these assumptions, often, commodity tools and libraries are employed without further analysis and caution for negative interplay.

Instead of contributing to the existing complexity, we choose a different approach. In this paper, we outline the feasibility and potential impact of the optimization of common building blocks of BFT-SMR systems. We systemize existing work in terms of common model assumptions and identify optimization potential. Finally, we choose the building block of networking transport as a representative example and analyze its optimization space, both in context of general BFT-SMR systems and a case study of the HotStuff protocol. We describe behavior, challenges, and desired configuration of network transports for use in byzantine agreement, and identify lossy links as the main catalyst for significant performance differences between protocols and configurations.

1 Introduction

Roughly 40 years ago, Lamport, Shostak, and Pease published their seminal papers on agreement in the presence of faults [1, 2]. The authors modeled behavior and solvability limitations for arbitrary behaving – byzantine – faulty processes, and provided basic solutions to consensus with Byzantine Fault Tolerance (BFT). The State Machine Replication (SMR) paradigm [3] allows building resilient distributed systems by introducing redundancy, masking a predefined number of f faults. A computational process, is redundantly executed on n independent nodes, we call *replicas*. By enforcing a total order on the inputs to the replica state machines, e.g., via (BFT) consensus, all $(n - f)$ non-faulty or *honest* replicas will exhibit the same behavior. Aggregating the final replica results allows masking faults of a minority of faulty replicas. However, SMR and BFT consensus come at a price. Replica execution needs to be synchronized, and the necessary protocols incur a significant communication and processing overhead. Consequently, a plethora of work has been dedicated to the optimization of SMR and BFT systems.

Despite these activities, limitations for practical use remain. Hence there is potential for further optimization. In fact, the limitations are so significant that even modern systems, focused on scalability, struggle to maintain performance beyond hundreds of replicas [4–7]. The theoretical bounds for communication complexity depend on the model assumptions, but modern systems are typically in the order of $\mathcal{O}(n^2)$ [5]. Besides limitations from performance and scalability constraints, Chondros et al. [8] identified a lack of usability of existing approaches, opaque performance overhead for target use cases, and significant engineering effort as an obstacle for practical deployment. Kotla et al. [9] identified system and configuration complexity as a hindrance of practical adoption, as a sensible choice in the parameter space is increasingly hard.

Even though many publications aim for construction of *practical* systems, a majority of BFT-SMR optimization research is focused on the study of conceptual modifications to agreement algorithm or model assumptions [10]. Problems with this approach have been identified on multiple accounts. As outlined by the authors of Zyzzyva [9]: *"Why another BFT protocol? The state of the art for BFT state machine replication is distressingly complex."*. This statement is underlined by a detailed survey of total order broadcast algorithms and their complexity as early as 2004 by Défago et al. [11]. Guerraoui et al. [10] assess BFT-SMR systems to be "*notoriously difficult to develop, test and prove*". Singh et al. [12] suggest that algorithm-level optimization is mostly subject to tradeoffs and even go so far as to state that "*one-size-fits-all protocols may be hard if not impossible to design in practice*".

Our Approach Instead of contributing to the existing complexity and introducing – possibly breaking – changes to protocols with already proven safety properties, we argue for a different approach. As we show in §2, a majority of practical systems shares a common subset of model assumptions. These also result in common building blocks, used across implementations. By optimizing a necessary building block, safe improvements can be achieved that apply to a large number of systems. Furthermore, we argue that design and optimization of *practical* BFT-SMR systems require revisiting some of the established assumptions. We agree with recent research on longest chain consensus [13,14] that a mismatch between the established, idealized network model and the challenges of *practical* BFT-SMR systems exists. Therefore, we focus on analysis and optimization of the underlying networking layer. We study the effects of different transport protocol behavior on BFT-SMR systems in general and conduct a case study on an exemplary state-of-the-art protocol, HotStuff [5].

Key Contributions

KC1 Outline feasibility and impact of building block optimization in BFT-SMR systems

KC2 Systematization of practical BFT-SMR systems in terms of networking architecture

KC3 Optimization space analysis of the network transport building block for BFT-SMR systems, including a case study of HotStuff

As many systems share network transport building blocks, our findings are generally applicable to all related BFT-SMR protocols. The rest of the paper is organized as follows: In §2 we argue for optimization of building blocks in BFT-SMR systems as a high-impact approach (**KC1**, **KC2**). For this paper, we choose the building block of network transport for further study and analysis. We discuss related work in §3. In §4 we introduce HotStuff, an exemplary, state-of-the-art BFT-SMR system. We conduct a network impact case study on HotStuff in §5, identifying bottlenecks and discussing message loss (**KC3**). In §6 we explore the optimization space of the network transport layer for BFT-SMR protocols in general (**KC3**). We conclude our paper in §7.

2 Building Block Optimization

To make our case for the impact of optimizing BFT-SMR building blocks and select suitable targets, we outline common model assumptions and techniques, used in existing systems. A diverse set of optimization approaches for BFT-SMR protocols has been developed. However, even for fundamentally different systems, a common subset of core assumptions emerges:

I. Reliable communication [4–7, 15–24]

II. Point-to-point interconnections between all nodes [4–7, 10, 15–19, 21–28]

III. Authenticated messages [4–7, 9, 10, 15–29]

From a BFT-SMR protocol perspective, these assumptions offer a more relaxed design space. They help to mask network-related transmission failures (**I.**), abstract possible underlying network topology as well as influence on transmission by malicious nodes (**II.**) and reduce the required number of replicas (e.g., assuming synchronous communication) to achieve consensus (**III.**). To approximate these assumptions in real deployments, additional overhead from underlying technology is introduced. Assumptions I. and II. are typically realized using TCP/IP connections between replicas [4–7, 15, 16, 26, 28]. Assumption II. is typically not satisfied through direct hardware interconnections (consider deployment in large scale Wide Area Network (WAN) setups [7]) but only emulated by logical connections between each possible pair of nodes. On a network level, the exchanged data is then still routed across a (TCP/)IP network. In addition to BFT-SMR protocol message authentication (e.g., via cryptographic signatures) from Assumption III., some systems employ generalized secure channel implementations such as TLS as part of their networking stack [4–7, 15]. Consequently, optimization of the transport protocol or networking stack directly improves performance of all protocols, that employ the respective building block.

3 Related Work

A plethora of approaches, optimizing BFT-SMR systems has been published so far (§2). We are aware of approaches that leverage special networking hardware [30–32] or communication primitives like Remote Direct Memory Access (RDMA) [33, 34] for performance improvements. However, these specialized requirements limit the applicability to e.g., data center deployments. As we focus on general network transport optimization, we consider this work only slightly related. For brevity, we discuss only the most closely related approaches, theoretically investigating usage and modification of standard network transport in BFT-SMR systems. Secondly, we discuss works with general or network bottleneck analysis.

Study of Network Transport Clement et al. proposed the BFT protocol Aardvark [35], focused on more robust and performant operation under actual byzantine behavior in the network. The authors studied the performance of a selection of BFT protocols, as well as their response to flooding with TCP and UDP traffic. Resulting performance degradation with either transport protocol is attributed to the implementation, not the general protocol design. Chondros et al. [8] discuss the usage of UDP in PBFT and outline its possible robustness and performance limitations in case of packet loss. The authors conclude, that in a practical setup, unreliable communication has a significant impact on performance and robustness. Lorünser et al. [36] conducted a probabilistic study of the impact of TCP and UDP usage on PBFT. The authors propose an optimistic usage of UDP-based communication, combined with forward error-correcting codes to address possible packet loss. Validation of the model is conducted using simulations with OMNeT++. Packet loss of up to 30 % is studied, for which the authors assess significant performance differences.

Recently, more research has started to question and extend the prevalent networking assumptions (§2) to address the gap between model and real-world implementation. This work mostly applies to the permissionless case of webscale Blockchain systems. Motivated by spamming attacks on longest chain Proof-of-Stake (PoS) consensus, Neu et al. [13] introduce an extended networking model to capture bandwidth constraints. They propose modified download rules for which they prove security in bandwidth-constrained networks. Another contribution to longest chain PoS consensus is presented by Corretti et al. [14]. The authors propose a byzantine-resilient gossip protocol, designed to impede resource-restricted attackers.

Bottleneck Analysis in Literature Previous work identified several causes for performance limitations in BFT-SMR systems. For single-leader-based protocols, the leader is consistently identified as a system bottleneck node [7, 9, 19, 32, 37–39]. However, in a more detailed analysis of the underlying bottleneck, the literature provides varying reasoning for different systems and environments. In high-performance-low-latency environments [32, 33, 39] as well as massively scaled scenarios [4, 7] the processing overhead of network communication is identified as bottleneck. Wang et al. [33], as well as Dang et al. [32] identify OS kernel processing to be the main source of overhead. Stathakopoulou et al. [7] assume leader bandwidth to be the first, and client authentication to be the second bottleneck.

Older protocols, which use asymmetric cryptography for signing replica votes, identify cryptographic processing cost as main overhead [40, 41]. But also newer works, which employ MAC-based authentication [9, 25], modern BFT-SMR optimization techniques such as batching, multiple leaders or threshold signatures [5, 19, 26, 42] argue for cryptography processing to be the main bottleneck. Other publications just identify a general CPU bottleneck, without further distinguishing a cause [37, 43, 44]. To summarize, except for the leader node in single-leader protocols, a "general" bottleneck is not consistently identified, as it depends on the system environment, protocol and configuration. Previous work has assessed a tendency towards increased network overhead in large WANs and high-performance-low-latency networks. Additionally, the choice of protocol parameters influences the shift of the major overhead source. As already outlined by Stathakopoulou et al. [7], a smaller payload size per request shifts the bottleneck towards a CPU and verification load, while larger payload sizes may result in bandwidth limitations.

4 HotStuff

HotStuff [5] is a leader-based BFT-SMR protocol, designed for the partial synchrony model, and is used as base for the design of the Libra Blockchain project [45]. HotStuff assumes authenticated, reliable, and point-to-point connections; realized in the author implementation through a combination of TCP/TLS channels and elliptic curve signatures on votes.

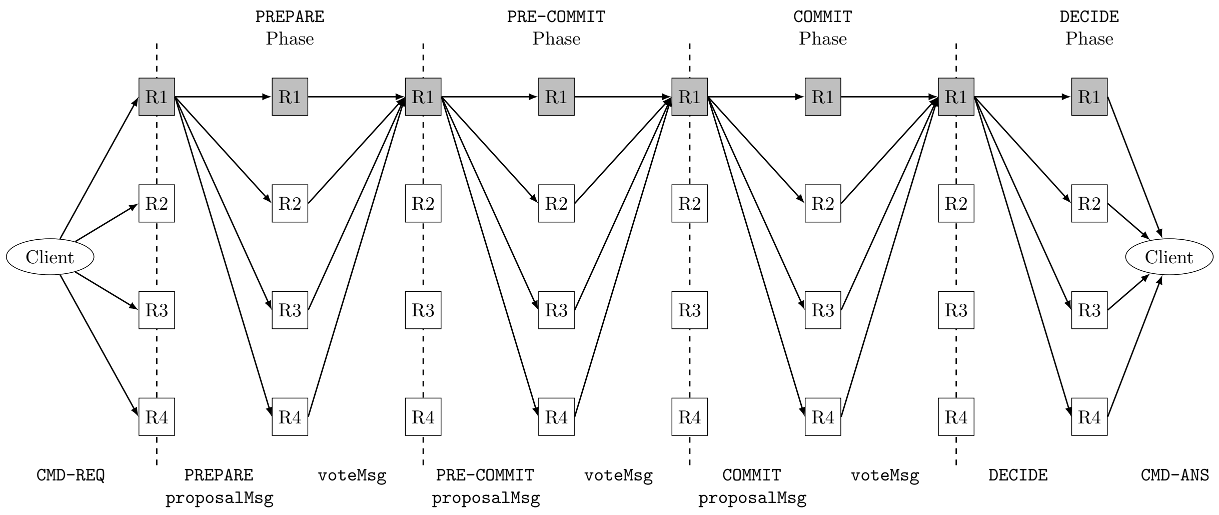

Figure 1: Communication pattern in basic HotStuff [5]

Basic Communication Communication in a practical HotStuff deployment comprises two main categories. (1) Communication between a client and the replicas, and (2) communication of replicas with each other. Each replication or *view* starts with a client, sending a command request to all replicas. After the replicas have reached agreement on request order, each replica sends back a response to the client. The client waits for at least $(f + 1)$ matching responses from the replicas to register the issued command request as committed. Communication between replicas is conducted in a broadcast-vote pattern, as displayed in Fig. 1. Except for fetching missing state, replicas are not communicating between each other, but only with the current leader. The leader broadcasts a voting request to all replicas which verify the message and respond with a vote. The leader collects the votes and, given $(n - f)$ votes, forms a Quorum Certificate (QC) from the signatures of all correct replies. This QC is then used to justify the next step in the agreement protocol. Note that a leader replica both broadcasts to itself, as well as votes for its own broadcasts.

In more detail, one view is organized into three core *phases*, namely `PREPARE`, `PRE-COMMIT`, and `COMMIT`. In the `PREPARE` phase the current leader collects $(n - f)$ `NEW-VIEW` messages, legitimatizing his current reign and carrying the latest valid QC, known to the replicas. Using this knowledge, the leader forms a proposal for the next unprocessed client request (batch), justified by the most recent, valid QC. Following the same, basic communication pattern, in the `PRE-COMMIT` and `COMMIT` phases the leader asks the replicas by broadcast to advance to the next phase, collects replica votes and forms another QC. After receiving $(n - f)$ `COMMIT` votes, the leader executes the final `DECIDE` phase, in which this `COMMIT` QC is wrapped into a `DECIDE` message and sent to the replicas. Upon receiving this notification, the replicas execute the requested command and answer the client. Replicas persist their operation state, including decision, operation hashes and QCs in a *block*, which is appended to the local replicated data store, the "Blockchain".

Pipelining As an optimization to the basic protocol described above, where each decision on one (batch of) client commands requires to pass through multiple phases, HotStuff introduced pipelining of views. Since all phases are very similar in their basic communication pattern of broadcast-vote, a single QC can be used in parallel in different view executions to justify progress. The similarity of the executed phases is also translated into notation, where no distinction between phases is made, and only `GENERIC` phases

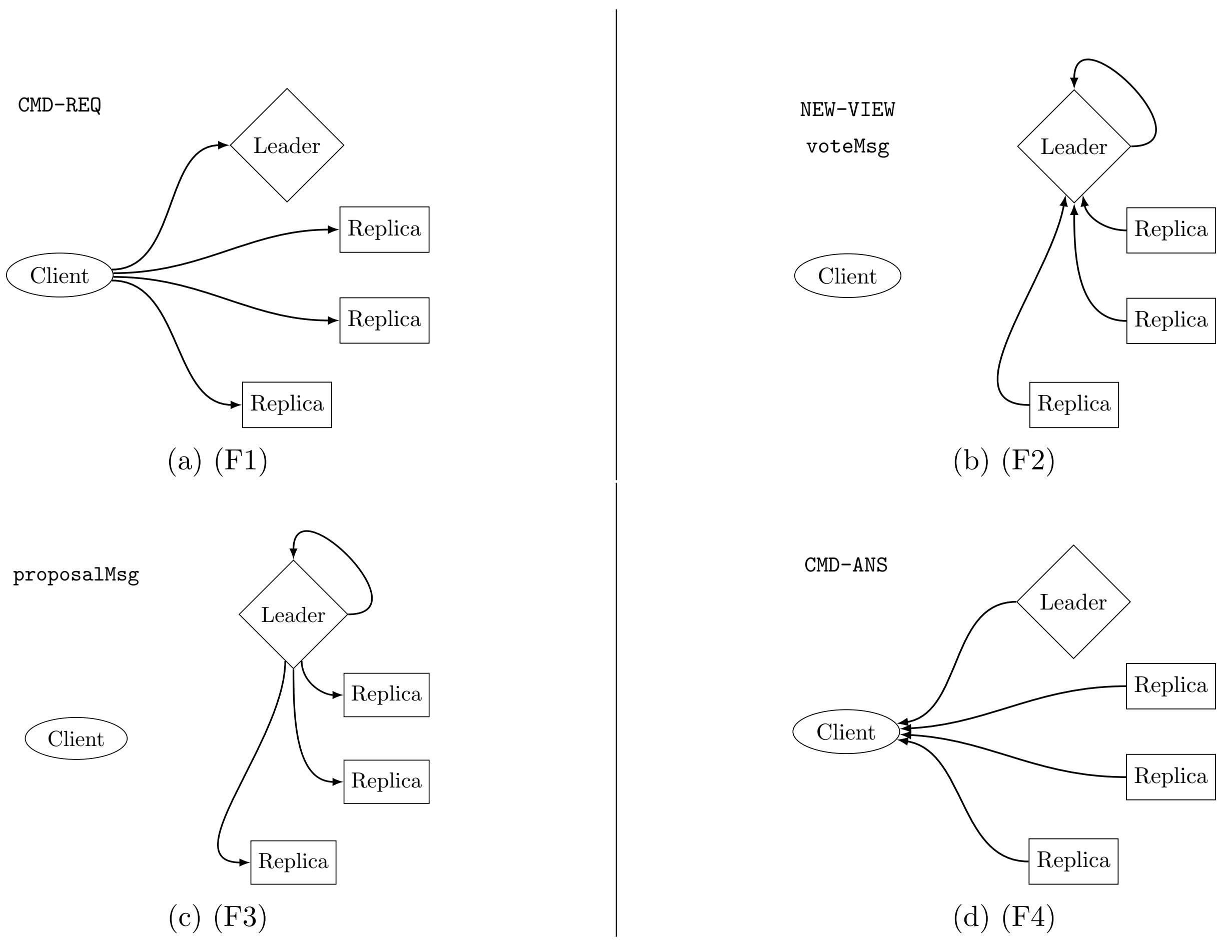

Figure 2: Regular communication flows in HotStuff

and `GENERIC`-QCs exist (Algorithm 3, [5])

5 Network Impact Case-Study

We study network effects on HotStuff as exemplary, leader-based BFT-SMR system. This includes a flow analysis to identify bottlenecks, and a discussion of message loss impact. The case study serves as reference model for our optimization analysis of network transports in general BFT-SMR (§6).

5.1 Flow Analysis

We analyze frequency and payload size of transmissions during a typical protocol run. To simplify the model, we assume $n = 4$, a single client issuing requests and no lost command requests from client to replicas. As outlined in §4, HotStuff features two dominant communication patterns. One between client and replicas and the second between the replicas themselves. Differentiating by transmission direction, we end up with four general types of data flows between nodes, pictured in Fig. 2. We enumerate these four flows as Flow 1 (**F1**) to Flow 4 (**F4**). Flows in Fig. 2 consist of a certain set of messages, which we take into account for our analysis. We adopt the message names where defined in the original HotStuff paper. **F1** comprises only the client to send a command request (`CMD-REQ`) to all available replicas. **F2** describes communication from the replicas to the current leader, such as `NEW-VIEW` messages and votes (`voteMsg`) for received proposals. **F3** contains new proposal messages from the current leader to other replicas. Finally, **F4** describes replicas, sending command answers (CMD-ANS) back to the client, after commit of the respective requested operation.

5.1.1 Payload Analysis

Assume that peer P has outgoing ($\rightarrow$) communication in flow ϕ. Then we denote this flow as $P_\phi^\rightarrow$ and the size of the transmitted payload as $|P_\phi^\rightarrow|$. Incoming communication is denoted by ($\leftarrow$) respectively. The payload size of a certain message m is denoted by $|m|$. Following this notation, we now analyze the payload size of the roles of Client (C), Replica (R) and Leader (L) for the flows and message names defined in Fig. 2. Note that *leader* and *replica* are handled as separate roles in the following discussion, but in practice one physical node fulfills both roles at the same time, incurring the total overhead of both roles.

$$|C_1^\rightarrow| = n \cdot |\texttt{CMD-REQ}| \tag{1}$$
$$|R_1^\leftarrow| = |\texttt{CMD-REQ}| \tag{2}$$

In **F1**, the client sends out one command request to each registered replica, which receives exactly this request.

$$|R_2^\rightarrow| = |\texttt{NEW-VIEW}| \tag{3}$$
$$|\overline{R_2^\rightarrow}| = |vote(\texttt{GENERIC})| \tag{4}$$
$$|L_2^\leftarrow| = n \cdot |R_2^\rightarrow| = n \cdot |\texttt{NEW-VIEW}| \tag{5}$$
$$|\overline{L_2^\leftarrow}| = n \cdot |\overline{R_2^\rightarrow}| = n \cdot |vote(\texttt{GENERIC})| \tag{6}$$

In **F2**, replicas either send NEW-VIEW messages to the new leader (3), or vote for the current proposal (4). Since the available codebase implements the pipelined HotStuff variant, votes and proposals are of type GENERIC. Conversely, the respective leader receives at least $(n - f) \in \mathcal{O}(n)$ messages in either case. While the HotStuff code only considers the first $(n - f)$ valid votes for further cryptographic processing (e.g., creation of QCs), the incoming messages are still rudimentary processed before this decision. For our model we assume a scenario in which all replicas participate in this step (fault-free execution, maximum possible load). Thus, we model the payload strain with n responses.

$$|L_3^\rightarrow| = n \cdot |proposal(\texttt{GENERIC})| \tag{7}$$
$$|R_3^\leftarrow| = |proposal(\texttt{GENERIC})| \tag{8}$$

In **F3**, the leader transmits the next proposal to all replicas, which receive this message once.

$$|R_4^\rightarrow| = |\texttt{CMD-ANS}| \tag{9}$$
$$|C_4^\leftarrow| = n \cdot |\texttt{CMD-ANS}| \tag{10}$$

In final flow **F4**, all replicas answer the client request. While the client only requires $(f+1)$ matching CMD-ANS messages in theory, excess messages are still received and interpreted by both networking stack and HotStuff client code. We therefore model the payload strain in (10) with n responses.

Independent of the actual payload size, we can directly identify two potential bottlenecks in this communication structure. As outlined in previous literature (§3), the single leader is subject to high transmission and processing volumes, scaling with the number of replicas n. However, considering a single client setup, also the client is subjected to considerable transmission and processing load, scaling with the number of replicas n. The operation of multiple parallel clients, requesting independent operations, affects both F1 and F4. Denoting the number of clients as c, we find

$$|C_1^{\rightarrow}|' = n \cdot |\texttt{CMD-REQ}| \tag{11}$$

$$|R_1^{\leftarrow}|' = c \cdot |\texttt{CMD-REQ}| \tag{12}$$

$$|R_4^{\rightarrow}|' = c \cdot |\texttt{CMD-ANS}| \tag{13}$$

$$|C_4^{\leftarrow}|' = n \cdot |\texttt{CMD-ANS}| \tag{14}$$

As we see, an increasing number of clients results in more strain on the replicas (cf. (12), (13)), potentially shifting the bottleneck towards the replicas. The general strain on the client becomes more pronounced, when command batching is active. While agreement protocol messages between replicas (F2 and F3) only contain a batch of *hashes* of client commands to execute, client requests (F1) and responses (F4) directly contain batches of the *full payload*[1]. In other words, the message size of |`CMD-REQ`| and |`CMD-ANS`| scales multiplicatively with both batch size and payload size. As `CMD-ANS` messages in F4 are sent by a majority of the n replicas at roughly the same time, the client strain in this scenario is considerable.

5.2 Effect of message loss

If not masked by the underlying transport protocol, packet loss may significantly impact operation and performance of the BFT-SMR protocol. In the following we analyze this impact on HotStuff flows (Fig. 2) and argue for the necessity of reliable communication for safety and liveness.

F1 In F1, loss of `CMD-REQ` messages can cause instability of the protocol execution. In HotStuff, consensus decisions over a client command are formed via communication of *command hashes*, not the actual command payloads. In the HotStuff implementation, the commands to execute are only transmitted via `CMD-REQ` messages from client to replica. While the replicas can request missing "blocks" (§4) from other replicas, if they receive a proposal with an unknown block hash, these transmitted blocks do not contain the actual payloads to execute. Furthermore, replicas do not check if they received a `CMD-REQ` message from the client before voting on proposals. Hence, if transmission of a `CMD-REQ` fails, the replica is not able to obtain (and finally execute) the missing commands, except if the agreement fails and the commands are resent by the client. At the same time, these replicas will not send a `CMD-ANS` message to the client in F4, even if the agreement on the hashes succeeded. Thus, either replicas, missing the commands, will fall behind the state of other replicas, or the client will not receive enough $(f+1)$ responses to accept the committed decision. If the transmission of `CMD-REQ` to the leader fails, it will not create a proposal for these commands and a new leader is elected. If a client waits indefinitely for some answer to its issued `CMD-REQ` messages, the whole process might get stuck for subsequent transmission failures to leaders, as the number of `CMD-REQ` messages in flight is limited in the HotStuff implementation.

F2 Loss of a `voteMsg` does not necessarily result in a failed agreement, as long as the leader still receives $(n - f)$ other valid votes from which he can form a QC. If the leader makes progress, the next successful proposal message will allow the replica to catch up.

[1]The HotStuff codebase allows independent configuration of request/response payload size, we assume equal payload sizes for our paper.

If the leader receives less than $(n - f)$ valid votes, the current view cannot continue and a new leader will be elected. Leader election in HotStuff is realized by the replicas sending `NEW-VIEW` messages to the new leader. Once the new leader has collected $(n - f)$ `NEW-VIEW` messages, he can create a new valid proposal. Otherwise the leader will never create a proposal for this view and execution will stall until the next `NEW-VIEW` interrupt.

F3 In case a `proposalMsg` is lost, the affected replica r can not participate in the current voting round. Additionally, r will not transition its view state, as it is missing the QC and block from the proposal. However, the current leader and other replicas that received the proposal can indeed make progress, assuming there are enough correct replicas left to form a quorum. If the other replicas make progress, r cannot directly participate in the next view and first needs to catch up to the current block using `BLOCK-REQ` messages. Due to this delay, r could potentially lag behind, preventing progress.

F4 Loss of `CMD-ANS` messages mainly affects the client in his assumption of the replica state. If $(f + 1)$ valid and equal responses are received, the BFT-SMR protocol can continue regularly. However, if the replicas accepted and executed a client command c, but the client *does not* receive enough responses, the client view and the actual state of the system diverge. This is a problem if e.g., c is not an idempotent operation on the replica state. The client assumes that execution of c failed and might schedule a request of c for a second time. The exact implications of this behavior is dependent on the application, that is to be replicated.

6 Network Transport Analysis

In this section we analyze transport protocol characteristics, configuration space, congestion control and their interaction with BFT-SMR systems in general. We choose UDP, TCP, and SCTP [46–48] as representative transport protocols for our discussion. Where suitable, we provide context from our HotStuff case study (§5) for clarification.

6.1 Reliable transmission

As established in §2, the main task of the transport protocol is to provide the abstraction of a reliable, point-to-point connection between peers. Therefore, we focus our discussion to two situations, operation without and operation with packet loss. We deem dedicated attacks on transport-layer level (e.g., TCP SYN Flood) out of the scope of this paper and refer to the respective literature for performance impact and defense strategies. Care should be taken to avoid IP fragmentation by choice of protocol or configuration parameters, as it bears potential for both performance reduction and security concerns [49, 50]. Either the transport protocol or the above application (layer) should ensure that transmitted payloads do not violate the path Maximum Transmission Unit (MTU).

Operation without loss If the BFT-SMR protocol is executed in an environment without loss, masking of omission failures does not need to be handled by the transport protocol. Devoted resources to this problem are unnecessary overhead.
We first discuss UDP. Due to its connectionless nature, no initial setup between communication partners is necessary. UDP thus avoids the small delay caused by initial handshake round trips. The average delay impact of the connection setup becomes negligible, the

longer the connection actively persists. For operation modes with strict low-latency requirements or modes where requests are sent in high-throughput bursts, with considerable pause between two bursts, the impact may be higher. No additional state is saved for transmitted or received packets. If no IP fragmentation takes place, possible reordering of packets during transmission does not affect plain UDP throughput. Reordering needs to be handled by the consuming application/layer, if necessary at all. Without omission failures, UDP thus provides small initial delay and reduced processing and state overhead during transmission.

For TCP, a three-way-handshake (> 1 Round-trip Time (RTT)) is necessary before sending any data. This extra delay must be taken into account for low-latency requirements or burst sending operation. One of the core features of TCP is its abstraction of an ordered byte-stream communication interface. Without omission failures, logic and state keeping for retransmission of lost packets is unnecessary overhead. So is the logic for ordered delivery, in case ordering is not required. Additionally, TCP features mechanisms for Congestion Control (CC), which may contribute to but also interfere with performant BFT-SMR protocol operation (§6.3). Adequate scaling of transmitted BFT-SMR payloads to prevent MTU exceeding is helpful to achieve good performance. However, TCP is also able to autonomously split submitted data, if the size of the payload, the configuration, and a suitably negotiated Maximum Segment Size (MSS) [47] allow. A more detailed discussion of protocol configuration is provided in §6.2.

Consequently, without omission failures, TCP usage results in state and processing overhead. More so, if the upper layer does not require strictly ordered transmission. Initial connection setup delay might be detrimental to low-latency request and burst transmission operation.

Finally, we analyze SCTP. A four-way-handshake (≥ 2 RTT) is required before transmission of payloads can proceed. Akin to TCP, this might negatively affect low-latency requirements or burst sending operation. Without omission failures, SCTP's retransmission logic and state are unnecessary. Ordered delivery for messages can be deactivated, providing more flexibility. Like TCP, SCTP provides CC mechanisms. SCTP additionally provides path MTU discovery and user data fragmentation [48]. Thus, without omission failures, SCTP involves more overhead than TCP, considering a costlier initial handshake, as well as retransmission and CC logic overhead.

Operation with loss During the operation of BFT-SMR systems, especially in setups without Quality of Service (QoS) guarantees, packet loss can occur. In order to prevent such omission failures to be interpreted as faulty replicas by the consensus protocol, typically reliable connections are assumed (§2). If these omission failures are not masked, they potentially trigger expensive slow-path fallback routines in optimistic protocols [6,9, 51] or a leader-change in leader-based protocols. If the number of omission failures exceeds the expected number of faults f, progress can be prevented completely. Depending on the employed transport protocol in a practical setup, the effects of packet loss on performance and general system behavior may differ.

In case of UDP, no reliability and no additional state logic is provided. UDP does not track if a packet is lost and does not take any further action. No state is changed. Thus omission failure handling responsibility is directly transferred to the higher layer. The effects on the BFT-SMR protocol are immediate. Our analysis of omission failure impact on HotStuff (§5.2) demonstrates, that state desynchronization is possible when using plain UDP on client $\leftrightarrow$ replica links. Safety impact on other BFT-SMR protocols may vary, although unhandled omission failures will likely reduce performance.

TCP provides explicit logic for handling retransmission of lost segments, as well as congestion and flow control. TCP can detect loss of segments by keeping track of sequence numbers, following the amount of transmitted bytes. The receiver sends Acknowledgments (ACKs) to the sender, stating the sequence number of the next expected byte. As reordering of segments during transmission can occur, not every out-of-order reception is caused by loss. To detect loss, modern TCP employs two techniques: A Retransmission Timeout (RTO) [47] and Duplicate Acknowledgments (DupAcks) [52]. The sender expects a (cumulative) ACK for each byte transmitted. If no ACK is received for a certain time, a retransmission is triggered. The actual timeout (RTO) is calculated as a bounded, smoothed and delayed estimation of the current RTT [47]. If a retransmission is triggered because the RTO expired, the current RTO is doubled, resulting in an exponential "back off" behavior [53]. In modern versions of the Linux kernel (e.g., v4.19.194), the lower and upper bounds for TCP RTO are defined as 200 ms and 120 s, respectively, assuming a kernel timer interrupt frequency of 1000 Hz. However, usually the delay between two successively sent segments is much smaller than the configured `TCP_RTO_MIN` value. If a segment is lost during transmission, the receiver might receive data out-of-order. In this case, the receiver sends an immediate DupAck to inform the sender of the originally expected data. Since TCP masks omission failures by retransmission, the BFT-SMR protocol is affected in terms of performance. Besides the overhead due to retransmission logic, suboptimal choice of timeout values (e.g., RTO) and omission faults interpreted as network congestion can reduce the available transport protocol throughput [54].

In a similar fashion to TCP, SCTP provides reliable transmission as well as CC and Flow Control (FC). In order to realize reliable transmission, SCTP employs sequence numbers. However, as SCTP also allows for unordered transmission, network transmission sequence and sequence of delivery to the higher-level application are kept logically separate. In other words, SCTP accepts and caches incoming datagrams, even if the contained data chunks are out-of-order. Instead of exclusively cumulative ACKs as in TCP, SCTP also uses Selective Acknowledgements (SACKs), which acknowledge certain ranges of received data chunks, using explicit *Gap Ack* specifications [48]. To detect loss, SCTP employs two techniques: Explicit gap specification in SACKs and an RTO. The calculation of SCTP RTO is very similar to the calculation performed by TCP, including the exponential back-off behavior described before. The recommended values for SCTP RTO configuration [48] are slightly differing from the TCP defaults with `RTO.Min` = 1 s and `RTO.Max` = 60 s. SCTP also provides CC. The basic algorithms used by SCTP are also based on TCP CC, as defined in RFC 2581 [52]. Differences mainly emerge for multihoming setups. As SCTP masks omission failures by retransmission, again the upper layer BFT-SMR protocol is affected in terms of performance. While suboptimal configuration of protocol parameters (e.g., RTO) can have a negative impact, configuration is easier for SCTP due to its accessible socket option interface.

6.2 Impact of Protocol Configuration

While a complete analysis of all options is out-of-scope, we discuss parameters with potential impact upon execution of a BFT-SMR protocol. We begin with an analysis of general protocol options and discuss details of CC afterwards. We limit our discussion of general transport protocol options to network stack implementations for a recent Linux Kernel (i. e., v4.19.194). Configuration is generally possible through three interfaces, the `proc` filesystem, socket options and modification of kernel parameters or code. Most configuration options are provided by the TCP stack, SCTP does not offer configuration via `procfs` and the number of relevant UDP configuration options is negligible. Relevant options can be categorized along two classes: (I) Delay and coalescence of small writes

Option	Layer	Effect
tcp_autocorking	procfs	Merge socket writes in the same flow, if at least one packet in Queuing Discipline (qdisc) or device transmit flow
TCP_CORK	Socket	Queue small submitted writes until a full frame can be sent, this option is removed or 200 ms elapsed
TCP_NODELAY/ SCTP_NODELAY	Socket	Nagle's Algorithm off, segments are sent out as early as possible
TCP_QUICKACK	Socket	Toggle Quick ACK mode. If active, ACKs are sent immediately. Non-permanent, overridden by kernel.
TCP_DELACK_MIN/ TCP_DELACK_MAX	Kernel	Configure min and max delay time if delayed ACKs are active

Table 1: Delay and Coalescence behavior parameters

and (II) retransmission behavior. Documentation of available `procfs` and socket options is provided in the Linux Programmer's Manual pages for UDP, TCP, and SCTP [55–57].

Delay and Coalescence of Writes In certain scenarios, deliberate delay and merge of many small writes to a socket can be advantageous. From a protocol efficiency perspective, sending a large number of packets with a small payload results in bandwidth and processing overhead. A number of related options is given in Table 1. TCP offers multiple parameters to coalesce socket writes. If `tcp_autocorking` is enabled, small socket writes are buffered and coalesced, if at least one packet from the same flow is currently in a qdisc or device transmit flow. Thus, latency of a single segment might increase in case of many small, subsequent writes to the same target. In the context of typical HotStuff operation, this situation can only occur in client-replica communication (**F1**, **F4**), if the client sends multiple small `CMD-REQ` messages to a replica and the replica answers with multiple small `CMD-ANS` messages after a successful commit. In all other flows (§5.1), write calls to the same target depend on previous input (e.g., the next leader / replica message). This request-response communication structure is also present in other BFT-SMR protocols. While a small payload increases relative protocol payload overhead, minimizing latency for `CMD-REQ` and `CMD-ANS` messages (or their equivalents in other BFT-SMR protocols) should be prioritized, if the additional delay of coalescing writes exceeds the processing and transmission time for the protocol overhead.

The `TCP_CORK` option buffers writes until a full frame can be sent, the option is removed or 200 ms have elapsed. While activation conditions are different in that manual corking is possible, usage implications are analogous to `tcp_autocorking`. Both TCP and SCTP offer a respective `NODELAY` option which disables Nagle's Algorithm [58]. The write-merge behavior for Nagle's Algorithm mainly depends on the size of the issued writes. If both data size of the planned write and window size are large enough, the full segment is sent immediately. Partial segments are only sent without buffering and merging, if there is no unacknowledged data in flight. Hence the general behavior and implications upon BFT-SMR protocols are similar to the previous options and are mainly affecting client-replica communication, since the request-response pattern allows for piggybacked ACK in replica-replica communication. TCP also allows to delay and coalesce explicit, non-piggybacked ACKs under special circumstances, involving the receipt of non-full segments and mostly unidirectional communication. While this delay should not be big enough to trigger an RTO, delayed ACKs can interact badly with some configurations. For example, delayed ACKs on receiver side interact badly with active Nagle's algorithm on sender side,

Option	Layer	Effect
`tcp_frto` / `tcp_frto_response`	procfs	Configure forward RTO discovery to prevent unnecessary retransmissions upon spurious RTO
`SCTP_RTOINFO`	Socket	Configure initial, min and max RTO
`TCP_RTO_MIN / MAX`	Kernel	Configure min and max RTO value

Table 2: Retransmission behavior parameters

where the sender delays writes until an ACK is received, but the receiver delays ACKs in expectation of more incoming writes. For applications that require timely ACKs for mostly unidirectional data sent, but do not fill a segment with a single write, delayed ACKs can reduce performance. Reducing the ACK delay may counter this behavior. Delayed ACKs also affects CC behavior (and thus the performance) of the protocol (§6.3). The TCP ACK delay behavior can be configured through multiple configuration parameters. We focus our discussion on three central options here. The socket option `TCP_QUICKACK` allows to toggle Quick ACK mode, in which ACKs are sent immediately. However, this option is not permanent and Quick ACK mode can be left and entered by the kernel, depending on TCP protocol state. Thus, manual steering of Quick ACK mode is non-trivial as it requires knowledge of internal TCP processing state, might interact badly with CC, and requires frequent re-activation. If not operating in Quick ACK mode, TCP delays ACKs to reduce protocol overhead. The effective delay is bounded by the `TCP_DELACK_MIN/MAX` parameters, which are defined to have a minimum value of 40 ms and a maximum value of 200 ms for a kernel interrupt frequency of 1000 Hz per default. In context of BFT-SMR protocols, delayed ACKs do not negatively influence most flows due to their message exchange structure. For practical choices of batch and payload size, no flow from Fig. 2 holds a scenario in which a party issues many writes, smaller than segment size, and requires timely explicit ACKs to continue sending. For small payloads and deactivated batching, ACK delay might need to be optimized to prevent stalling in **F1**, if the BFT-SMR client limits the number of requests in flight (e.g., as in HotStuff) too aggressively.

Retransmission Behavior Reliable transport protocols typically offer a range of options to configure retransmission behavior. A selection of relevant options is given in Table 2. As outlined in §6.1, TCP uses a combination of DupAck and RTO to trigger retransmissions. However, due to sporadic network delay or misconfiguration, the RTO can expire even though a segment arrives. This *spurious* RTO causes unnecessary retransmissions and interacts poorly with TCP congestion control. To counteract this behavior, modern TCP implements Forward RTO-Recovery (F-RTO) [59]. The algorithm tries to detect spurious RTOs based on information from incoming ACKs. If detected, the typical slow start behavior is avoided and operation continues, based on old timeout values and windows. The `tcp_frto` option allows for configuration if and which F-RTO algorithm should be active. If a spurious RTO is detected, the respective response behavior can be configured through the `tcp_frto_response` option. Modern Linux TCP implementations (e.g., Kernel v4.19.194) offer three possible reactions, which halve the congestion window and slow-start threshold after one RTT (0), immediately (1) or not at all (2). Results by Sarolahti [60] suggest that the overall performance of the chosen method depends on the available link bandwidth, where more conservative behavior (0) performs better for low-bandwidth links and aggressive restoration (2) results in more throughput on high-bandwidth links. Hence, configuration for BFT-SMR systems should include F-RTO and be conducted accordingly.

As outlined by Allman et al. [61], configuration of e.g., RTO provides a powerful tool to trade response latency against the risk of premature timeouts and thus unnecessary retransmissions. Both TCP and SCTP offer options to adjust RTO variables. For SCTP, configuration is possible on socket level using the `SCTP_RTOINFO` option. It allows to configure the initial, minimum and maximum RTO value. For TCP, dynamic configuration is not easily possible, since the `TCP_RTO_MIN / MAX` values are hardcoded in the Linux kernel code (§6.1). In context of BFT-SMR systems, the RTO value calculation is mostly relevant in case of message loss. Consensus decision latency in e.g., HotStuff is significantly smaller than the minimum RTO in both SCTP and TCP (cf. measured latencies between 10 and 30 ms for HS3-p1024 [5]). Multiple subsequent omission failures can trigger a RTO and cause a multiple of typical consensus latency in delay. Following the observation from [61], reduction of the minimum RTO reduces latency if omission failures occur and network congestion is less likely in the target network.

6.3 Congestion Control

To prevent overloading of network resources, both TCP and SCTP implement CC. In this section, we briefly analyze the impact and implications of CC on BFT-SMR operation. Since CC in SCTP closely follows algorithms and rationale from TCP, we limit our explicit discussion to TCP here. First we generally discuss CC algorithms in the context of BFT-SMR algorithms and afterward analyze a representative CC algorithm in operation with HotStuff.

General Discussion The response to network congestion is typically a reduction of throughput on the strained links. While load reduction in response to an actual congestion event is helpful (i.e., in order to prevent congestive collapse [62]), performance degradation due to negative interaction with typical BFT-SMR operation should be avoided. Detection of congestion events and response are defined by the CC algorithm in effect. The approaches mainly differ in terms of congestion detection method and target network scenario. As outlined by Afanasyev et al. [63], optimization of CC revolves around balancing (1) throughput against network load and (2) loss recovery latency against unnecessary retransmissions. As concrete constraints vary with the target environment, there is no one-size-fits-all algorithm that performs best in all scenarios. The main difference between CC in context of general BFT-SMR operation and CC in commodity use cases, are assumptions on the expected behavior of participating peers. In BFT-SMR scenarios, malicious participants are explicitly part of the model. In the design process of transport protocols, basic security concerns are usually addressed (e.g., defense against sequence number attacks [64] or SYN flooding [65] in TCP) but protection against stronger adversaries is typically assumed to be handled by another layer. For example, the SCTP specification suggests employing IPSec [66] to provide extended confidentiality and integrity guarantees [48].

Hence, CC in a BFT-SMR setting is potentially subject to stronger adversarial behavior and may require additional protective measures. We consider a detailed evaluation of the ecosystem of CC algorithms in BFT-SMR context to be out of the scope of this paper and defer this task to future work.

HotStuff Case-Study In §5.1.1 we identified client and leader nodes as bottleneck peers. Hence, impact of changes to CC state is most relevant on these nodes. Independent of the physical interconnections between replicas and clients, links and network segments of bottleneck nodes are expected to experience most strain and contribute most to potential network congestion. Generally, CC measures are employed on a flow base between two communicating nodes. The CC variables are influenced by (1) underlying network events and (2) communication partner behavior, which may include the same events but for different reasons (e.g., compromise).

As outlined in §2, the conceptual end-to-end connections between all nodes are usually not reflected in hardware. Thus, due to the physical network architecture, a third party could theoretically influence the CC state of two communication partners. For the case study, we choose TCP Cubic [67] in Linux v4.19.194 as representative CC algorithm, since it is the default used by the Linux Kernel since version `2.6.19` (2006) [68] and by Windows 10 since 2017 [69]. TCP Cubic modifies the CC variables based on a cubic function of the elapsed time since the last congestion event. The detection of a "congestion event" is based on packet loss. The sender enters the loss state if sent data is acknowledged too late or not at all. Note that this loss detection is also subject to F-RTO (§6.2).

As actual network congestion should always trigger a congestion event eventually, we discuss if regular BFT-SMR operation can also trigger "spurious" congestion events and thus reduce performance. First consider ACKs that arrive too late. A TCP receiver can piggyback ACKs for received data onto its own replies with payloads. As long as bidirectional data is regularly and successfully delivered, ACKs can be attached to this communication. A significant amount of communication in HotStuff involves an alternating request-response pattern, as given through flows **F2** and **F3**. If communication is mostly unidirectional (e.g., the receiver does not directly send back reply payloads, such as in **F1**), explicit ACKs are generated. RFC5681 [70] specifies that ACKs should be generated for at least every second full-sized segment and ACKs must be generated within 500 ms of the arrival of the first unacknowledged packet. In any case, as long as an ACK arrives before the RTO is triggered, no congestion event is detected and throughput is not reduced. Now consider cases where ACKs are received out-of-order or not at all. Both cases are not caused by regular BFT-SMR protocol operation but by either (1) underlying network effects or (2) malicious activity. To reduce the congestion window of another node an attacker would need to modify or drop traffic between two target victim nodes on hardware links. While integrity protection of transport layer information could be realized by employing a secure channel (e.g., IPSec [66]) on Layer 3, this step adds even more bandwidth and processing overhead.

7 Conclusion and Future Work

We studied the impact of network transport protocols on BFT-SMR systems in general and HotStuff, in particular. Due to the prevalence of TCP as underlying networking transport stack, our results also bear applicability to all BFT-SMR setups with the same building blocks. In our theoretical study of HotStuff communication, we identified leader and client nodes as potential operation bottlenecks. We demonstrate that unmasked message loss potentially violates protocol safety, but at least incurs performance overhead. In our analysis of the transport protocol configuration space, we identify a set of preferable configuration options for operation of BFT-SMR systems. Further optimization of parameters is possible and dependent on the target networking environment. We identify no general negative reciprocity between CC and BFT-SMR algorithms in general, and HotStuff and TCPCubic in particular. In future work we aim to validate the presented results by implementation and measurement under realistic conditions. Furthermore, we aim to investigate optimization of other BFT-SMR building blocks such as message authentication mechanisms. We note that further study of the extended attacker model of byzantine behavior in context of CC algorithms for use in BFT-SMR setups may yield helpful results for the implementation of practical systems.

References

[1] M. Pease, R. Shostak, and L. Lamport, “Reaching Agreement in the Presence of Faults,” *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[2] L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[3] F. B. Schneider, “Implementing Fault-Tolerant Services Using the State Machine Approach: A tutorial,” *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[4] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, “The Honey Badger of BFT Protocols,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 31–42.

[5] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, “HotStuff: BFT Consensus with Linearity and Responsiveness,” in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing.* ACM, 2019, pp. 347–356.

[6] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, “SBFT: A Scalable and Decentralized Trust Infrastructure,” in *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN).* IEEE, 2019, pp. 568–580.

[7] C. Stathakopoulou, D. Tudor, M. Pavlovic, and M. Vukolić, “Mir-BFT: Scalable and Robust BFT for Decentralized Networks,” *Journal of Systems Research*, vol. 2, no. 1, 2022.

[8] N. Chondros, K. Kokordelis, and M. Roussopoulos, “On the Practicality of ’Practical’ Byzantine Fault Tolerance,” in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing.* Springer, 2012, pp. 436–455.

[9] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzzyva: Speculative Byzantine Fault Tolerance,” in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 45–58.

[10] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, “The Next 700 BFT Protocols,” in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 363–376.

[11] X. Défago, A. Schiper, and P. Urbán, “Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey,” *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 372–421, 2004.

[12] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, “BFT Protocols Under Fire.” in *NSDI*, vol. 8, 2008, pp. 189–204.

[13] J. Neu, S. Sridhar, L. Yang, D. Tse, and M. Alizadeh, "Longest Chain Consensus Under Bandwidth Constraint," Cryptology ePrint Archive, Paper 2021/1545, 2021, https://eprint.iacr.org/2021/1545. [Online]. Available: https://eprint.iacr.org/2021/1545

[14] S. Coretti, A. Kiayias, C. Moore, and A. Russell, "The Generals' Scuttlebutt: Byzantine-Resilient Gossip Protocols," *Cryptology ePrint Archive*, 2022.

[15] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary," in *2009 28th IEEE International Symposium on Reliable Distributed Systems*. IEEE, 2009, pp. 135–144.

[16] A. Bessani, J. Sousa, and E. E. Alchieri, "State Machine Replication for the Masses with BFT-SMART," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 355–362.

[17] I. Abraham, G. Gueta, D. Malkhi, and J.-P. Martin, "Revisiting Fast Practical Byzantine Fault Tolerance: Thelma, Velma, and Zelma," *arXiv preprint arXiv:1801.10022*, 2018.

[18] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync Hotstuff: Simple and Practical Synchronous State Machine Replication," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 106–118.

[19] Z. Avarikioti, L. Heimbach, R. Schmid, and R. Wattenhofer, "FnF-BFT: Exploring Performance Limits of BFT Protocols," *arXiv preprint arXiv:2009.02235*, 2020.

[20] B. Y. Chan and E. Shi, "Streamlet: Textbook Streamlined Blockchains," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020, pp. 1–11.

[21] S. Gupta, S. Rahnama, and M. Sadoghi, "Permissioned Blockchain Through the Looking Glass: Architectural and Implementation Lessons Learned," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2020, pp. 754–764.

[22] J. Niu and C. Feng, "Leaderless Byzantine Fault Tolerant Consensus," *arXiv preprint arXiv:2012.01636*, 2020.

[23] O. Naor and I. Keidar, "Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR," in *34th International Symposium on Distributed Computing*, 2020.

[24] P. Kuznetsov, A. Tonkikh, and Y. X. Zhang, "Revisiting Optimal Resilience of Fast Byzantine Consensus," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 343–353.

[25] M. Castro, B. Liskov *et al.*, "Practical Byzantine Fault Tolerance," in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[26] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, "RBFT: Redundant Byzantine Fault Tolerance," in *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 2013, pp. 297–306.

[27] J.-P. Martin and L. Alvisi, "Fast Byzantine Consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, 2006.

[28] S. Alqahtani and M. Demirbas, "BigBFT: A Multileader Byzantine Fault Tolerance Protocol for High Throughput," in *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)*. IEEE, 2021, pp. 1–10.

[29] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "UpRight Cluster Services," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 277–290.

[30] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "NetChain: Scale-Free Sub-RTT Coordination," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 35–49.

[31] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading Distributed Applications onto SmartNICs using iPipe," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 318–333.

[32] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé, "P4xos: Consensus as a Network Service," *IEEE/ACM Transactions on Networking*, vol. 28, no. 4, pp. 1726–1738, 2020.

[33] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui, "APUS: Fast and Scalable PAXOS on RDMA," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 94–107.

[34] S. Rüsch, I. Messadi, and R. Kapitza, "Towards Low-Latency Byzantine Agreement Protocols Using RDMA," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 146–151.

[35] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults," in *NSDI*, vol. 9, 2009, pp. 153–168.

[36] T. Lorünser, B. Rainer, and F. Wohner, "Towards a Performance Model for Byzantine Fault Tolerant Services," in *CLOSER*, 2022, pp. 178–189.

[37] I. Moraru, D. G. Andersen, and M. Kaminsky, "There Is More Consensus in Egalitarian Parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 358–372.

[38] M. Bravo, Z. István, and M.-K. Sit, "Towards Improving the Performance of BFT Consensus For Future Permissioned Blockchains," *arXiv preprint arXiv:2007.12637*, 2020.

[39] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. J. Marathe, A. Xygkis, and I. Zablotchi, "Microsecond Consensus for Microsecond Applications," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 599–616.

[40] M. K. Reiter, "Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart," in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994, pp. 68–80.

[41] D. Malkhi and M. Reiter, “A High-Throughput Secure Reliable Multicast Protocol,” *Journal of Computer Security*, vol. 5, no. 2, pp. 113–127, 1997.

[42] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, “Scalable and Probabilistic Leaderless BFT Consensus through Metastability,” *arXiv preprint arXiv:1906.08936*, 2019.

[43] M. Castro and B. Liskov, “Practical Byzantine Fault Tolerance and Proactive Recovery,” *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

[44] S. Alqahtani and M. Demirbas, “Bottlenecks in Blockchain Consensus Protocols,” in *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*. IEEE, 2021, pp. 1–8.

[45] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino, “State Machine Replication in the Libra Blockchain,” *The Libra Assn., Tech. Rep*, 2019.

[46] J. Postel, “User Datagram Protocol,” RFC 768 (INTERNET STANDARD), Internet Engineering Task Force, Aug. 1980. [Online]. Available: http://www.ietf.org/rfc/rfc768.txt

[47] ——, “Transmission Control Protocol,” RFC 793 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1122, 3168, 6093, 6528. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

[48] R. Stewart, “Stream Control Transmission Protocol,” RFC 4960 (Proposed Standard), Internet Engineering Task Force, Sep. 2007, updated by RFCs 6096, 6335. [Online]. Available: http://www.ietf.org/rfc/rfc4960.txt

[49] C. A. Kent and J. C. Mogul, “Fragmentation Considered Harmful,” *ACM SIGCOMM Computer Communication Review*, vol. 25, no. 1, pp. 75–87, 1995.

[50] J. C. Mogul and C. A. Kantarjiev, “Retrospective on Fragmentation Considered Harmful,” *ACM SIGCOMM Computer Communication Review*, vol. 49, no. 5, pp. 41–43, 2019.

[51] K. Kursawe and V. Shoup, “Optimistic Asynchronous Atomic Broadcast,” in *International Colloquium on Automata, Languages, and Programming*. Springer, 2005, pp. 204–215.

[52] M. Allman, V. Paxson, and W. Stevens, “TCP Congestion Control,” IETF, RFC 2581, Apr. 1999. [Online]. Available: http://tools.ietf.org/rfc/rfc2581.txt

[53] V. Paxson, M. Allman, J. Chu, and M. Sargent, “Computing TCP’s Retransmission Timer,” IETF, RFC 6298, Jun. 2011. [Online]. Available: http://tools.ietf.org/rfc/rfc6298.txt

[54] P. P. Mishra, D. Sanghi, and S. K. Tripathi, “TCP Flow Control in Lossy Networks: Analysis and Enhancement,” in *NETWORKS*. Citeseer, 1992, pp. 181–192.

[55] *UDP(7), Linux Programmer’s Manual*, 5th ed., 03 2021.

[56] *TCP(7), Linux Programmer’s Manual*, 5th ed., 03 2021.

[57] S. Samudrala, *SCTP(7), Linux Programmer's Manual*, 10 2005.

[58] J. Nagle, "Congestion Control in IP/TCP Internetworks," IETF, RFC 896, Jan. 1984. [Online]. Available: http://tools.ietf.org/rfc/rfc0896.txt

[59] P. Sarolahti, M. Kojo, K. Yamamoto, and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP," IETF, RFC 5682, Sep. 2009. [Online]. Available: http://tools.ietf.org/rfc/rfc5682.txt

[60] P. Sarolahti, "Congestion Control on Spurious TCP Retransmission Timeouts," in *GLOBECOM'03. IEEE Global Telecommunications Conference (IEEE Cat. No. 03CH37489)*, vol. 2. IEEE, 2003, pp. 682–686.

[61] M. Allman and V. Paxson, "On Estimating End-to-End Network Path Properties," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4, pp. 263–274, 1999.

[62] M. Gerla and L. Kleinrock, "Flow Control: A Comparative Survey," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 553–574, 1980.

[63] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock, "Host-to-Host Congestion Control for TCP," *IEEE Communications surveys & tutorials*, vol. 12, no. 3, pp. 304–342, 2010.

[64] F. Gont and S. Bellovin, "Defending against Sequence Number Attacks," IETF, RFC 6528, Feb. 2012. [Online]. Available: http://tools.ietf.org/rfc/rfc6528.txt

[65] W. Eddy, "TCP SYN Flooding Attacks and Common Mitigations," IETF, RFC 4987, Aug. 2007. [Online]. Available: http://tools.ietf.org/rfc/rfc4987.txt

[66] S. Kent and K. Seo, "Security Architecture for the Internet Protocol," IETF, RFC 4301, Dec. 2005. [Online]. Available: http://tools.ietf.org/rfc/rfc4301.txt

[67] S. Ha, I. Rhee, and L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant," *ACM SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, 2008.

[68] S. Hemminger and D. Miller, "[TCP]: make cubic the default," https://github.com/torvalds/linux/commit/597811ec167fa01c926a0957a91d9e39baa30e64, 2006.

[69] P. Balasubramanian, "Updates on Windows TCP," https://datatracker.ietf.org/meeting/100/materials/slides-100-tcpm-updates-on-windows-tcp, 2017, last-accessed: 2022-06-20.

[70] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," IETF, RFC 5681, Sep. 2009. [Online]. Available: http://tools.ietf.org/rfc/rfc5681.txt