Scalable Byzantine Fault Tolerance via Partial Decentralization

Balaji Arun
Virginia Tech
balajia@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

ABSTRACT

Byzantine consensus is a critical component in many *permissioned* Blockchains and distributed ledgers. We propose a new paradigm for designing BFT protocols called DQBFT that addresses three major performance and scalability challenges that plague past protocols: (i) high communication costs to reach geo-distributed agreement, (ii) uneven resource utilization hampering performance, and (iii) performance degradation under varying node and network conditions and high-contention workloads. Specifically, DQBFT divides consensus into two parts: 1) durable command replication without a global order, and 2) consistent global ordering of commands across all replicas. DQBFT achieves this by decentralizing the heavy task of replicating commands while centralizing the ordering process.

Under the new paradigm, we develop a new protocol, Destiny that uses a combination of three techniques to achieve high performance and scalability: using a trusted subsystem to decrease consensus's quorum size, using threshold signatures to attain linear communication costs, reducing client communication. Our evaluations on 300-replica geo-distributed deployment reveal that DQBFT protocols achieve significant performance gains over prior art: ≈3x better throughput and ≈50% better latency.

PVLDB Reference Format:
Balaji Arun and Binoy Ravindran. Scalable Byzantine Fault Tolerance via Partial Decentralization. PVLDB, 15(9): 1739 - 1752, 2022.
doi:10.14778/3538598.3538599

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

1 INTRODUCTION

Byzantine consensus protocols are a perfect fit for solving the agreement problem in *consortium* Blockchain platforms [7] due to their ability to shield the system from known but potentially mistrustful participants while reaching consensus efficiently, as opposed to Proof-of-Work-based [47] techniques. The fundamental requirement of any Blockchain platform is scalability to hundreds of nodes deployed around the world. Traditional Byzantine Fault-Tolerant (BFT) consensus protocols [17, 18, 23, 35, 36, 48] suffer from intrinsic design issues that inhibit their scalability in geographically distributed (geo-distributed) deployments.

Most deterministic BFT consensus protocols [17, 23, 26, 35] adopt the primary-backup approach, where a designated primary replica is responsible for ordering and replicating the client-submitted commands among the replicas. Relying on a dedicated replica to perform both these operations is detrimental to performance, especially at scale. In particular, such an approach causes a) load imbalance among primary and backup replicas, because the primary sends larger messages containing client commands, while backups send small state messages; b) under utilization of resources at backup replicas, because the primary saturates its network resources before the replicas, diminishing their individual potential; c) remote clients to pay high WAN latencies by sending requests to the primary than clients that are local to the primary; and d) poor tolerance to primary failures [29]. Client commands in Blockchain applications (e.g. smart contracts) are large in the order of kilobytes [3, 7]. This limits the number of commands that a primary can multicast with its bandwidth to *all* replicas.

Existing BFT solutions that overcome these downsides of the primary-backup approach have drawbacks. Specifically, the rotating primary [53, 54, 58] and multi-primary [29, 50] approaches do not take into account many aspects of modern geographically distributed systems including variations in node hardware, network bandwidth, and available resources. In such settings, a slow node can quickly degrade the overall performance. Some decentralized approaches [8, 28] exploit the commutativity of client commands and track dependencies to order conflicting commands. This requires additional coordination to process concurrent conflicting commands degrading performance.

Towards Partial Decentralization. To overcome these drawbacks, we present DQBFT (for Divide and conQuer BFT), a paradigm for designing highly scalable consensus protocols by partially decentralizing the core consensus process into two distinct and concurrent steps that may be handled at potentially different replicas. Rather than adopting a completely decentralized approach where individual replicas replicate commands and also coordinate to find a *total order*, DQBFT divides the task of consensus into two: 1) durable replication of client commands without a global order at correct replicas, and 2) ordering of the commands to guarantee a *total order*. Durable replication is carried out by each individual replica for the commands it receives from clients, while ordering is performed by a dedicated sequencer. Ordering involves assigning a *global* order to a replica that has proposed a command. Thus, unlike the rotating primary and other multi-primary techniques [29, 50, 53, 58], our approach can seamlessly accommodate variations in node hardware, network bandwidth, and available resources.

The DQBFT approach is unique in that it allows for concurrent progression of the two stages in the absence of failures. DQBFT uses separate instances of consensus protocols at individual replicas to carry out replication providing load balancing of client commands among the replicas, while another consensus protocol is responsible for assigning the global order to individual replicas. This simultaneous replication and ordering allows DQBFT to avoid the latency

Proceedings of the VLDB Endowment, Vol. 15, No. 9 ISSN 2150-8097.
doi:10.14778/3538598.3538599

penalties due to the additional communication steps. However, to limit the impact of Byzantine replicas in certain situations, DQBFT requires that replication precede global ordering on a per-replica basis. Decoupling replication from ordering has been proposed in the crash-fault model [38, 46, 60], but, these protocols do not scale to hundreds of geo-distributed Byzantine replicas, require special network hardware, and/or are not oblivious to conflicts.

Towards Highly Scalable Consensus. While the DQBFT paradigm can be adopted into existing BFT protocols, we show, analytically in Figure 3 and empirically in Section 5, that such instantiations do not scale their performance to hundreds of replicas. Therefore, we present Destiny, the flagship instantiation of the DQBFT paradigm with three enhancements each of which contribute to achieve high performance while scaling to hundreds of replicas. Briefly, the techniques include: (1) using a hardware-assisted trusted subsystem to increase fault-tolerance and decrease quorum sizes; (2) linear communication for scalability; and (3) using threshold cryptography for optimal linear communication.

BFT protocols require $3f + 1$ replicas and three communication steps among two-thirds of replicas to reach agreement. In contrast, Hybrid consensus protocols [13, 55] use trusted subsystems to require only $2f + 1$ replicas and two communication steps among *majority* replicas to reach agreement. We show that such efficiency combined with the reduction in the number of messages exchanged per commit via linear communication patterns is key in leveraging the benefits of the DQBFT paradigm at scale (see Figures 2 and 3). With this insight, we adopt and linearize the common-case communication of a recent Hybrid protocol, Hybster [13], producing Linear Hybster. Both the replication and ordering steps of Destiny use instances of *Linear Hybster*.

The ability of Hybrid protocols to tolerate more faults and use smaller size quorums enable scalability in geo-distributed environments. Further, trusted execution environments are now available at commodity-scale (e.g., Intel SGX [22], ARM's TrustZone [39]), making Hybrid protocols more feasible. Regardless, the DQBFT paradigm is generally applicable to any BFT protocol and does not require Hybrid fault assumptions. Destiny leverages DQBFT and the Hybrid model to improve performance. Many Blockchain solutions already depend on trusted execution environments for privacy-focused computations [1, 59], and thus, can easily take advantage of the added performance provided by Hybrid protocols.

Contributions. Section 2 discusses the differences between BFT and Hybrid protocols and the challenges existing in the landscape. Section 3 proposes DQBFT, a paradigm for designing scalable BFT protocols by partially decentralizing the replication and ordering concerns. The technique can be applied to most primary-backup protocols to achieve high performance and scalability. Section 4 proposes Destiny, a Hybrid protocol and the flagship instantiation of the DQBFT paradigm that scales to hundreds of geo-distributed replicas. Section 5 presents a comprehensive evaluation of the state-of-the-art protocols and four DQBFT protocols, including Destiny, in a geo-distributed deployment with various system sizes ranging from 19 up to 301 replicas, withstanding between $f = 6$ and 150 Byzantine failures. Our evaluations reveal that the DQBFT variants of PBFT [17], SBFT [26], and Hybster [13] – DQPBFT, DQSBFT, and DQHybster – outperform their vanilla counterparts with up to an order of magnitude better performance. Furthermore, these protocols tolerate lagging replicas better than other multi-primary protocols with at least 20% better throughput. Destiny provides 40% better throughput than DQSBFT and up to 70% lower latency than any other state-of-the-art protocol.

2 BACKGROUND

In this section, we provide the necessary background for understanding the rest of the paper.

2.1 Byzantine Consensus

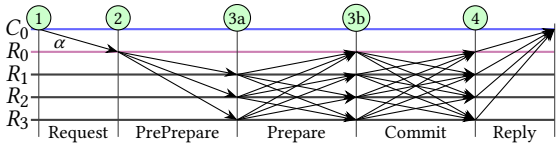
A Byzantine Fault-Tolerant (BFT) protocol consists of a set of replicas that agree on the order of client-issued commands and execute them in the agreed order. The protocol proceeds in a series of views, where in each view, a *primary* replica proposes and sequences commands, which are executed by all non-faulty replicas in the prescribed order. Before executing the commands, correct replicas must ensure that (i) the commands are replicated at enough correct replicas and (ii) enough correct replicas observe the same sequence of commands from the primary. This function is carried out by the *Agreement* algorithm, by exchanging command and state information between replicas. Some protocols (e.g., PBFT [17]) commit in three phases and require consent from a supermajority (i.e., 67%) of replicas, while some others (e.g., SBFT [26]) require consent from all replicas and commit in two phases during non-faulty periods.

When the primary ceases to make timely progress or misbehaves by sending different sequence of commands to different replicas, the *View Change* algorithm is invoked by non-faulty replicas to replace the faulty primary. The primary of the new view, determined by the view number, collects the replica-local states of enough replicas, computes the initial state of the new view, and proceeds with the agreement algorithm in the new view. If a view change does not complete in time, another one is triggered for the next primary.

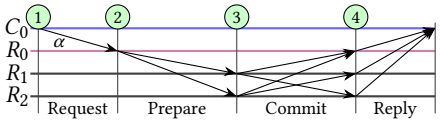
Replicas use the *Checkpoint algorithm* to limit their memory requirements by garbage collecting the states for those commands that have been executed at enough correct replicas. Replicas exchange information to produce the checkpoint state. When some replicas fall behind the rest of the system, the checkpoint state is used to bring them up to date via the *state transfer* algorithm.

Consensus with Trusted Subsystems. Replicas in the BFT model may fail to send one or more messages specified by the protocol or even send messages not specified by the protocol. These replicas can also equivocate, i.e., make conflicting statements to compromise consistency, without being detected. To tolerate such behaviors, BFT protocols require supermajority *quorums* — the subset of replicas that is used to make decisions at different phases of consensus — in asynchronous systems.

The hybrid fault model, in contrast, uses a trusted subsystem to prevent replicas from equivocating [19, 37]. The trusted subsystem is a local service that exists at every replica, and certifies the messages sent by the replicas to ensure that malicious replicas cannot cause different correct replicas to execute different sequences of commands. The trusted subsystem, typically, consists of a monotonically increasing counter that is paired with an attestation mechanism (signatures/message authentication codes). It assigns a unique counter to a message and generates a cryptographic attestation over



(a) PBFT



(b) Hybster, hybrid protocol

Figure 1: Agreement protocol steps: PBFT and Hybster.

the pair. Thus, each outbound message is bound to a unique counter value. When correct replicas receive the message pairs, they process them in increasing counter order. Thus, when a faulty replica sends two different messages to two correct replicas, only one will process the message, while the other will wait for the message with the missing counter value, eventually detecting equivocation.

Since equivocation is prevented using the trusted subsystem, f additional correct replicas that were required for traditional BFT protocols to balance the impact of f malicious replicas are no longer required in the hybrid fault model. The result is smaller quorums. The system size (N) of traditional BFT protocols is $3f + 1$; hybrid protocols improve this to $2f + 1$.

Example 2.1. BFT vs Hybrids. To illustrate the fundamental design differences between BFT and hybrid protocols, we compare PBFT (a BFT protocol) and Hybster [13] (a hybrid protocol). The agreement algorithm of PBFT and Hybster operates in three and two phases, respectively, as illustrated in Figure 1. We describe the two protocols hand-in-hand and only highlight their differences.

(1) The execution starts when the client sends a command to the primary replica. The client signs its command to ensure that a malicious replica cannot tamper the command without detection. (2) The primary receives the command and proposes it to all replicas with a sequence number. The sequence number defines the *global* execution order with respect to other commands. The message is certified using a *message authentication code* (MAC). Hybster uses the trusted subsystem to produce the MAC. The counter value of Hybster maps to the sequence number assigned to the command. Thus, two different commands are never assigned the same sequence number in Hybster. Also, note that in the example, Hybster requires three nodes, while PBFT requires four.

The replicas receive the proposal from the primary. (3) **Hybster** replicas acknowledge the proposal to each other and wait for a majority of responses to commit and execute the command. (3a) **PBFT** replicas exchange the proposal with each other to ensure that they received the same proposal from the primary (i.e., to ensure no equivocation). Note that Hybster avoids this step using the trusted subsystem. The proposal is validated if a supermajority of nodes respond with the same proposal from the primary. (3b) **PBFT** replicas exchange commit messages. They execute the command upon collecting a supermajority quorum of these messages and reply to the client. At the end of this step, in both the protocols, a correct replica is able to recover the command, even if f replicas fail including the primary. (4) Clients wait until they receive identical replies from at least $f + 1$ replicas. This is because, waiting for only one potentially malicious replica may yield an incorrect result.

2.2 Decentralizing Consensus

A major problem with quorum-based BFT consensus protocols [17, 26, 58] that underpin numerous Blockchain infrastructures [14] is their reliance on a designated primary replica to order client commands. The maximum theoretical throughput at which a primary can replicate client commands is $T_p = B/((N-1)pm)$, where B is the primary's network bandwidth and pm is the size of payload message and N is the number of replicas [29]. However, to ensure safety, replicas exchange state messages with each other and these messages must be taken into account. Figure 2 summarizes the theoretical throughput equations for protocols including PBFT and Hybster, and Figure 3 plots them for two payload sizes. Note that this throughput is calculated based on replica bandwidth only; in practice, the throughput is also affected by available computation and memory resources. The primary sends $(N-1)pm$ bytes, while other replicas only receive roughly pm bytes each, causing load imbalance and underutilization of replica resources. By distributing the primary's responsibility and allowing all replicas to replicate the command payloads concurrently, one can achieve maximum throughput $T_{cmax} = (NF \cdot B)/((N-1)pm + (NF-1)pm)$, where $NF = N - f$ is the number of non-faulty replicas. The literature presents multiple methodologies to accomplish this.

In the first approach, referred to as *static* ordering, the sequence numbers used to order the commands are statically partitioned among replicas. Replicas use their allocated set of sequence numbers to propose and commit commands, either in parallel [29] or in round-robin fashion [58]. To ensure linearizability [32], replicas must execute commands in the order of their sequence numbers. Such an approach cannot adapt to variations in node hardware and network bandwidth. A slow replica can throttle the system performance as commands must be effectively executed at the speed at which the slowest replica can propose and commit commands. Examples of protocols that adopt variants of this approach include Hotstuff [58], RCC [29], MirBFT [50], and Dispel [57].

In the second approach, referred to as *dependency*-based ordering, replicas commit commands after exchanging dependency metadata, and execute those commands in a deterministic order satisfying the dependency information. Commands with conflicting operations are totally ordered while others are partially ordered [24]. Such dynamic ordering minimizes the overhead of ordering non-conflicting commands, because their reordering does not cause inconsistent system state. Such protocols [9, 45] incur higher overhead when the number of conflicting commands is high, degrading performance.

Tree-based dissemination mechanisms [48] have been shown to alleviate the primary's load, but these require additional latency-inducing steps equivalent to the height of the dissemination trees.

3 THE DIVIDE AND CONQUER PARADIGM

We propose DQBFT, a paradigm for building high-performance BFT protocols that overcomes the aforementioned challenges in existing protocols. To do so, DQBFT decentralizes the responsibility of the primary based on the two important actions performed by a

Protocol	Messages	Throughput	Phases
PBFT	$N + 2N^2$	$B/(N-1)(pm + 3sm)$	3
Hybster	$N + N^2$	$B/(N-1)(pm + sm)$	2
DQPBFT (ours)	$2N + 4N^2$	$\frac{NF * B}{(N-1)(pm+3sm) + (NF-1)(pm+4(N-1)sm) + (N-1)(4sm)}$	4 or 6
Destiny (ours)	$7N$	$\frac{NF * B}{(N-1)(pm+3sm) + (NF-1)(pm+3sm) + (N-1)(4sm)}$	5 or 7

Figure 2: Comparison of single-primary and DQBFT protocols. B: bandwidth per replica; N: system size; NF: non-faulty replicas; pm: size of payload messages; sm: size of state messages.

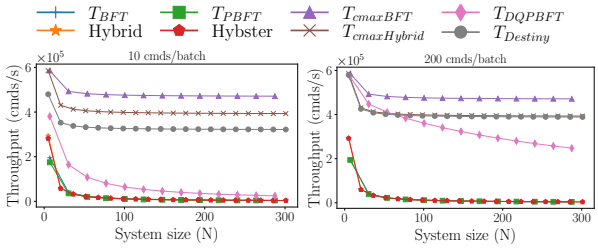


Figure 3: Maximum theoretical throughput in a system with $B = 1Gbit/s$, $NF = N - f$, $sm = 250B$, and $pm = 5KiB$ (left) and $100KiB$ (right) respectively.

consensus protocol: i) request dissemination with partial ordering and ii) global ordering. Request dissemination is a decentralized operation and does not require replicas to coordinate, but only acknowledge receipt. In contrast, global ordering requires replicas to coordinate to ensure that the system has a single view of the sequence of operations. To simplify this process, a primary is chosen to propose the global ordering for the commands.

Under the DQBFT paradigm, clients can send commands to any replica. Replicas can individually disseminate and order client commands using multiple instances of a consensus protocol. While this ensures that the commands are disseminated to the correct replicas, the order produced is local to the replica, i.e. *a partial order*, and not global. The *primary* replica produces a global order among the partial orders produced by the individual replicas.

Such a *partially-decentralized* approach has the following benefits. First, the decentralized process of dissemination distributes load evenly across replicas and enables clients to connect to the nearest replica in geo-distributed deployments. Second, for ordering, the global view of all commands enables the sequencer to order them optimally, i.e., each newly proposed command can be dynamically assigned to the first unused global sequence number. Unlike other multi-primary [29, 50] and rotating-leader [53, 54, 58] protocols whose performances suffer due to slow replicas, our technique allows replicas to execute commands at their own pace without being bottle-necked by slower replicas. Moreover, such an approach is oblivious to conflicts (unlike, for e.g., ezBFT [8], Aliph [28]).

3.1 Design

At a high level, the DQBFT paradigm is composed of two sub-protocols: the dissemination protocol and the global ordering protocol. The dissemination protocol employs multiple instances of consensus, called D-instances, to enable every replica to disseminate and partially order its client commands. Meanwhile, the global ordering protocol uses a single instance of consensus, called the O-instance, that agrees on the global order among the partial orders produced by the D-instances. There are as many D-instances as there are replicas, and every replica is the coordinator of at least one D-instance. A replica proposes commands in a series of sequence numbers belonging to its own D-instance to produce its partial order. The primary proposes D-instance sequence numbers in O-instance's sequence number space to effectively produce a global order from the replica-specific partial orders.

To tolerate Byzantine faults, both dissemination and ordering should be handled by a BFT consensus protocol. Primary-based protocols with the following properties [55] can be used for instantiating protocols under the DQBFT paradigm.

(P1) If a correct replica executes a command α at sequence number $\mathcal{S}$ in view v, then no correct replica will execute $\alpha' \neq \alpha$ at $\mathcal{S}$.

(P2) If a correct replica executes a command α at sequence number $\mathcal{S}$ in view v, no correct replica will execute α with sequence number $\mathcal{S}' > \mathcal{S}$ in any view $v' > v$.

(P3) During a stable view where the communication between correct replicas is synchronous, a proposed client command is committed by correct replicas.

(P4) A view v will eventually transition to a new view $v' > v$ if enough correct replicas request for it.

Many primary-based BFT and Hybrid protocols provide these properties and can be instantiated under the DQBFT paradigm [13, 18, 26, 35]. In Section 5, we evaluate four such instantiations.

Using consensus protocols for both D-instances and the O-instance allows the dissemination and the global ordering steps to proceed concurrently. While dissemination is in progress, the ordering protocol *optimistically* proposes a global ordering for the command. This allows for the communication steps of both protocols to overlap, and thereby effectively reduces the overall number of communication steps. Note that such concurrent processing does not strain the network. Since the ordering protocol is only ordering the sequence numbers, the message sizes are constant and only a few bytes. The dissemination protocol carries a larger and variable payload containing the client commands. See Figure 2 for a comparison of DQBFT protocols and Figure 3 for theoretical throughput analysis.

3.2 DQBFT

In this section, we describe the DQBFT paradigm in detail and show how it accomplishes its goal of decentralizing the dissemination and global ordering steps, and prevents slow replicas from bottlenecking the system performance. For the sake of exposition, we describe DQBFT by applying it to PBFT. Figure 4 illustrates DQBFT's separation of dissemination and ordering steps.

3.2.1 Agreement Protocol. Figure 5 presents the agreement protocol. We assume that the O-instance primary is elected beforehand. A replica R_n that receives the client command, say α, becomes the command's initial *coordinator*. We say initial, because when the coordinator fails, it will be replaced by the View Change procedure. The coordinator is responsible for partial ordering α with respect to

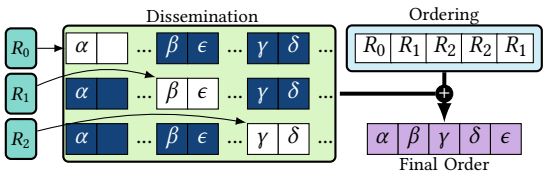


Figure 4: DQBFT's dissemination and ordering steps.

the commands previously coordinated by it. The coordinator uses its D-instance protocol and assigns a sequence number $\mathcal{S}_{ni}$ and runs the consensus protocol to disseminate the command to other replicas. Concurrently, the O-instance primary optimistically globally orders the D-instance sequence number $\mathcal{S}_{ni}$. The O-instance primary uses the PrePrepare message sent by the D-instance primary as the request for finding the global order for $\mathcal{S}_{ni}$. Note that the O-instance protocol only orders the D-instance sequence numbers, only and not the commands. The O-instance primary R_p assigns a sequence number $\mathcal{S}_{pk}$ to D-instance number $\mathcal{S}_{ni}$ and runs the consensus protocol to produce a global order for α.

3.2.2 Execution. A command α proposed by replica R_n is *decided* at a replica when it has been committed under a D-instance sequence number $\mathcal{S}_{ni}$, and $\mathcal{S}_{ni}$ has been committed under a O-instance sequence number $\mathcal{S}_{pk}$. However, the command cannot be executed yet. The command is considered *ready* for execution only after all the corresponding commands mapped to the O-instance sequence numbers up to $\mathcal{S}_{pk}$ have been committed and executed. Replicas execute the command and respond to the client.

3.2.3 Checkpoint and State-transfer Protocols. As described previously, consensus protocols use the checkpoint mechanism to reduce the memory footprint at the replicas by garbage collecting logs for previously executed commands. This procedure also aids in bringing any lagging replicas to the latest state, by allowing up-to-date replicas to exchange the checkpoint data and recent logs via the state-transfer protocol. Note that replicas can be lagging as a result of the primary's intentions. This has implications for DQBFT.

Example 3.1. Consider a DQBFT instantiation of PBFT with $N = 4$ ($f = 1$) replicas. Let R_0 be the O-instance primary and be Byzantine. Let R_1 and R_2 use their D-instances to commit two commands, say, α and β, by sending them only to quorum replicas R_0, R_1, and R_2. Let R_0, the O-instance primary, order the D-instances using quorum replicas R_0, R_1, and R_3. Now, R_1 is the only correct replica that can execute both the commands and respond to the client. Neither R_2 nor R_3 have all the necessary O-instance and D-instance messages, respectively, to execute the commands. Thus, these replicas must transfer up-to-date state from other correct replicas before execution.

We now discuss measures to prevent malicious replicas from stalling the progress of other replicas.

3.2.4 Controlling Byzantine Behavior. Although the optimistic ordering mechanism reduces the number of effective communication steps, Byzantine replicas can still prolong the latency by refusing to send messages (see Example 3.1), thereby negatively affecting performance. A common technique to prevent this behavior is by flooding the `D-PrePrepare` messages [5]. When a correct replica receives a `D-PrePrepare` message, it will multicast the message to all other replicas. This will ensure that the `D-PrePrepare` message will be received by other replicas in one communication step after it is initially received by a correct replica. In larger systems, we observed that using a random subset of few replicas was equally effective while reducing the additional bandwidth requirements.

Despite such techniques, a coordinator can still collude with the primary and cause a global ordering slot to be committed without disseminating the command. This will eventually cause a view change (described below), which when frequent can reduce the overall performance of the system. To prevent this behavior, we fall back to a more pessimistic approach on a per D-instance basis.

After a view change, the D-instance will be placed under *probation*, during which the O-instance primary will assign sequence numbers pessimistically after its commands are disseminated. If the D-instance appears to behave for a certain period (denoted using sequence numbers that exponentially increases with each view change), the optimistic mode is restored. If correct replicas identify that the O-instance primary assigns sequence numbers optimistically for a D-instance on probation period even after some grace period, the O-instance primary will be replaced via a view change. During this time, correct replicas will only respond after the respective command is disseminated.

3.2.5 View Change Protocol. The view change protocol is used to restore progress whenever a D-instance coordinator or the O-instance primary fails to do so, either deliberately or due to non-malicious causes (e.g., network disruptions). An important characteristic of DQBFT is that it leverages the existing view change procedure of the underlying consensus protocols without modifications. We assume an eventually synchronous [15] network between replicas where messages can be lost, be arbitrarily delayed, or arrive in any order, so it is impossible to distinguish a malicious primary or coordinator that does not send any messages from a network fault [25]. Thus, such protocols can guarantee progress only during periods of synchrony when messages arrive in bounded time.

Since, in DQBFT, replicas can serve multiple roles (e.g., primary, coordinator, backup) at the same time, it is possible that a replica makes progress on a subset of roles while ceasing progress on other roles. By using the view change procedures of the respective D-instances and the O-instance, we ensure that only those primary and coordinator roles that do not make progress are replaced, without affecting the other roles. The view change can cause a replica to coordinate multiple D-instances including its own, however a replica is allowed to propose new commands using only its D-instance.

Case 1: D-instance fails but O-instance is active. A client sends its command to its *assigned* coordinator. If it does not receive $f + 1$ responses for its command in time, it forwards the command to all replicas periodically. If timeouts happen often, a correct client can adapt by sending future commands to f or more replicas. Replicas will respond to the client if they have a reply. Correct replicas that have not yet seen the D-instance sequence number assigned for the command will forward the command to the target coordinator and wait for the coordinator to assign a sequence number under its D-instance and send the initial message. If the timers expire before receiving the message, correct replicas will invoke the view-change procedure for that D-instance. The failure of a D-instance does not

A DQPBFT replica executes the following sub-protocols:

Dissemination Protocol handled by D-instances:

N instances of the PBFT protocol are used for dissemination. Each replica "owns" one instance and replicates its client commands with that instance. The prefix "D-" and the replica identifier embedded in the messages helps to identify the protocol instance.

(1) **D-PrePrepare.** A replica R_n receives a client command α and sends a $\langle$`D-PrePrepare`, $v_n, R_n, \mathcal{S}_{ni}, \alpha\rangle$ message to all replicas. v_n is the view number and $\mathcal{S}_{ni}$ is the lowest available sequence number.

(2) **D-Prepare.** A replica R_m that receives a PrePrepare message $\langle$`D-PrePrepare`, $v_n, R_n, \mathcal{S}_{ni}, \alpha\rangle$ ensures the validity of the view and sequence numbers. Consequently, R_m sends a $\langle$`D-Prepare`, $v_n, R_n, \mathcal{S}_{ni}, Hash(\alpha)\rangle$ message to all the replicas.

(3) **D-Commit.** A replica that collects $2f + 1$ valid `D-Prepare` messages, sends the $\langle$`D-Commit`, $v_n, R_n, \mathcal{S}_{ni}, Hash(\alpha)\rangle$ message. A replica that receives $2f + 1$ valid Commit messages marks the operation as *disseminated*.

Global Ordering Protocol handled by the O-instance:

(1) **O-PrePrepare.** Case (i): If R_n is in *optimistic* mode, then the primary R_p of the O-instance assigns a global ordering number $\mathcal{S}_{pk}$ as soon as it receives the $\langle$`D-PrePrepare`, $v_n, R_n, \mathcal{S}_{ni}, \alpha\rangle$ message from R_n. Case (ii): If R_n is in *pessimistic* mode, then R_p assigns $\mathcal{S}_{pk}$ only after the operation corresponding to sequence number $\mathcal{S}_{ni}$ is marked as *disseminated*. Once assigned, Primary R_p sends the $\langle$`O-PrePrepare`, $v_p, R_p, \mathcal{S}_{pk}, R_n, \mathcal{S}_{ni}\rangle$ message to all replicas.

(2) **O-Prepare.** A replica R_q that receives the $\langle$`O-PrePrepare`, $v_p, R_p, \mathcal{S}_{pk}, R_n, \mathcal{S}_{ni}\rangle$ message ensures the validity of the view and sequence numbers. R_q also ensures that, Case (i): in the optimistic mode, there exists a corresponding `D-PrePrepare` message or Case (ii): in the pessimistic mode, the command has been disseminated, waiting if necessary. It then sends a $\langle$`O-Prepare`, $v_p, R_p, \mathcal{S}_{pk}\rangle$ message to all the replicas.

(3) **O-Commit.** A replica collects $2f + 1$ valid `O-Prepare` messages, and sends the $\langle$`O-Commit`, $v_p, R_p, \mathcal{S}_{pk}\rangle$ message. A replica that receives $2f + 1$ valid Commit messages commits its sequence number $\mathcal{S}_{pk}$ to map to R_n's sequence number $\mathcal{S}_{ni}$ and starts the execution procedure.

Figure 5: DQBFT execution using the PBFT Protocol for both the D-instance and O-instance protocols.

affect the O-instance progress, but can affect the execution phase. With the optimistic mode, it is possible that the O-instance primary globally orders the D-instance sequence number, but the sequence number did not commit before the view change, and no correct replica is aware of the command in that sequence number. Thus, total ordering of command and execution must wait until a new coordinator is chosen for the D-instance, and it disseminates either a command or a special *no-op* command. The *no-op* is proposed for all for sequence numbers that do not have a command associated but were committed in the O-instance

Case 2: O-instance fails but D-instances are active. When the O-instance primary fails, D-instances will continue disseminating commands, but they will not be globally ordered. After the O-instance finishes a view change, D-instances must send their requests to the new O-instance primary. The respective D-instance coordinator and a subset of correct replicas (see Section 3.2.4) will periodically send the D-PrePrepare to the new primary until the O-PrePrepare is received. A client can also time out if the O-instance primary fails to make timely progress. Therefore, correct replicas monitor the O-instance primary to ensure that it assigns corresponding global sequence numbers for those commands that have been *disseminated* in time. A view-change is triggered if necessary.

Case 3: Both O-instance and D-instance fail at once. The failure of the O-instance primary or a D-instance coordinator does not affect other active D-instances from disseminating commands. The view change protocols for the failed instances are run independently. If the D-instance changes view before O-instance does, it will continue disseminating new commands (same as Case 2). If the O-instance changes view before D-instance does, the O-instance will receive and order the sequence numbers for active D-instances (same as Case 1). If the new primary or coordinator fails to make progress, the respective instance undergoes another view change.

When a previously failed replica restarts, the view-change protocol is used to reinstate the replica's D-instance, i.e. make the original replica the coordinator of its D-instance. After the replica restarts, other replicas will trigger a view change, skipping views if necessary, to reinstate the replica immediately. Note that $f + 1$ replicas must agree to skipping views, so Byzantine replicas alone cannot reinstate. A correct replica will ensure that a recovered replica is participating in the protocol as a health check before agreeing to the view-change. Once reinstated, the replica must face probation.

3.2.6 Client. A client command contains an operation and a monotonically increasing timestamp. Every replica caches the last executed timestamp and the reply for each client. This is used to ensure that the replicas do not execute duplicate operations and to provide a reply to the client when required. Similar to other multi-primary protocols [29], each client is assigned to a replica to prevent request duplication attacks, where faulty clients can send duplicate commands to multiple replicas simultaneously. Even though replicas deduplicate commands during execution preserving safety, it can nullify the throughput improvements achieved by using multiple primaries. In DQBFT, this assignment is carried out by running consensus on a special `ASSIGN` message via the O-instance.

3.3 Correctness

DQBFT guarantees the following properties of a consensus protocol:

- **Safety.** Any two correct replicas will execute the same sequence of client requests.
- **Liveness.** A client request proposed by a replica will eventually be executed by every correct replica.

Lemma 3.2. *If a correct replica executes a command α whose D-instance sequence number $\mathcal{S}_{ni}$ is mapped to O-instance sequence number $\mathcal{S}_{pk}$ in view v, no correct replica will execute $\beta \neq \alpha$ at O-instance sequence number $\mathcal{S}_{pk}$ in view v.*

Proof. The D-instance and O-instance protocols satisfy Property P1. Thus, α is committed by replica R_n's D-instance at sequence number $\mathcal{S}_{ni}$ by correct replicas, and $\mathcal{S}_{ni}$ is the value committed at O-instance number $\mathcal{S}_{pk}$. Now, say a correct replica R_m executes β at $\mathcal{S}_{pk}$. This would entail that either (i) β was committed at $\mathcal{S}_{ni}$ by

correct replicas, or (ii) some $\mathcal{S}_{nj}$ assigned to β was committed at O-instance $\mathcal{S}_{pk}$ instead of $\mathcal{S}_{ni}$. This contradicts Property P1. □

Lemma 3.3. *If a correct replica executes a command α whose D-instance sequence number $\mathcal{S}_{ni}$ is mapped to O-instance sequence number $\mathcal{S}_{pk}$ in view v, no correct replica will execute $\beta \neq \alpha$ at O-instance sequence number $\mathcal{S}_{pk}$ in any view $v' > v$.*

Proof. Note that each instance satisfies Property P2. Thus, α will remain chosen at $\mathcal{S}_{ni}$ at all higher views, and $\mathcal{S}_{ni}$ will be mapped to $\mathcal{S}_{pk}$ at all higher views. Suppose a correct replica executes β at view $v' > v$, then either (i) β is assigned to $\mathcal{S}_{ni}$ by correct replicas, or (ii) some $\mathcal{S}_{nj}$ assigned to $beta$ was committed at O-instance $\mathcal{S}_{pk}$ instead of $\mathcal{S}_{ni}$. Both the conditions contradict Property P2. □

Theorem 3.4. *Any two correct replicas commit the same sequence of operations.*

Proof. Satisfied by Lemmas 3.2 and 3.3. □

Lemma 3.5. *During a stable view of the O-instance and the D-instance, a proposed client command is executed by a correct replica.*

Proof. In a stable view, a correct primary will propose client requests in a timely fashion to the replicas (Property P3). Thus, the D-instance primary will ensure dissemination of the client requests. Since there are at most f faulty replicas, there will remain $N - f$ correct ones that will respond to the primary's messages. Thus, the client commands will be committed during the view by correct replicas after receiving from a correct D-instance primary. The O-instance primary, being one of the correct replicas, will receive the D-instance primary's PrePrepare with the client command and sequence number. The correct O-instance primary will send the D-instance sequence number as its command to correct replicas, and it will be committed in the view. Thus, the client command will be assigned a global order by correct replicas mapping the D-instance sequence number to the corresponding O-instance. □

Lemma 3.6. *A view v will eventually transition to a new view $v' > v$ if at least $N - f$ replicas request for it.*

Proof. The proof follows directly from Property P4 applied to both the D-instances and the O-instance. □

Theorem 3.7. *A command sent by a correct client is eventually executed by correct replicas.*

Proof. During a stable view, Lemma 3.5 shows that the proposed command is learned by the correct replicas. When the view is unstable and the replica timers expire properly, $f+1$ correct replicas will request a view change. By Lemma 3.6, a new view v' will be installed. However, if less than $f + 1$ replicas request the view change, then the remaining replicas that do not request the view change will follow the protocol properly. Thus, the system will stay in view v and the replicas will continue to commit commands in the view. When proposals are not committed in time or when more than f replicas request a view change, then all correct replicas will request a view change, and it will be processed as in Lemma 3.5.

Even after a view change, the new view v' may not necessarily be stable. If the new primary deviates from the algorithm or does not make timely progress, correct replicas will request another view change and move to the next view. Since there can only be at most f faulty replicas, after at most $f + 1$ view changes, a stable view will be installed. Furthermore, if the faulty primary follows the algorithm enough such that a view change cannot be triggered, by Lemma 3.5, correct replicas will continue to commit commands. □

The individual consensus protocols satisfy linearizability [18]. The following theorem states that a command executed after committing via a D-instance and an O-instance satisfy linearizability.

Theorem 3.8. Linearizability*: If α and β are commands, and the request for β arrives after α is ready, then α will be executed before β.*

Proof. When α is ready, there must be at least i O-instance sequence numbers belonging to R_n. We prove this by contradiction. Assume there are less than i sequence numbers for R_n, but α is ready. This can happen only because there is a view change, and correct replicas observe less than i sequence numbers. However, since α was ready for execution before the view change, there is at least one correct replica that will ensure that the primary of the new view enforces no less than i instances, which is a contradiction.

When β is received after α is ready, there should be at least i O-instance sequence numbers committed belonging to R_n. There exists two cases. Case (i): If the O-instance primary is non-faulty, it will only assign sequence numbers in monotonically increasing order, so there will be no empty slots. Case (ii): After a O-instance view change, correct replicas will observe at least i sequence numbers belonging to R_n since α is ready, and they will ensure that the new primary enforces the i sequence numbers for R_n. □

4 DESTINY

We now present Destiny, the flagship instantiation of DQBFT that is designed to scale to hundreds of replicas, and achieve consistently high throughput and low latency even under high loads. While DQBFT is a general paradigm that can benefit any primary-based BFT protocol, our performance evaluation (in Section 5) reveals that not all protocols equally benefit from this approach (see Figure 3). Destiny takes advantage of the paradigm and achieves higher performance than state-of-the-art techniques [29, 50] at the scale of tens to hundreds of replicas.

Destiny assumes the Hybrid fault model in order to tolerate more faults than BFT protocols for the same system size and also benefit from smaller quorums ($f + 1$ instead of $2f + 1$). Destiny leverages a custom variant of Hybster [13], called Linear Hybster, to achieve its goal of higher performance and greater scalability. Linear Hybster improves Hybster's normal-case communication complexity from quadratic to linear using threshold signatures and specialized collector roles. The collector aggregates messages from replicas and re-broadcasts them to all replicas. Since the messages are cryptographically signed, threshold signatures [16, 49, 51] are used to reduce the number of outgoing collector messages from linear to constant. The same mechanism is employed for responding to the client. Clients wait for a single aggregated reply from a collector replica, instead of waiting for replies from $f + 1$ replicas. The collector replica collects the signatures from $f + 1$ replicas and sends a single response and signature to the client.

4.1 Preliminaries

4.1.1 Fault Assumptions and Cryptography. Destiny assumes the Hybrid fault model – the BFT model augmented with trusted components – in which replicas can behave arbitrarily, except the trusted

subsystem, which can only fail by crashing. Every replica, however, is capable of producing cryptographic signatures [33] that faulty replicas cannot break. We also assume a computationally bounded adversary that cannot do better than known attacks. The communication between replicas and clients is authenticated using public key infrastructures (PKI) like TLS. Being a hybrid protocol, Destiny only requires $N = 2f + 1$ replicas to tolerate f arbitrary failures.

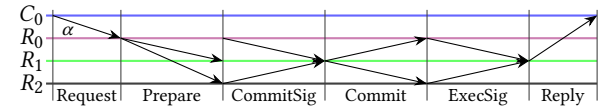
We consider an adversary that controls all the system software including the operating system. However, the adversary cannot read or modify the trusted subsystem's memory at run-time or decipher the secrets held inside it. Furthermore, the trusted subsystem is capable of generating cryptographic operations that the adversary cannot break. We also assume that the adversary cannot compromise the trusted subsystem's protections on participating nodes (e.g., via physical attacks). Preventing rollback attacks require replicating the subsystem state [43], which hybrid protocols perform during agreement implicitly. Further, note that compromise of the trusted component leads to safety violation of the protocol.

Destiny uses threshold signatures to linearize communication via the collector. The threshold signature with a threshold parameter t allows any subset t from a total of n signers to produce a valid signature on any message. It also ensures that no subset less than size t can produce a valid signature. For this purpose, each signer holds a distinct private signing key that can be used to generate the corresponding signature share. The signature shares of a signed message can be combined into a single signature that can be verified using a single public key. We use a threshold signature scheme based on Boneh-Lynn-Shacham (BLS) signatures [42]. Particularly, we use the BLS12-381 [12] signature scheme that produces 192-byte signature shares. The aggregate signatures are also 192 bytes long.

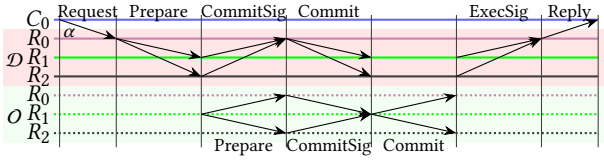
4.1.2 The ThreshSign Subsystem. ThreshSign is a local service that exists on every replica. It allows for creating and verifying different types of threshold signatures for a message m using a specified counter tc and a corresponding counter value tv. By hosting part of ThreshSign in a trusted subsystem, the system guarantees a set of properties (described later) even if the replica is malicious.

ThreshSign provides the following functions:

- **Independent Counter Signature Shares** with input (m, tc, tv'). ThreshSign generates such a signature for a message m if the provided new value tv' for counter tc is *greater than* its current value tv. It updates the counter tc's value to tv' and computes a signature share using the subsystem's instance ID, counter tc's ID, its new value tv', current value tv, and the message m.
- **Aggregate Signature Shares**. It returns a single signature by aggregating at least t valid threshold signatures.
- **Verify Signature**. It verifies the aggregated signature *sig* using the public key and indicates whether message m was signed by t replicas with counter value tv of counter tc.
- **Continuing Counter Certificates** with input (m, tc, tv, tv'). ThreshSign generates a message authentication code (MAC) certificate for a message m if the submitted new value tv' for counter tc is *greater than or equal to* its current value tv. It updates the counter tc's value to tv' and computes a signature share using its private key share, the subsystem's instance ID, counter tc's ID, its new value tv', current value tv, and the message m.



(a) Linear Hybster. R_0 is the primary and R_1 is the collector.



(b) Destiny. R_0 and R_1 serve both primary and collector roles for the D-instance and the O-instance, respectively.

Figure 6: Linear Hybster and Destiny Agreement Execution.

- **Verify Certificate** with input (m, mac, tc, tv, tv'). It verifies the MAC certificate mac using the secret key and returns true if message m is assigned a continuing certificate that transitions the counter tc from tv to tv'.

ThreshSign also provides the capabilities of TrInX [13, 37], the original Hybster's trusted component to aid the view change and state transfer mechanisms. We implement the ability to instantiate multiple ThreshSign instances within a single trusted subsystem. Every instance can host a variable number of counters as needed by the protocol. For instance, Hybster requires certificates using at least three different counters for different protocol phases (e.g., checkpoints, view changes). Furthermore, the signing and the certifying functions must be hosted securely along with the private keys inside the trusted subsystem, while the signature aggregation and verification functions may be hosted outside as they only deal with public keys. We rely on attestation services provided by hardware vendors to verify that the code running inside the enclave is secure and perform any initialization steps.

4.2 Linear Hybster

We now discuss the modifications to Hybster [13], to achieve linear communication in the common case to create Linear Hybster.

Figure 6a shows Linear Hybster's execution steps in the normal case. Hybster commits a command in two steps and requires clients to wait for $f + 1$ replies. A quadratic number of *Commit* messages are exchanged by replicas in an all-to-all communication, which bottlenecks throughput. We use a collector and an additional communication step to reduce this quadratic communication to linear. In *Linear Hybster*, replicas send the *Commit* messages to a collector (up to $f + 1$ can be used for fault-tolerance), which aggregates at least $f + 1$ messages and sends them to other replicas. Hybster uses *TrInX*, a *trusted* MAC provider which requires any pair of replicas to use unique secret keys to exchange messages between them. We replace *TrInX* with ThreshSign subsystem. ThreshSign is configured with a threshold of $f + 1$ out of $N = 2f + 1$ total replicas.

Hybster (and most BFT protocols) require that the clients wait for equivalent replies from at least $f + 1$ replicas to defend against

incorrect responses from malicious replicas. Linear Hybster, in contrast, reduces this $f+1$ communication to one single message using threshold signatures. For this purpose, Linear Hybster uses another instance of threshold signatures, π, with threshold $f+1$. Now, once the client command is executed at each replica, the result of execution is signed using π and sent to the collector in a ExecSig message. The collector collects and aggregates signature shares from $f+1$ valid ExecSig messages and generates an ExecProof message. This message is sent to the replicas as well as the client along with the result of execution. The client validates the aggregated signature, accepts the result, and returns.

View Change. A replica triggers a view change if it does not receive timely messages from the leader, or if it receives a proof that the leader is faulty (either via a publicly verifiable contradiction from the client or when $f+1$ replicas complain).

Replica R_m supports a new primary R'_p of a view $v+1$ by sending a ViewChange message with the Prepares for all order numbers in its current ordering window in view v. A continuing counter certificate is attached to the message to ensure that even if replica R_i is faulty, it includes all the Prepares it is aware of up to the current order number. After sending the ViewChange message for $v+1$, replica R_m is prohibited from participating in view v. Due to the use of continuing counter certificates, a new leader R'_p can determine all the proposals of the former primary R_p by collecting only a quorum of ViewChange messages.

Once a correct leader R'_p collects at least $f+1$ ViewChange messages, it begins constructing the new view. It is possible that the new leader is lagging behind the current ordering window, in which case the new leader invokes the state-transfer protocol to request the checkpoint messages and the service state from an up-to-date replica. A replica cannot establish as a new leader until its ordering window overlaps with the ViewChange messages. Since only f replicas can be faulty at most, there is at least one correct replica that contains the adequate information to help the new primary move to the new ordering window.

Unlike the agreement protocol, the view change mechanism uses continuing counter certificates provided by ThreshSign. For a view change, replicas individually must announce their current view and their intended view, unlike normal case execution where replicas jointly accept a proposed command. Continuing counter certificates serve this purpose well, allowing replicas to individually prove their log state to other replicas and the new primary.

4.3 Protocol

Destiny is an instantiation of DQBFT using the Linear Hybster protocol presented above. Destiny uses $7N$ messages and five phases in the optimistic (seven in the pessimistic) case to execute each command (see Figure 2). Due to linear communication, Destiny's theoretical throughput closely matches the maximum concurrent throughput $T_{cmaxHybrid}$ (Figure 3). For brevity, we only provide an overview of Destiny, leveraging the description in Section 3.

Agreement Protocol. Destiny commits both D-instances and O-instances using the Linear Hybster protocol with acknowledgements from a majority quorum. Figure 6b illustrates the communication steps of the Agreement protocol in the optimistic case. The messages in the normal phase protocol are signed by invoking the *Independent Counter Signature Shares* function of the corresponding ThreshSign instance. This ensures the following properties: (i) *Uniqueness*: the same counter value is not assigned to two different messages, and (ii) *Monotonicity*: the counter value assigned to a message will always be greater than the previous counter value.

Execution and Acknowledgement. Replicas execute commands as they become *ready* for execution. After execution, as in Linear Hybster, replicas forward the signed result to a collector, which then aggregates $f+1$ signatures. The collector sends this signature back to the replicas and to the client, indicating that the client's command was executed. Note that this step does not require the use of the trusted subsystem.

Checkpoint, State-transfer and View change Protocols. Destiny uses the respective checkpoint, state-transfer, and view change algorithms of the underlying Linear Hybster protocol.

Example 4.1. Figure 6b illustrates how Destiny optimistically commits a command using the D- and O-instance protocols.

Assume that R_1 serves the primary role in the O-instance protocol. A client submits command α to replica R_0. R_0 becomes the *initial* coordinator of α. We also assume that R_0 and R_1 will play the collector roles for the D-instance and O-instance, respectively. A replica playing the collector role is responsible for collecting signature shares, aggregating them, and multicasting the combined signature. R_0 selects the lowest unused sequence number in its D-instance space, assigns it to α, and disseminates the command by multicasting a `D-Prepare` message.

R_1 receives the `D-Prepare` message and triggers the O-instance protocol for replica R_0. R_1 proposes R_0's ID and α's sequence number to the next available O-instance sequence number (say j) and sends a `O-Prepare` message. Replicas accept either Prepare messages and send the corresponding `D-CommitSig` and `O-CommitSig` messages, respectively, to the commit collectors, R_0 and R_1. The `D-CommitSig` and the `O-CommitSig` messages are signed by the σ_0 and τ ThreshSign instances, respectively. The respective commit collectors wait for at least $f+1$ valid `D-CommitSig` (respectively `O-CommitSig`) messages and invoke the ThreshSign subsystem to aggregate the signature shares in the commit messages into a single signature. The aggregated signatures are sent via the corresponding `D-Commit` and `O-Commit` messages.

Replicas receive the valid `O-Commit` and `D-Commit` messages, commit the command, and mark it for execution. After executing the command at the global order number j, each replica signs the resulting state and sends a signed `ExecSig` message to the *execution* collector. The execution messages are signed using a ThreshSign instance that is different from the ones used during commit. This ThreshSign instance does not require trust and is kept outside of the trusted subsystem. The execution collector R_0 collects at least $f+1$ valid `ExecSig` messages, aggregates the signatures into a single `ExecProof` message, and sends the message to all the replicas. It also sends this message to the client along with the result of execution. The client verifies the signature, accepts the result, and returns.

5 EVALUATION

We implemented multiple protocols under DQBFT and evaluated them against state-of-the-art single-primary and multi-primary protocols. Our evaluation answers the following questions:

(1) What is the impact of batching on protocol performance?
(2) How well do the protocols scale their performance when increasing the system size from 10s to 100s of replicas in a geo-distributed deployment?
(3) What is performance impact under replica failures?
(4) How do the DQBFT protocols compare to other multi-primary protocols?

5.1 Protocols under Test

Our evaluation includes the following state-of-the-art protocols.

Single-primary protocols. We evaluate PBFT [18], Hybster [13], and SBFT [26]. We use the variant of PBFT that uses MACs that are computationally cheaper than signatures. SBFT uses linear communication and $3f+c+1$ fast-path quorum with $3f+2c+1$ replicas. We set c to zero, because increasing c does not improve fault tolerance. Chained Hotstuff [58] is a rotating-primary protocol.

Multi-primary protocols. Prime [5] allows individual replicas to disseminate commands using Reliable Broadcast, and a primary provides an ordering for the disseminated commands periodically. Dispel [57] uses Reliable Broadcast to disseminate commands, and uses multiple instances of leaderless binary consensus to order the commands. MirBFT [50] allows multiple replicas to act as primaries concurrently by distributing sequence numbers evenly. It uses the notion of an epoch to define which replicas can be primaries during a certain period. RCC [29] allows multiple replicas to act as primaries and uses the notion of rounds to facilitate a global execution order. In each round, one command is committed by each of the primaries and a deterministic execution order is decided.

DQBFT protocols. DQPBFT, DQSBFT, and DQHybster are DQBFT instantiations of the original protocols PBFT, SBFT, and Hybster, respectively. We also evaluated Linear Hybster and Destiny.

We implemented all the protocols in a common framework in Golang. We favored our own implementations over the author versions for a fair and consistent evaluation. For instance, the authors' version of Hotstuff only disseminates command hashes [50], but all our implementations disseminate actual payloads. In addition, the source code for RCC and Hybster were not publicly available. The trusted components were implemented in C++ using the Intel SGX SDK [22] . Our implementations of BFT protocols perform out-of-order processing of commands, except Hotstuff, which does not support out-of-order processing because it rotates the primary's role regularly. For Hybrid protocols, out-of-order processing is limited due to the use of counter-based trusted components: creation of signatures using the trusted components happen in order, whereas all other message processing happens out-of-order.

5.2 Experimental Setup

We used SGX-enabled virtual machines (VMs) available on Microsoft's Azure [44] platform. We obtained VMs from ten different datacenter regions: six in North America, three in Europe, and one in South East Asia. The protocols were deployed in each of these regions leveraging multiple VMs. The number of VMs depends on the experiment. Each VM consists of 8 vCPUs and 32GB of memory (best available at the time of experiments). The VMs were part of a Kubernetes [27] cluster and the protocol replicas and clients were deployed as *pods*. We placed one replica pod per VM and placed the clients on different VMs. We designated a replica in Eastern US to serve the primary's role. The network latencies between regions are in [2]. The bandwidth between replicas ranged from 400 Mb/s (between US and Asia) and 6 Gb/s (within same region).

We carried out experiments for five different values of N (the number of replicas): 19, 49, 97, 193, 301 tolerating 6, 16, 32, 64, 100 BFT and 9, 24, 48, 96, 150 Hybrid failures, respectively. For each experiment, replicas were evenly spread among the ten regions. Clients send requests in a closed-loop, meaning they wait for the result of the previous request before sending the next one. Unless otherwise stated, clients are evenly spread across all the region and send commands to their local replicas for multi-primary protocols and to the primary for single-primary protocols. Our performance numbers account for both the consensus and execution time. We use Prometheus [56] timeseries database to collect metrics from the replicas *periodically* and report our results. The state is a fully-replicated in-memory key value store, a useful abstraction for building other applications including smart contract engines [26]. The workload is 100% put operations with 20-byte keys and random values. The command payload size is 512 bytes. Unless otherwise stated, each batch contains 200 commands producing a proposal size of ≈106kB. The size of other protocol messages are around 100B for DQPBFT and 300B for Destiny.

5.3 Experiments

5.3.1 Batching Experiment. First, we measured the impact of batching commands on protocol performance. Increasing the batch size increases the size of the proposal message multicast by the primary (or the coordinator in the case of DQBFT). For this experiment, we deployed $N = 97$ replicas, increased the batch size from 10 to 1200 commands/batch, and measured the performance. Figure 7 shows the results. The single-primary protocols reach their maximum throughput at batch size of 100, as their primaries' are bandwidth saturated. Hybster's performance is limited by the overhead of in-order MAC attestation mechanism that requires command payloads to be copied into the trusted enclave. In contrast, threshold signatures in Linear Hybster require only the command hash to be copied into the enclave. This along with linear communication complexity enables Linear Hybster to compete with PBFT-MAC [18] and SBFT. Chained Hotstuff's throughput is significantly limited because it rotates the primary for each batch that disallows out-of-order processing of multiple batches simultaneously. Thus, its latency is higher because each replica must wait for 96 other replicas to propose before its turn.

The multi-primary protocols show multifold increases in throughput compared to single-primary protocols by virtue of allowing multiple replicas to propose simultaneously. RCC, MirBFT, and DQPBFT perform similarly because under non-faulty scenarios their effective behaviors are the same. Note that RCC is also a BFT paradigm and can also be instantiated with SBFT; we observed that its performance to be on par with DQSBFT's performance in this setting. Destiny's performance exceeds all other protocols with 35% better throughput than the next best protocol DQSBFT and 40% lower latency than other multi-primary protocols. Destiny performs better because aggregating $f + 1$ signature shares is computationally cheaper than aggregating $3f + 1$ shares [52], and the $f + 1$

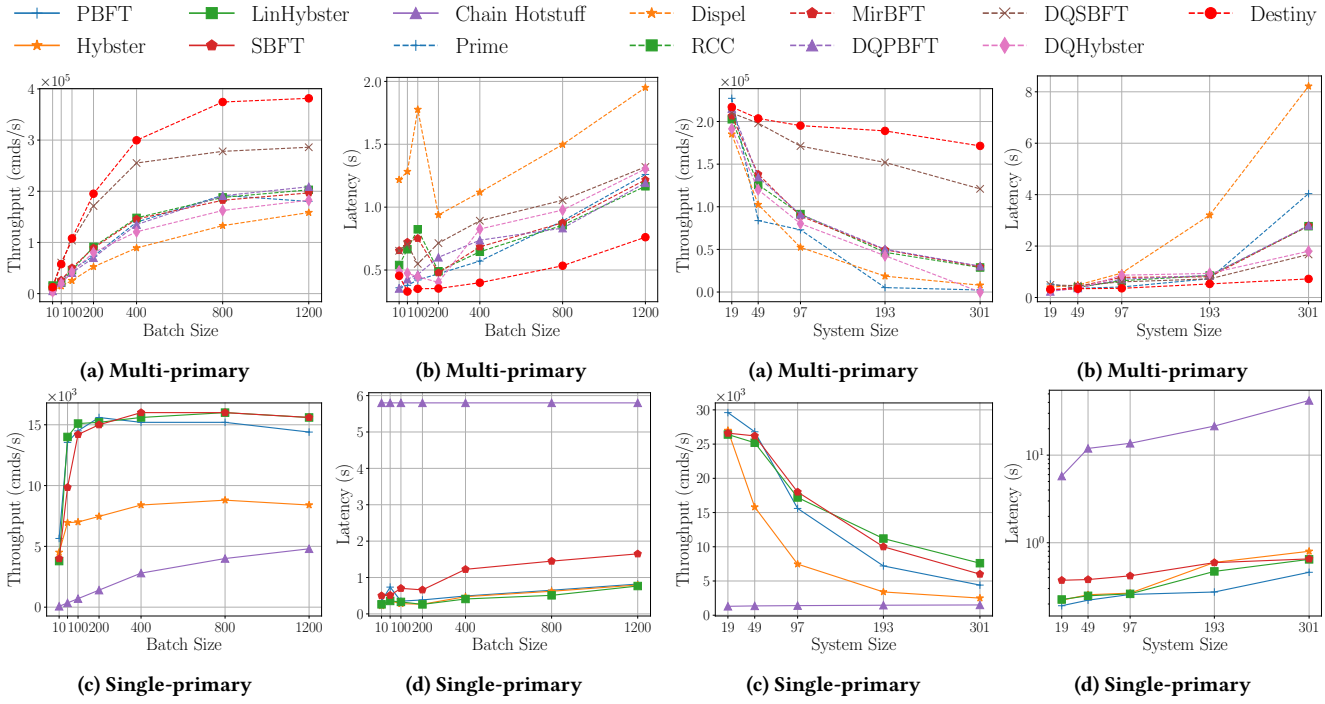


Figure 7: Performance versus Batch Size (N=97).

Figure 8: Performance versus System Size.

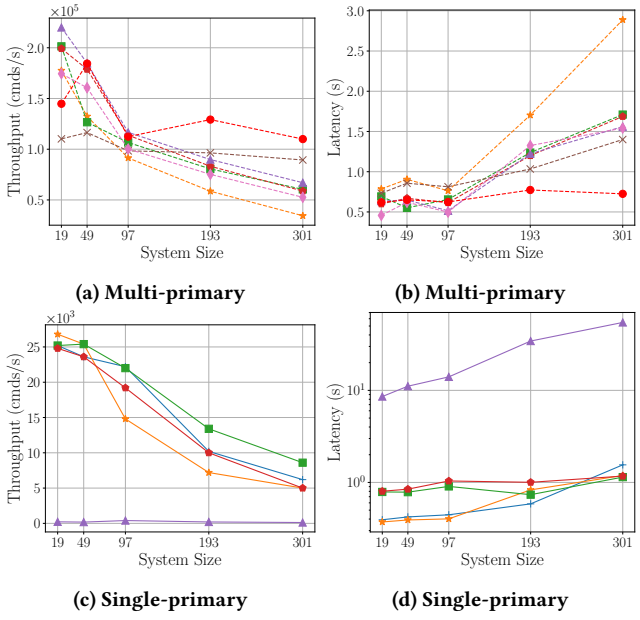


Figure 9: Performance under f failures.

quorum gives f additional replicas to provide redundancy from slow nodes and staggering network, unlike SBFT. Note that this experiment also serves to demonstrate the impact of increasing the command size because the execution overheads are small for our key-value store. For instance, a batch with 200 1024-byte commands will perform similarly to a batch with 400 512-byte commands.

5.3.2 Scalability Experiment. Second, we measured the performances of the protocols while increasing the system size, i.e., the number of replicas, from 19 to 301 replicas. Figure 8 shows the results. The performances of single-primary protocols decrease with increasing N since the primaries must send the initial payload ($\approx 100kB$) to all the replicas. On the other hand, multi-primary protocols have a higher peak throughput than single-primary protocols by virtue of enabling multiple replicas to send the initial payload that distributes the bandwidth requirements among all replicas. As with the batching experiment, the performance trends for RCC, MirBFT, and DQPBFT are similar. Destiny's scales better than all other protocols. At $N = 301$, Destiny provides 40% higher throughput and 70% lower latency than the next best protocol DQSBFT.

We also analyzed the network and CPU utilization at $N = 97$. In single-primary protocols, the primaries used ≈6Gbps of network and 50%-65% of CPUs, but the replicas used only ≈115Mbps of network and 10%-20% CPUs. In DQBFT protocols, the average network and CPU usages were ≈1.5Gbps and 65%. Destiny's CPU usage reached 95% indicating that the other DQBFT protocols were limited by their bandwidth (inline with Figure 3).

5.3.3 Scalability under Failures. We also evaluated the protocols under failures by repeating the scalability experiment with f failed replicas. Failed replicas are equally spread among the ten regions . Figure 9 shows the results. Note that SBFT, DQSBFT, and Destiny are more negatively impacted by f failures than *DQPBFT*, RCC, and MirBFT. Both SBFT and *DQSBFT* must take the slow path with additional communication steps since a fast quorum, which is equal to the system size, is unavailable. Thus, in this case, DQSBFT

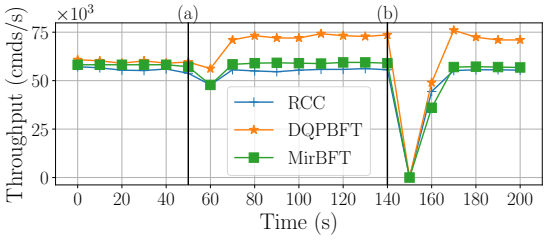


Figure 10: Throughput timeline with slow replicas and injected failures. At (a), the number of clients is doubled in all but one region causing a replica to slow down. At (b), a random replica is killed to invoke the view change procedure.

only performs as good as DQPBFT. Further, Destiny must wait for messages from all the regions instead of only a majority. Despite, Destiny performs better than multi-primary protocols at 193 and 301 replicas because its linear communication pays off at that scale.

Note that Dispel's latency is substantially lower in the failure case than in the failure-free case as a result of fewer replicas participating in a given round. Similarly, we also observe lower latencies for other multi-primary protocols than the failure-free case as fewer messages are sent and processed by each replica.

5.3.4 Single Replica Failure Experiments. While the previous experiments show that Destiny performs better than existing multi-primary protocols, the other DQBFT protocols, DQPBFT and DQSBFT, only perform as good as their RCC counterparts. So far, the number of clients were balanced equally among the regions. However, in practice, it may not be feasible to ensure a uniform request rate across all replicas, because certain regions may have more load than others, e.g., due to geographical characteristics such as time zones, or even Byzantine behaviors. Therefore, we devised an experiment to compare the performance of DQPBFT with RCC and MirBFT when request rates are imbalanced among replicas.

For this experiment, we deployed 97 replicas that are spread among the ten regions, and increased the number of clients non-uniformly over time. Figure 10 shows the results. Initially, at $t = 0$, clients are spread evenly among the replicas, during which all three protocols, namely, RCC, DQPBFT, and MirBFT, perform similarly as their behavior is effectively the same under these conditions. At $t = 50$, we double the number of clients in all regions except one, namely South East Asia. Following this, at $t = 60$, the replicas are overwhelmed by the sudden increase in requests from the new clients and lose throughput momentarily before bouncing back. As the system stabilizes, DQPBFT's throughput increases by 25% while that of RCC and MirBFT remain the same as before. The O-instances in DQBFT protocol enable each replica to deliver commands at its own pace without waiting for other replicas' deliveries. In contrast, the round-robin deliveries in MirBFT and RCC throttles all the replicas to deliver commands at the speed of the slowest replica.

When replicas fails, the protocols stop delivering commands. DQPBFT must recover the failed replica's non-disseminated but globally-ordered D-instance sequence numbers, while RCC must stop the failed instance and reconcile its current round state. Figure 10 shows this effect at $t = 140$ when a replica fails. The protocols use their respective view change procedures to restore progress.

6 RELATED WORK

Numerous performance-oriented single-primary BFT protocols [10, 18, 26, 35, 55] have been proposed in literature. In Section 2.2, we discussed the limitations of primary-based, the rotating-leader [53, 54, 58] and dependency-based ordering [8, 28] approaches.

Request dissemination [5, 20, 21, 31] has been proposed as a means to relieve primary's workload. These solutions use Reliable Broadcast, which lack the agreement property, forcing replication to always precede ordering, thus increasing the overall latency.

The idea of separating replication and global ordering has been explored in the crash fault model [11, 38, 60]. SDPaxos [60] separates replication from ordering, and uses a consensus protocol for both the tasks. DQBFT's separation technique can be viewed as the BFT counterpart, but our design is optimized for scalability to hundreds of replicas, while SDPaxos focuses on minimizing latency in up to five-replica deployments. Furthermore, distributed log protocols (e.g. Corfu [11]) use a benign sequencing node to dictate global order. To prevent malicious sequencers from violating consistency, the O-instance in DQBFT must assign sequence numbers by reaching BFT consensus. Moreover, the interaction between D- and O- instances must ensure that none of the instances compromise the safety/liveness properties of each other. NoPaxos [38] requires special network devices, thus is suitable only within a datacenter.

Various trusted component designs have been proposed previously for Hybrid protocols [13, 19, 34, 55]. The trusted counter design is simple and memory-efficient compared to log-based designs. Among the known Hybrid protocols, we chose Hybster because the protocol's is designed specifically for commodity processors with trusted subsystems such as Intel SGX. Threshold secret shares can be used in place of threshold signatures [40], but requires creating a set of secret key shares for each command and exposing it when committing each command. This requires additional computational and network resources. PoET [4] uses the trusted component to dictate the minimum time period between block proposals, thus adopting the synchrony timing model.

Alternate (e.g. XFT [41]) and mixed fault models (e.g. Hierarchical [6, 30]) have been proposed to improve performance in geo-distributed systems. XFT assumes synchronous communication among majority replicas for safety, while Destiny assumes the trusted component for its safety. Unlike mixed fault models, DQBFT tolerates f global failures without limits on regional failures.

7 CONCLUSION

In conclusion, we show that DQBFT is an effective paradigm for designing highly scalable BFT protocols. Furthermore, with Destiny, we show that linear communication and smaller quorums elevate the performance of DQBFT protocols.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their multiple rounds of extremely diligent, meticulous, and insightful comments which greatly improved the paper. This work is supported in part by US National Science Foundation under grant CNS 1523558, Air Force Office of Scientific Research under grants FA9550-15-1-0098 and FA9550-16-1-0371, and Office of Naval Research under grant N00014-17-1-2297.

REFERENCES

[1] 2021. sgxwallet: SKALE SGX-based hardware crypto wallet. https://github.com/skalenetwork/sgxwallet. Accessed: 2021-07-02.

[2] 2022. Azure Network Latency Statistics. https://docs.microsoft.com/en-us/azure/networking/azure-network-latency. Accessed: 2022-02-13.

[3] 2022. Bitcoin Transaction Size Chart. https://bitcoinvisuals.com/chain-tx-size. Accessed: 2022-02-13.

[4] 2022. Hyperledger Sawtooth. https://sawtooth.hyperledger.org/docs/core/nightly/master/architecture/poet.html. Accessed: 2022-02-13.

[5] Y. Amir, B. Coan, J. Kirsch, and J. Lane. 2011. Prime: Byzantine Replication under Attack. *IEEE Transactions on Dependable and Secure Computing* 8, 4 (July 2011), 564–577. https://doi.org/10.1109/TDSC.2010.70

[6] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. 2010. Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. *IEEE Transactions on Dependable and Secure Computing* 7, 1 (Jan 2010), 80–93. https://doi.org/10.1109/TDSC.2008.53

[7] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. [n.d.]. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal, 2018) *(EuroSys '18)*. ACM, 30:1–30:15. https://doi.org/10.1145/3190508.3190538

[8] B. Arun, S. Peluso, and B. Ravindran. 2019. ezBFT: Decentralizing Byzantine Fault Tolerant State Machine Replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*.

[9] B. Arun, S. Peluso, and B. Ravindran. 2019. ezBFT: Decentralizing Byzantine Fault-Tolerant State Machine Replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 565–577. https://doi.org/10.1109/ICDCS.2019.00063

[10] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The Next 700 BFT Protocols. 32, 4 (2015), 12:1–12:45. https://doi.org/10.1145/2658994

[11] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A Distributed Shared Log. *ACM Trans. Comput. Syst.* 31, 4, Article 10 (Dec. 2013), 24 pages. https://doi.org/10.1145/2535930

[12] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. 2003. Constructing Elliptic Curves with Prescribed Embedding Degrees. In *Proceedings of the 3rd International Conference on Security in Communication Networks* (Amalfi, Italy) *(SCN'02)*. Springer-Verlag, Berlin, Heidelberg, 257–267. http://dl.acm.org/citation.cfm?id=1766811.1766837

[13] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, 222–237. https://doi.org/10.1145/3064176.3064213

[14] Alysson Bessani, João Sousa, and Marko Vukolić. 2017. A Byzantine Fault-tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers* (Las Vegas, Nevada) *(SERIAL '17)*. ACM, New York, NY, USA, Article 6, 2 pages. https://doi.org/10.1145/3152824.3152830

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

[16] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *J. Cryptol.* 18, 3 (July 2005), 219–246. https://doi.org/10.1007/s00145-005-0318-0

[17] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, Berkeley, CA, USA, 173–186. http://dl.acm.org/citation.cfm?id=296806.296824

[18] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.* 20, 4 (Nov. 2002), 398–461. https://doi.org/10.1145/571637.571640

[19] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested Append-only Memory: Making Adversaries Stick to Their Word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. ACM, 189–204. https://doi.org/10.1145/1294261.1294280

[20] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. Upright Cluster Services. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. ACM, 277–290. https://doi.org/10.1145/1629575.1629602

[21] Miguel Correia, Giuliana S. Veronese, and Lau Cheuk Lung. 2010. Asynchronous Byzantine Consensus with 2f+1 Processes. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (Sierre, Switzerland) *(SAC '10)*. Association for Computing Machinery, New York, NY, USA, 475–480. https://doi.org/10.1145/1774088.1774187

[22] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. 2016, 086 (2016), 1–118.

[23] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. 2006. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, 177–190.

[24] Xavier Défago, André Schiper, and Péter Urbán. [n.d.]. Total order broadcast and multicast algorithms: Taxonomy and survey. 36 ([n. d.]), 372–421. Issue 4. https://doi.org/10.1145/1041680.1041682

[25] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (apr 1985), 374–382. https://doi.org/10.1145/3149.214121

[26] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 568–580. https://doi.org/10.1109/DSN.2019.00063

[27] Google. [n.d.]. Kubernetes. https://kubernetes.io/.

[28] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The Next 700 BFT Protocols. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. ACM, 363–376. https://doi.org/10.1145/1755913.1755950

[29] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *Int. Conf. on Data Engineering (ICDE)*.

[30] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proc. VLDB Endow.* 13, 6 (Feb. 2020), 868–883. https://doi.org/10.14778/3380750.3380757

[31] Vassos Hadzilacos and Sam Toueg. 1994. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems.* Technical Report. USA.

[32] Maurice P. Herlihy and Jeannette M. Wing. [n.d.]. Linearizability: a correctness condition for concurrent objects. 12, 3 ([n. d.]), 463–492. https://doi.org/10.1145/78969.78972

[33] Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The Elliptic Curve Digital Signature Algorithm (ECDSA). *Int. J. Inf. Secur.* 1, 1 (Aug. 2001), 36–63. https://doi.org/10.1007/s102070100002

[34] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. ACM, 295–308. https://doi.org/10.1145/2168836.2168866

[35] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: Speculative Byzantine Fault Tolerance. (2007), 45–58. https://doi.org/10.1145/1294261.1294267

[36] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: Speculative Byzantine Fault Tolerance. 27, 4 (2010), 7:1–7:39. https://doi.org/10.1145/1658357.1658358

[37] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Boston, Massachusetts) *(NSDI'09)*. USENIX Association, 1–14.

[38] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 467–483.

[39] ARM Limited. [n.d.]. ARM Security technology: building a secure system using TrustZone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. ARM Technical White Paper, Accessed: 2018-11-06.

[40] J. Liu, W. Li, G. O. Karame, and N. Asokan. 2019. Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing. *IEEE Trans. Comput.* 68, 1 (Jan 2019), 139–151. https://doi.org/10.1109/TC.2018.2860009

[41] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. 2016. XFT: Practical Fault Tolerance Beyond Crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, 485–500.

[42] Ben Lynn. 2007. *On the implementation of pairing-based cryptosystems.* Ph.D. Dissertation. Stanford University Stanford, California.

[43] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1289–1306. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic

[44] Microsoft. [n.d.]. Azure Confidential Computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[45] Iulian Moraru. 2015. Egalitarian Distributed Consensus. http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-CS-14-133.pdf.

[46] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, 358–372. https://doi.org/10.1145/2517349.2517350

[47] Satoshi Nakamoto. 2008. *Bitcoin: A peer-to-peer electronic cash system.* Technical Report.

[48] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. 2021. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 35–48. https://doi.org/10.1145/3477132.3483584

[49] Victor Shoup. 2000. Practical Threshold Signatures. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques* (Bruges, Belgium) *(EUROCRYPT'00)*. Springer-Verlag, Berlin, Heidelberg, 207–220. http://dl.acm.org/citation.cfm?id=1756169.1756190

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

[51] C Stathakopoulous and Christian Cachin. 2017. Threshold signatures for blockchain systems. *Swiss Federal Institute of Technology* (2017).

[52] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. 2020. Towards Scalable Threshold Cryptosystems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 877–893. https://doi.org/10.1109/SP40000.2020.00059

[53] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2009. Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems (SRDS '09)*. IEEE Computer Society, Washington, DC, USA, 135–144. https://doi.org/10.1109/SRDS.2009.36

[54] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2010. EBAWA: Efficient Byzantine Agreement for Wide-Area Networks. In *Proceedings of the 2010 IEEE 12th International Symposium on High-Assurance Systems Engineering (HASE '10)*. IEEE Computer Society, Washington, DC, USA, 10–19. https://doi.org/10.1109/HASE.2010.19

[55] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Comput.* 62, 1 (Jan. 2013), 16–30. https://doi.org/10.1109/TC.2011.221

[56] Julius Volz and Björn Rabenstein. 2015. Prometheus: A Next-Generation Monitoring System (Workshop). USENIX Association, Dublin.

[57] Gauthier Voron and Vincent Gramoli. 2019. Dispel: Byzantine SMR with Distributed Pipelining. *CoRR* abs/1912.10367 (2019). arXiv:1912.10367 http://arxiv.org/abs/1912.10367

[58] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591

[59] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town Crier: An Authenticated Data Feed for Smart Contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 270–282. https://doi.org/10.1145/2976749.2978326

[60] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. 2018. SD-Paxos: Building Efficient Semi-Decentralized Geo-Replicated State Machines. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 68–81. https://doi.org/10.1145/3267809.3267837