The Bedrock of Byzantine Fault Tolerance: A Unified Platform for BFT Protocols Analysis, Implementation, and Experimentation

Mohammad Javad Amiri[1] *Chenyuan Wu*[1] *Divyakant Agrawal*[2] *Amr El Abbadi*[2]
Boon Thau Loo[1] *Mohammad Sadoghi*[3]
[1]*University of Pennsylvania,* [2]*University of California Santa Barbara,* [3]*University of California Davis*

Abstract

Byzantine Fault-Tolerant (BFT) protocols cover a broad spectrum of design dimensions from infrastructure settings, such as the communication topology, to more technical features, such as commitment strategy and even fundamental social choice properties like order-fairness. The proliferation of different BFT protocols has made it difficult to navigate the BFT landscape, let alone determine the protocol that best meets application needs. This paper presents *Bedrock*, a unified platform for BFT protocols analysis, implementation, and experimentation. Bedrock proposes a design space consisting of a set of dimensions and explores several design choices that capture the trade-offs between different design space dimensions. Within Bedrock, a wide range of BFT protocols can be implemented and uniformly evaluated under a unified deployment environment.

1 Introduction

Distributed systems rely on fault-tolerant protocols to provide robustness and high availability [40, 54, 60, 84, 99, 138, 189]. While cloud systems, e.g., Google's Spanner [84], Amazon's Dynamo [99], and Facebook's Tao [60], rely on crash fault-tolerant protocols, e.g., Paxos [157], to establish consensus, a Byzantine fault-tolerant (BFT) protocol is a key ingredient in distributed systems with non-trustworthy infrastructures, e.g., permissioned blockchains [1–3, 29, 42, 64, 79, 124, 155, 205], permissionless blockchains [61, 147, 149, 176, 241], distributed file systems [14, 71, 82], locking service [83], firewalls [52, 119, 120, 211, 220, 239], certificate authority systems [245], SCADA systems [38, 146, 197, 244], key-value datastores [50, 103, 123, 134, 211], and key management [179].

BFT protocols use the State Machine Replication (SMR) technique [156, 212] to ensure that non-faulty replicas execute client requests in the same order despite the concurrent failure of at most f Byzantine replicas. BFT SMR protocols are different along several dimensions, including the number of replicas, processing strategy (i.e., optimistic, pessimistic, or robust), supporting load balancing, etc. While dependencies and trade-offs among these dimensions lead to several design choices, there is currently no unifying tool that provides the foundations for studying and analyzing BFT protocols' design dimensions and their trade-offs. We envision that such a unifying foundation will provide an in-depth understanding of existing BFT protocols, highlight the trade-offs among dimensions, and will enable protocol designers to choose among several dimensions to find the protocol that best fits the characteristics of their applications.

This paper presents *Bedrock*, a unified platform that enables us to analyze, implement, and experiment with partially asynchronous SMR BFT protocols within the design space of possible variants. Bedrock presents a design space to characterize BFT protocols based on different dimensions that capture the environmental settings, protocol structure, QoS features, and performance optimizations. Each protocol is a plausible point in the design space. Within the design space, Bedrock defines a set of design choices demonstrating trade-offs between different dimensions. For example, the communication complexity can be reduced by increasing the number of commitment phases or the number of phases can be reduced by adding more replicas. Each design choice expresses a *one-to-one function* to map plausible input points (i.e., a BFT protocol) to plausible output points (i.e., another BFT protocol) in the design space.

The Bedrock platform has three main practical uses:

- **BFT protocols analysis.** Bedrock can be used to analyze and navigate the evergrowing BFT landscape to principally compare and differentiate among BFT protocols. The Bedrock design space and its design choices provide fundamentally new insights into the strengths and weaknesses of existing BFT protocols.
- **BFT protocols implementation.** Within Bedrock, a wide range of BFT protocols, e.g., PBFT [72], SBFT [126], HotStuff [240], Kauri [194], Themis [142], Tendermint [63], Prime [24], PoE [129], CheapBFT [139], Q/U [5], FaB [182], and Zyzzyva [150], are implemented. The Bedrock implementation supports different stages of protocols, e.g., ordering, execution, view-change, and checkpointing. A domain-specific language (DSL) is

provided to rapidly prototype BFT protocols by specifying the protocol config, including the chosen value for each dimension in the design space, the list of roles, phases, states, and exchange messages of the protocol. Bedrock also includes a plugin manager to first, implement protocol-specific behaviors that can not be specified by the protocol config and second, enable users to add their own methods, dimensions, or values to support more protocols or to modify existing dimensions, e.g., add a new signature algorithm.

- **BFT protocols experimentation.** In addition to rapid prototyping, the unified deployment environment of Bedrock enables users to experimentally evaluate and compare different BFT protocols proposed in diverse settings and contexts under one unified platform. To our best knowledge, our paper presents the largest (and most varied) number of BFT protocols compared and experimented with within a single unified platform.

The paper makes the following contributions.

- A design space for BFT protocols, a set of design choices and possible design trade-offs are presented to help users analyze BFT protocols and understand how different protocols are related to each other.
- We present *Bedrock*, a platform that aims to unify BFT protocols. Bedrock derives valid protocols by combining different design choices in the design space.
- A wide range of BFT protocols can be implemented in Bedrock. The DSL specifications result in orders of magnitude reduction in code size compared to equivalent open-source implementations, greatly improving code readability and the ability to rapidly prototype protocols.
- The unified experimentation environment of Bedrock provides for the first time new opportunities to evaluate and compare different existing BFT protocols fairly and efficiently (e.g., identical programming language, used libraries, cryptographic tools, etc).

2 Bedrock Overview

System model. A BFT protocol runs on a network consisting of a set of nodes that may exhibit arbitrary, potentially malicious, behavior. BFT protocols use the State Machine Replication (SMR) algorithm [156, 212] where the system provides a replicated service whose state is mirrored across different deterministic replicas. At a high level, the goal of a BFT SMR protocol is to assign each client request an order in the global service history and execute it in that order [216]. In a BFT SMR protocol, all non-faulty replicas execute the same requests in the same order (*safety*) and all correct requests are eventually executed (*liveness*). In an asynchronous system, where replicas can fail, no consensus solutions guarantee both safety and liveness (FLP result) [114]. As a result, asynchronous consensus protocols rely on techniques such as randomization [45, 67, 118, 206], failure detectors [77, 177], hybridization/wormholes [85, 196] and partial synchrony [105, 108] to circumvent the FLP impossibility.

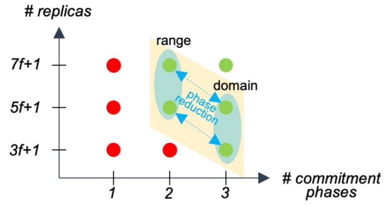


Figure 1: A simplified design space with two dimensions: number of replicas and number of commitment phases. Green dots (●) specify valid points (i.e., BFT protocols) while red dots (●) show invalid points (i.e., impossible protocols). A design choice, i.e., phase reduction, is a one-to-one transformation function that maps a protocol in its domain to another protocol in its range.

Bedrock assumes the partial synchrony model as it is used in most practical BFT protocols [72, 126, 150, 240]. In the partial synchrony model, there exists an unknown global stabilization time (GST), after which all messages between correct replicas are received within some unknown bound Δ. Bedrock further inherits the standard assumptions of existing BFT protocols. First, while there is no upper bound on the number of faulty clients, the maximum number of concurrent malicious replicas is assumed to be f. Second, replicas are connected via an unreliable network that might drop, corrupt, or delay messages. Third, the network uses point-to-point bi-directional communication channels to connect replicas. Fourth, the failure of replicas is independent of each other, where a single fault does not lead to the failure of multiple replicas. This can be achieved by either diversifying replica implementation (e.g., n-version programming) [37, 115] or placing replicas at different geographic locations (e.g., datacenters) [48, 109, 219, 230]. Finally, a strong adversary can coordinate malicious replicas and delay communication. However, the adversary cannot subvert cryptographic assumptions.

Usage model. Bedrock aims to help application developers analyze, implement, and experimentally evaluate BFT protocols within one unified platform and find the BFT protocol that fits the characteristics of their applications. To achieve this goal, the Bedrock platform makes available the design dimensions of BFT protocols and different design choices, i.e., trade-offs between dimensions, to application developers to tune. Figure 1 illustrates an example highlighting the relation between design space, dimensions, design choices, and protocols in Bedrock. For the sake of simplicity, we present only two dimensions of the design space, i.e., number of replicas and number of commitment phases (the design space of Bedrock consists of more than 10 dimensions as described in Section 3). Each dimension, e.g., number of replicas, can take different values, e.g., $3f+1$, $5f+1$, $7f+1$, etc. A BFT protocol is then a point in this design space, e.g., $(3, 3f+1)$. Note that each dimension not presented in this figure also takes a value, e.g., communication strategy is assumed to be pessimistic.

Moreover, only a subset of points is valid and represents BFT protocols. In Figure 1, green dots (●) specify valid points (i.e., BFT protocols) while red dots (●) show invalid points

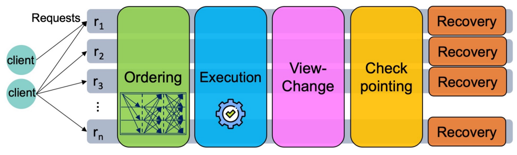


Figure 2: Different stages of replicas in a BFT protocol

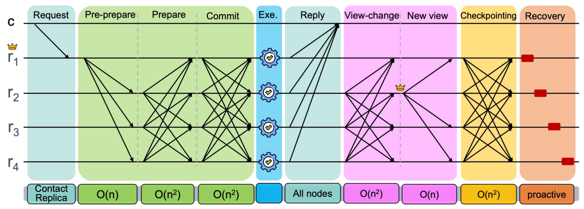


Figure 3: Different stages of PBFT protocol

(i.e., impossible protocols). For example, there is no (pessimistic) BFT protocol with $3f+1$ nodes that commits requests in a single commitment phase. A design choice (Section 4) is then a one-to-one function that maps each BFT protocol in its domain to another protocol in its range. For example, phase reduction (through redundancy) maps each protocol with $3f+1$ nodes and 3 phases of communication, e.g., PBFT [72], to a protocol with $5f+1$ nodes and 2 phases of communication, e.g., FaB [182] (assuming both protocols are pessimistic and follow clique topology). The domain and range of each design choice are a subset of BFT protocols in the design space.

BFT protocols structure. In a BFT protocol, as presented in Figure 2, clients communicate with a set of replicas that maintain a copy of the application state. A replica's lifecycle consists of ordering, execution, view-change, checkpointing, and recovery stages. The goal of the ordering stage is to establish agreement on a unique order among requests executing on the application state. In leader-based consensus protocols, a designated *leader* node proposes the order and, to ensure fault tolerance, needs to get agreement from a subset of the nodes, referred to as a *quorum*. In the execution stage, requests are applied to the replicated state machine. The view-change stage replaces the current leader. Checkpointing is used to garbage-collect data and enable trailing replicas to catch up, and finally, the recovery stage recovers replicas from faults by applying software rejuvenation techniques.

PBFT Protocol. To better illustrate the Bedrock design space, we give an overview of the PBFT protocol [71, 72] as a driving example. PBFT, as shown in Figure 3, is a leader-based protocol that operates in a succession of configurations called *views* [111, 112]. Each view is coordinated by a *stable* leader (primary) and the protocol *pessimistically* processes requests. In PBFT, the number of replicas, n, is assumed to be $3f+1$ and the ordering stage consists of pre-prepare, prepare, and commit phases. The pre-prepare phase assigns an order to the request, the prepare phase guarantees the uniqueness of the assigned order and the commit phase guarantees that the next leader can safely assign the order.

During a normal case execution of PBFT, clients send their signed request messages to the leader. In the pre-prepare phase, the leader assigns a sequence number to the request to determine the execution order of the request and multicasts a pre-prepare message to all *backups*. Upon receiving a valid pre-prepare message from the leader, each backup replica multicasts a prepare message to all replicas and waits for prepare messages from $2f$ different replicas (including the replica itself) that match the pre-prepare message. The goal of the prepare phase is to guarantee safety within the view, i.e., $2f$ replicas received matching pre-prepare messages from the leader replica and agree with the order of the request.

Each replica then multicasts a commit message to all replicas. Once a replica receives $2f+1$ valid commit messages from different replicas, including itself, that match the pre-prepare message, it commits the request. The goal of the commit phase is to ensure safety across views, i.e., the request has been replicated on a majority of non-faulty replicas. The second and third phases of PBFT follow the *clique* topology, i.e., have $O(n^2)$ message complexity. If the replica has executed all requests with lower sequence numbers, it executes the request and sends a reply to the client. The client waits for $f+1$ matching results from different replicas.

In the view change stage, upon detecting the failure of the leader of view v using timeouts, backups exchange view-change messages including requests that have been received by the replicas. After receiving $2f+1$ view-change messages, the designated stable leader of view $v+1$ (the replica with ID $= v+1 \bmod n$) proposes a new view message, including the list of requests that should be processed in the new view.

In PBFT, replicas periodically generate checkpoint messages and send them to all replicas. If a replica receives $2f+1$ matching checkpoint messages, the checkpoint is stable. PBFT includes a *proactive* recovery mechanism that periodically rejuvenates replicas one by one. PBFT uses either signatures [72] or MACs [71] for authentication. Using MACs, replicas need to send view-change-ack messages to the leader after receiving view-change messages. Since new view messages are not signed, these view-change-ack messages enable replicas to verify the authenticity of new view messages.

3 Design Space

In Bedrock, each BFT protocol can be analyzed along several dimensions. These dimensions (and values associated with each dimension) collectively help to define the overall design space of BFT protocols supported by Bedrock. The dimensions are categorized into four main families: *protocol structure* and *environmental settings* that present the core dimensions of BFT protocols and are shared among all BFT protocols, two optional *QoS features* including order-fairness and load balancing that a BFT protocol might support, and a set of *performance optimizations*, such as request pipelining, parallel execution, and trusted hardware, for tuning BFT protocols. Due to space limitations, the performance optimizations are discussed in Appendix A. In the rest of this section, we describe these families of dimensions in greater detail. As we describe each dimension, we prefix label them with "E" for environmental settings, "P" for protocol structure, etc. Hence, "E 1" refers to the first dimension in the environmental settings dimensions family.

This section is not meant to provide a fully exhaustive set of dimensions, but rather to demonstrate the overall methodology used to define dimensions usable in Bedrock.

3.1 Protocol Structure

Our first family of dimensions concerns customization of the protocol structure by Bedrock, which will further define the class of protocols permitted.

P 1. Commitment strategy. Bedrock supports BFT protocols that process transactions in either an optimistic, pessimistic, or robust manner. *Optimistic* BFT protocols make optimistic assumptions on failures, synchrony, or data contention and might execute requests without necessarily establishing consensus. An optimistic BFT protocol might make a subset of the following assumptions:

- a_1. The leader is non-faulty, assigns a correct order to requests and sends it to all backups, e.g., Zyzzyva [150],
- a_2. The backups are non-faulty and *actively* and *honestly* participate in the protocol, e.g., CheapBFT [139],
- a_3. All non-leaf replicas in a tree topology are non-faulty, e.g., Kauri [194],
- a_4. The workload is conflict-free and concurrent requests update disjoint sets of data objects, e.g., Q/U [5],
- a_5. The clients are honest, e.g., Quorum [34], and
- a_6. The network is synchronous (in a time window), and messages are not lost or delayed, e.g., Tendermint [62].

Optimistic protocols are classified into *speculative* and *non-speculative* protocols. In non-speculative protocols, e,g., SBFT [126] and CheapBFT [139], replicas execute a transaction only if the optimistic assumption holds. Speculative protocols, e.g., Zyzzyva [150] and PoE [129], on the other hand, optimistically execute transactions. If the assumption is not fulfilled, replicas might have to rollback the executed transactions. Optimistic BFT protocols improve performance in fault-free situations. If the assumption does not hold, the replicas, e.g., SBFT [126], or clients, e.g., Zyzzyva [150], detect the failure and use the fallback protocol.

Pessimistic BFT protocols, on the other hand, do not make any optimistic assumptions about failures, synchrony, or data contention. In pessimistic BFT protocols, replicas communicate to agree on the order of requests. Finally, *robust* protocols, e.g., Prime [24], Aardvark [83], R-Aliph [34], Spinning [229] and RBFT [35], go one step further and consider scenarios where the system is under attack by a very strong adversary.

In summary, BFT protocols demonstrate different performances in failure-free, low-failure, and under-attack situations. Optimistic protocols deliver superior performance in failure-free situations. However, in the presence of failure, their performance is significantly reduced, especially when the system is under attack. On the other hand, pessimistic protocols provide high performance in failure-free situations and are able to handle low failures with acceptable overhead. However, they show poor performance when the system is under attack. Finally, robust protocols are designed for under-attack situations and demonstrate moderate performance in all three situations.

P 2. Number of commitment phases. The number of commitment (ordering) phases or *good-case latency* [12] of a BFT SMR protocol is the number of phases needed for all non-faulty replicas to commit when the leader is non-faulty, and the network is synchronous. We consider the number of commitment phases from the first time a replica (typically the leader) receives a request to the first time any participant (i.e., leader, backups, client) learns the commitment of the request, e.g., PBFT executes in 3 phases.

P 3. View-change. BFT protocols follow either the *stable leader* or the *rotating leader* mechanism to replace the current leader. The stable leader mechanism [72, 126, 150, 182] replaces the leader when the leader is suspected to be faulty by other replicas. In the rotating leader mechanism [20, 64, 73–75, 83, 121, 133, 148, 155, 229, 230, 240], the leader is replaced periodically, e.g., after a single attempt, insufficient performance, or an epoch (multiple requests).

Using the stable leader mechanism, the view-change stage becomes more complex. However, the routine is only executed when the leader is suspected to be faulty. On the other hand, the rotating leader mechanism requires ensuring view synchronization frequently (whenever the leader is rotated). Rotating the leader has several benefits, such as balancing load across replicas [43, 44, 229], improving resilience against slow replicas [83], and minimizing communication delays between clients and the leader [109, 181, 230].

P 4. Checkpointing. The checkpointing mechanism is used to first, garbage-collect data of completed consensus instances to save space and second, restore in-dark replicas (due to network unreliability or leader maliciousness) to ensure all non-faulty replicas are up-to-date [72, 100, 129]. Checkpointing is typically initiated after a fixed window in a decentralized manner without relying on a leader [72].

P 5. Recovery. When there are more than f failures, BFT protocols, apart from some exceptions [81, 171], completely fail and do not give any guarantees on their behavior [100]. BFT protocols perform recovery using *reactive* or *proactive* mechanisms (or a combination [220]). Reactive recovery mechanisms detect faulty replica behavior [132] and recover the replica by applying software rejuvenation techniques [92, 136] where the replica reboots, reestablishes its connection with other replicas and clients, and updates its state. On the other hand, proactive recovery mechanisms recover replicas in periodic time intervals. Proactive mechanisms do not require any fault detection techniques; however, they might unnecessarily recover non-faulty replicas [100]. During recovery, a replica is unavailable. A BFT protocol can rely on $3f+2k+1$ replicas to improve resilience and availability during recovery where k is the maximum number of servers that rejuvenate concurrently [220]. To prevent attackers from disrupting the recovery process, each replica requires a trusted component, e.g., secure coprocessor [71], a synchronous wormhole [228] or a virtualization layer [102, 208], that remains operational even if the attacker controls the replica and a read-only memory that an attacker cannot manipulate. The memory content remains persistent (e.g., on disk) across machine reboots and includes all information needed for bootstrapping a correct replica after restart [100].

P 6. **Types of clients.** Bedrock supports three types of clients: requester, proposer, and repairer. *Requester* clients perform a basic functionality and communicate with replicas by sending requests and receiving replies. A requester client may need to verify the results by waiting for a number of matching replies, e.g., $f+1$ in PBFT [72], $2f+1$ in PoE [129] and PBFT (for read-only requests) [71], or $3f+1$ is Zyzzyva [150]. Using trusted components, e.g., Troxy [168], or threshold signatures, e.g., SBFT [126], the client does not even need to wait for and verify multiple results from replicas. Clients might also play the *proposer* role by proposing a sequence number (acting as the leader) for its request [5, 125, 178, 180]. *Repairer* clients, on the other hand, detect the failure of replicas, e.g., Zyzzyva [150], or even change the protocol configuration, e.g., Scrooge [213], Abstract [34], and Q/U [5].

3.2 Environmental Settings

Environmental settings, broadly speaking, encompass the deployment environment for a BFT protocol. These input parameters help scope the class of BFT protocols that can be supported to fit each deployment environment best.

E 1. **Number of replicas.** The first dimension concerns selecting BFT protocols based on the number of replicas (i.e., network and quorum size) used in a deployment. In the presence of f malicious failures, BFT protocols require at least $3f+1$ replicas to guarantee safety [56, 57, 88, 108, 163]. Using trusted hardware, the malicious behavior of replicas is restricted and safety can be guaranteed using $2f+1$ replicas [81, 87, 89, 208, 230, 230, 231]. Similarly, leveraging new hardware capabilities or using message-and-memory models the required number of replicas can be reduced to $2f+1$ [15–17]. On the other hand, the number of communication phases can be reduced by increasing the number of replicas to $5f+1$ [182] (its proven lower bound, $5f-1$ [12, 154]) or $7f+1$ [218]. A BFT protocol might also optimistically assume the existence of a quorum of $2f+1$ active non-faulty replicas (put f replicas as passive) to establish consensus [101, 139]. Using both trusted hardware and active/passive replication, the quorum size is further reduced to $f+1$ during failure-free situations [101, 102, 139].

E 2. **Communication topology.** Bedrock allows users to analyze BFT protocols based on communication topologies, including: (1) the star topology where communication is strictly from a designated replica, e.g., the leader, to all other replicas and vice-versa, resulting in linear message complexity [150, 240], (2) the clique topology where all (or a subset of) replicas communicate directly with each other resulting in quadratic message complexity [72], (3) the tree topology where the replicas are organized in a tree with the leader placed at the root, and at each phase, a replica communicates with either its child replicas or its parent replica, causing logarithmic message complexity [147, 148, 194], or (4) the chain topology where replicas construct a pipeline and each replica communicates with its neighbor replicas [34].

E 3. **Authentication.** Participants authenticate their messages to enable other replicas to verify a message's origin. Bedrock support both signatures, e.g., RSA [210], and authenticators [72], i.e., MACs [226]. Constant-sized threshold signatures [67, 214] have also been used to reduce the size of a set (quorum) of signatures. A protocol might even use different techniques (i.e., signatures, MACs) in different stages to authenticate messages sent by clients, sent by replicas in the ordering stage, and sent by replicas during view-change.

E 4. **Responsiveness, synchronization, and timers.** A BFT protocol is *responsive* if its normal case commit latency depends only on the actual network delay needed for replicas to process and exchange messages rather than any (usually much larger) predefined upper bound on message transmission delay [33, 200, 201, 215]. Responsiveness might be sacrificed in different ways. First, rotating the leader, the new leader might need to wait for a predefined time before initiating the next request to ensure that it receives the decided value from all non-faulty but slow replicas, e.g., Tendermint [155] and Casper [65]. Second, optimistically assuming all replicas are non-faulty, replicas (or clients) need to wait for a predefined upper bound to receive messages from all replicas, e.g., SBFT [126] and Zyzzyva [150].

BFT protocols need to guarantee that all non-faulty replicas will eventually be synchronized to the same view with a non-faulty leader enabling the leader to collect the decided values in previous views and making progress in the new view [59, 190, 191]. This is needed because a quorum of $2f+1$ replicas might include f Byzantine replicas and the remaining f "slow" non-faulty replicas might stay behind (i.e., in-dark) and not even advance views at all. *View synchronization* can be achieved by integrating the functionality with the core consensus protocol, e.g., PBFT [72], or assigning a distinct synchronizer component, e.g., Pacemaker in HotStuff [240], and hardware clocks [6].

Depending on the environment, network characteristics, and processing strategy, BFT protocols use different timers to ensure responsiveness and synchronization. Protocols can be configured with the following timers by Bedrock.

τ_1. Waiting for reply messages, e.g., Zyzzyva [150],

τ_2. Triggering (consecutive) view-change, e.g., PBFT [72],

τ_3. Detecting backup failures, e.g., SBFT [126],

τ_4. Quorum construction in an ordering phase, e.g., prevote and precommit timeouts in Tendermint [62],

τ_5. View synchronization, e.g., Tendermint [62],

τ_6. Finishing a (preordering) round, e.g., Themis [142],

τ_7. Performance check (heartbeat), e.g., Aardvark [83], and

τ_8. Atomic recovery (watchdog timer) to periodically hand control to a recovery monitor [70], e.g., PBFT [71].

3.3 Quality of Service

There are some optional QoS features that Bedrock can support. We list two example dimensions.

Q 1. **Order-fairness.** Order-fairness deals with preventing adversarial manipulation of request ordering [39, 68, 142, 143, 152, 153, 243]. Order-fairness is defined as: "if a large number of replicas receives a request t_1 before another request t_2, then t_1 should be ordered before t_2" [143]. Order-fairness has been partially addressed using different techniques: (1) monitoring the leader to ensure it does not initiate two new requests from the same client before initiating an old request of another client, e.g., Aardvark [83], (2) adding a preordering phase, e.g., Prime [24], where replicas order the received requests locally and share their own ordering with each other, (3) encrypting requests and revealing the contents only once their ordering is fixed [32, 66, 185, 223], (4) reputation-based systems [32, 96, 149, 166] to detect unfair censorship of specific client requests, and (5) providing opportunities for every replica to propose and commit its requests using fair election [9, 32, 121, 144, 166, 200, 237].

Q 2. **Load balancing.** The performance of fault-tolerant protocols is usually limited by the computing and bandwidth capacity of the leader [18, 21, 53, 78, 187, 188, 194, 234]. The leader coordinates the consensus protocol and multicasts/-collects messages to all other replicas in different protocol phases. Load balancing is defined as distributing the load among the replicas of the system to balance the number of messages any single replica has to process.

Load balancing can be partially achieved using the rotating leader mechanism, multi-layer, or multi-leader BFT protocols. When the rotating leader mechanism is used, one replica (leader) still is highly loaded in each consensus instance. In multi-layer BFT protocols [25, 131, 172, 193, 195], the load is distributed between the leaders of different clusters. However, the system still suffers from load imbalance between the leader and backup replicas in each cluster. In multi-leader protocols [22, 30, 36, 130, 222, 232], all replicas can initiate consensus to partially order requests in parallel. However, slow replicas still affect the global ordering of requests.

4 Design Choices Landscape

Given a set of specified dimension values in Section 3, each protocol represents a point in the Bedrock design space. In this section, using PBFT as a driving example, as illustrated in Section 2, we demonstrate how different points in the design space lead to different trade-offs.

4.1 Expanding the Design Choices of PBFT

Using PBFT and our design dimensions as a baseline, we illustrate a series of design choices that expose different trade-offs BFT protocols need to make. Each design choice acts as a one-to-one function that maps each valid input point (i.e., a protocol) to another valid output point in the design space.

Design Choice 1. ***(Linearization).*** This function explores a trade-off between communication topology and communication phases. The function takes a quadratic communication phase, e.g., prepare or commit in PBFT, and splits it into two linear phases: one phase from all replicas to a collector (typically the leader) and one phase from the collector to all replicas, e.g., SBFT [126], HotStuff [240]. The output protocol requires (threshold) signatures for authentication. The collector collects a quorum of (typically $n-f$) *signatures* from other replicas and broadcasts its message including the signatures as a certificate of having received the required signatures to every replica. Using threshold signatures [66, 67, 207, 214] the collector message size becomes constant.

Design Choice 2. ***(Phase reduction through redundancy).*** This function explores a trade-off between the number of ordering phases and the number of replicas. The function transforms a protocol with $3f+1$ replicas and 3 ordering phases (i.e., one linear, two quadratic), e.g., PBFT, to a fast protocol with $5f+1$ replicas and 2 ordering phases (one linear, one quadratic), e.g., FaB [182]. In the second phase of the protocol, matching messages from a quorum of $4f+1$ replicas are required. Recently, $5f-1$ has been proven as the lower bound for two-step Byzantine consensus [12, 154]. The intuition behind the $5f-1$ lower bound is that in an authenticated model, when replicas detect leader equivocation and initiate view-change, they do not include view-change messages coming from the malicious leader, reducing the maximum number of faulty messages to $f-1$ [12, 154].

Design Choice 3. ***(Leader rotation).*** This function replaces the stable leader with the rotating leader mechanism, e.g., HotStuff [240], where the rotation happens after each request or epoch or due to low performance (as discussed in P 3). This function eliminates the view-change stage and adds a quadratic phase or two linear phases (using the linearization function) to the ordering stage to ensure that the new leader is aware of the correct state of the system.

Design Choice 4. ***(Non-responsive leader rotation).*** This function replaces the stable leader mechanism with the rotating leader mechanism *without* adding a new ordering phase (in contrast to design choice 3) while sacrificing responsiveness. The new leader optimistically assumes that the network is synchronous and waits for a predefined known upper bound Δ (Timer τ_5) before initiating the next request. This is needed to ensure that the new leader is aware of the highest assigned order to the requests, e.g., Tendermint [63, 155] and Casper [65].

Design Choice 5. ***(Optimistic replica reduction).*** This function reduces the number of involved replicas in consensus from $3f+1$ to $2f+1$ while optimistically assuming all $2f+1$ replicas are non-faulty (assumption P 1, a_2). In each phase of a BFT protocol, matching messages from a quorum of $2f+1$ replicas is needed. If a quorum of $2f+1$ non-faulty replicas is identified, they can order (and execute) requests without the participation of the remaining f replicas. Those f replicas remain passive and are needed if any of the active replicas become faulty [101, 139]. Note that n is still $3f+1$.

Design Choice 6. ***(Optimistic phase reduction).*** Given a *linear* BFT protocol, this function optimistically eliminates two linear phases (i.e., the equivalence of a single quadratic prepare pahse) assuming all replicas are non-faulty, e.g., SBFT [126]. The leader (collector) waits for signed messages from all $3f+1$ replicas in the second phase of ordering, combines signatures and sends a signed message to all replicas. Upon receiving the signed message from the leader, each replica ensures that all non-faulty replicas have received the request and agreed with the order. As a result, the third phase of communication can be omitted and replicas can directly commit the request. If the leader has not received $3f+1$ messages after a predefined time (timer τ_3), the protocol fallbacks to its slow path and runs the third phase of ordering.

Design Choice 7. ***(Speculative phase reduction).*** This function, similar to the previous one, optimistically eliminates two linear phases of the ordering stage assuming that non-faulty replicas construct the quorum of responses, e.g., PoE [129]. The main difference is that the leader waits for signed messages from only $2f+1$ replicas in the second phase of ordering and sends a signed message to all replicas. Upon receiving a message signed by $2f+1$ replicas from the leader, each replica speculatively executes the transaction, optimistically assuming that either (1) all $2f+1$ signatures are from non-faulty replicas or (2) at least $f+1$ non-faulty replicas received the signed message from the leader. If (1) does not hold, other replicas receive and execute transactions during the view-change. However, if (2) does not hold, the replica might have to rollback the executed transaction.

Design Choice 8. ***(Speculative execution).*** This function eliminates the prepare and commit phases while optimistically assuming that all replicas are non-faulty (assumptions P 1, a_1 and a_2), e.g., Zyzzyva [150]. Replicas speculatively execute transactions upon receiving them from the leader. If the client does not receive $3f+1$ matching replies after a predefined time (timer τ_1) or it receives conflicting messages, the (repairer) client detects failure and communicates with replicas to receive $2f+1$ commit messages.

Design Choice 9. ***(Optimistic conflict-free).*** If requests of different clients are conflict-free (assumption P 1, a_4), there is no need for a total order among all transactions. This function eliminates all ordering phases while optimistically assuming that requests are conflict-free and all replicas are non-faulty. The client becomes the *proposer* and sends its request to all (or a quorum of) replicas where replicas execute the requests without any communication [5, 94].

Design Choice 10. ***(Resilience).*** This function increases the number of replicas by $2f$ enabling the protocol to tolerate f more failure with the same safety guarantees. In particular, optimistic BFT protocols that assume all $3f+1$ replicas are non-faulty (quorum size is also $3f+1$) tolerate zero failures. By increasing the number of replicas to $5f+1$ replicas, such BFT protocols can provide the same safety guarantees with quorums of size $4f+1$ while tolerating f failures, e.g., Zyzzyva5 [150], Q/U [5]. Similarly, a protocol with the network size of $5f+1$ can tolerate f more faulty replicas by increasing the network size to $7f+1$ [218].

This function can also provide high availability during the (proactive) recovery stage by increasing the number of replicas by $2k$ (the quorum size by k) where k is the maximum number of servers that recover concurrently [220].

Design Choice 11. ***(Authentication).*** This function replaces MACs with signatures for a given stage. Signatures are typically more costly than MACs. However, in contrast to MACs, signatures provide non-repudiation and are not vulnerable to MAC-based attacks from malicious clients. If a protocol follows the star communication topology where a replica needs to include a quorum of signatures as a proof of its messages, e.g., HotStuff [240], k signatures can be replaced with a threshold signature. In such protocols MACs cannot be used since MACs do not provide non-repudiation.

Design Choice 12. ***(Robust).*** This function makes a pessimistic protocol robust by adding a preordering stage to the protocol, e.g., Prime [24]. In the preordering stage and, upon receiving a request, each replica locally orders and broadcasts the request to all other replicas. All replicas then acknowledge the receipt of the request in an all-to-all communication phase and add the request to their local request vector. Replicas periodically share their vectors with each other. The robust function provides (partial) fairness as well. Robustness has also been addressed in other ways, e.g., using the leader rotation and a blacklisting mechanism in Spinning [229] or isolating the incoming traffic of different replicas, and checking the performance of the leader in Aardvark [83].

Design Choice 13. ***(Fair).*** This function transforms an unfair protocol, e.g., PBFT, into a fair protocol by adding a preordering phase to the protocol. In the preordering phase, clients send requests to all replicas, and once a round ends (timer τ_6), each replica sends a batch of requests in the received order to the leader. The leader then initiates consensus on the requests following the order of requests in the received batches. Depending on the order-fairness parameter γ ($0.5<\gamma\leq 1$) that defines the fraction of replicas receiving the requests in that specific order, at least $4f+1$ replicas ($n>\frac{4f}{2\gamma-1}$) replicas are needed to provide order-fairness [142, 143] [1].

[1]With $3f+1$ replicas, as shown in [142], order-fairness requires a synchronized clock [243] or does not provide censorship resistance [152].

Table 1: Comparing selected BFT protocols based on different dimensions of Bedrock design space

Protocol	E1. Nodes	E2. Topo.	E3. Auth.	E4. Timers	P1. Strategy	P2. Phases	P3. V-change	P5. Rec.	P6. Client	Q1. Fair.	Q2. Load.	Design Choices
PBFT [72]	$3f+1$	clique	MAC ‖ Sign	τ_1, τ_2, τ_8	pessimistic	3	stable	pro.	Req.	☐	☐	(11)
Zyzzyva [150]	$3f+1$	star	MAC ‖ Sign	τ_1, τ_2	optimistic (spec): a_1, a_2	1 (3)	stable	-	Rep.	☐	☐	8, (11)
Zyzzyva5 [150]	$5f+1$	star	MAC ‖ Sign	τ_1, τ_2	optimistic (spec): a_1	1 (3)	stable	-	Rep.	☐	☐	8, 10, (11)
PoE [129]	$3f+1$	star	MAC ‖ T-Sign	τ_1, τ_2	optimistic (spec): a_2	3	stable	-	Req.	☐	☐	1, 7, 11
SBFT [126]	$3f+1$	star	T-Sign	τ_1, τ_2, τ_3	optimistic: a_2	3 (5)	stable	-	Req.	☐	☐	1, 6, 11
HotStuff [240]	$3f+1$	star	T-Sign	τ_1, τ_2	pessimistic	7	rotating	-	Req.	☐	☐	1, 3, 11
Tendermint [63]	$3f+1$	clique	Sign	$\tau_1, \tau_2, \tau_5, \tau_6$	optimistic: a_6	3	rotating	-	Req.	☐	☐	4, 11
Themis [142]	$4f+1$	star	T-Sign	τ_1, τ_2, τ_6	pessimistic	$1+7$	rotating	-	Req.	■	☐	1, 3, 13, 11
Kauri [194]	$3f+1$	tree	T-Sign	τ_1, τ_2	optimistic: a_3	$7h$	stable*	-	Req.	☐	■	(3), 14, 11
CheapBFT [139]	$2f+1$	clique	MAC	τ_1, τ_2	optimistic: a_2	3	stable	-	Req.	☐	☐	5
FaB [182]	$5f+1$	clique	(Sign)	τ_1, τ_2	pessimistic	2	stable	-	Req.	☐	☐	2
Prime [24]	$3f+1$	clique	Sign	$\tau_1, \tau_2, \tau_6, \tau_7$	robust	6	stable	-	Req.	◧	☐	11, 12
Q/U [5]	$5f+1$	star	MAC	τ_1, τ_2	optimistic: a_4, a_5	1 (3)	stable	-	Rep.	☐	☐	9, 10
FLB	$5f-1$	clique	Sign	τ_1, τ_2	pessimistic	2	stable	-	Req.	☐	☐	1, 2, 11
FTB	$5f-1$	tree	T-Sign	τ_1, τ_2	optimistic: a_3	$3h$	stable	-	Req.	☐	■	1, 2, 14, 11

Hint: "T-Sign": threshold signatures, "Req": requester client, "Rep": repairer client, "Pro": proactive recovery. The number of phases in the slow path of protocols is shown in parentheses. While Kauri is implemented on top of HotStuff, it does not use rotating leaders. Prime provides partial fairness.

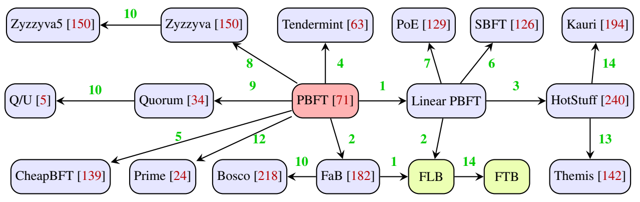


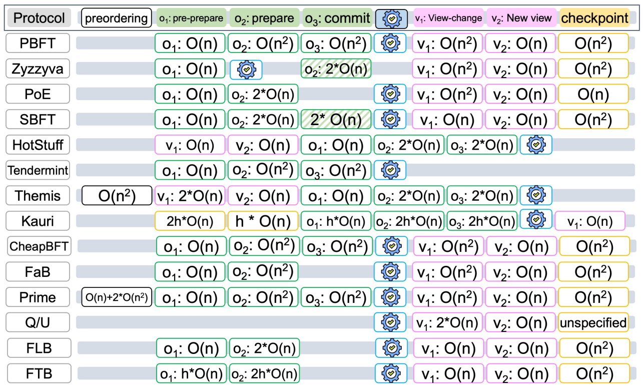
Figure 4: Derivation of protocols from PBFT using design choices

Design Choice 14. ***(Tree-based LoadBalancer).*** This function explores a trade-off between the communication topology and load balancing where load balancing is supported by organizing replicas in a tree topology, with the leader at the root, e.g., Kauri [194]. This function splits a linear communication phase into h phases where h is the tree's height and each replica uniformly communicates with its child/parent replicas in the tree. The protocol optimistically assumes all non-leaf replicas are non-faulty (assumption P 1, a_3). Otherwise, the tree is reconfigured (i.e., view change).

4.2 Deriving and Evolving Protocols

Figure 4 demonstrates the derivation of a wide spectrum of BFT protocols from PBFT using design choices. Table 1 provides insights into how each BFT protocol maps into the Bedrock design space. The table also presents the design choices used by each BFT protocol. A detailed explanation of protocols is presented in Appendix B.

Figure 5 focuses on different stages of replicas and demonstrates the communication complexity of each stage. The figure presents: (1) the preordering phases used in Themis and Prime, (2) the three ordering phases, e.g., pre-prepare, prepare or commit in PBFT (labeled by o_1, o_2, and o_3), (3) the



Protocol	preordering	o_1: pre-prepare	o_2: prepare	o_3: commit	⚙	v_1: View-change	v_2: New view	checkpoint
PBFT		o_1: O(n)	o_2: O(n^2)	o_3: O(n^2)	⚙	v_1: O(n^2)	v_2: O(n)	O(n^2)
Zyzzyva		o_1: O(n)	⚙	o_2: 2*O(n)		v_1: O(n^2)	v_2: O(n)	O(n^2)
PoE		o_1: O(n)	o_2: 2*O(n)		⚙	v_1: O(n^2)	v_2: O(n)	O(n)
SBFT		o_1: O(n)	o_2: 2*O(n)	2* O(n)	⚙	v_1: O(n)	v_2: O(n)	O(n^2)
HotStuff		v_1: O(n)	v_2: O(n)	o_1: O(n)	o_2: 2*O(n)	o_3: 2*O(n)	⚙	
Tendermint		o_1: O(n)	o_2: O(n^2)	o_3: O(n^2)	⚙			
Themis	O(n^2)	v_1: 2*O(n)	v_2: O(n)	o_1: O(n)	o_2: 2*O(n)	o_3: 2*O(n)	⚙	
Kauri		2h*O(n)	h * O(n)	o_1: h*O(n)	o_2: 2h*O(n)	o_3: 2h*O(n)	⚙	v_1: O(n)
CheapBFT		o_1: O(n)	o_2: O(n^2)	o_3: O(n^2)	⚙	v_1: O(n^2)	v_2: O(n)	O(n^2)
FaB		o_1: O(n)	o_2: O(n^2)		⚙	v_1: O(n^2)	v_2: O(n)	O(n^2)
Prime	O(n)+2*O(n^2)	o_1: O(n)	o_2: O(n^2)	o_3: O(n^2)	⚙	v_1: O(n^2)	v_2: O(n)	O(n^2)
Q/U					⚙	v_1: 2*O(n)	v_2: O(n)	unspecified
FLB		o_1: O(n)	o_2: 2*O(n)		⚙	v_1: O(n)	v_2: O(n)	O(n^2)
FTB		o_1: h*O(n)	o_2: 2h*O(n)		⚙	v_1: O(n)	v_2: O(n)	O(n^2)

Figure 5: Overview of BFT protocols

execution stage, (4) the view-change stages consisting of view-change and new-view phases (labeled by v_1 and v_2), and (5) the checkpointing stage. As can be seen, some protocols do not have all three ordering phases, i.e., using different design choices, the number of ordering phases is reduced. The dashed boxes present the slow-path of protocols, e.g., the third ordering phase of SBFT is used only in its slow-path. Finally, the order of stages might be changed. For example, HotStuff runs view-change (leader rotation) for every single message and this phase takes place at the beginning of a consensus instance to synchronize nodes within a view.

These case studies demonstrate the value of Bedrock in providing a unified platform for analyzing a range of existing BFT protocols. Note that the Bedrock platform enables users to implement new dimensions or design choices. For example, recently directed acyclic graph (DAG)-based BFT protocols [39, 97, 98, 116, 122, 141, 221, 236] have emerged

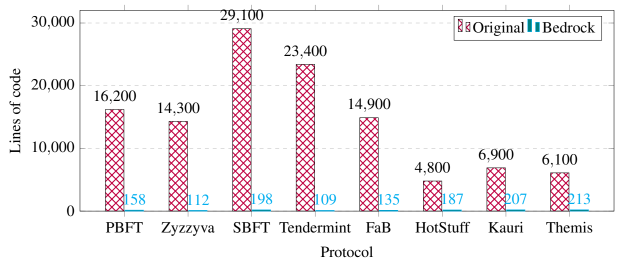


Figure 6: Lines of code in Bedrock and the original implementation

as an efficient way of establishing consensus. In DAG-based protocols and in each round, replicas independently send their own block of transactions as well as references to $2f+1$ received blocks (in the previous round) to other replicas in parallel. The references that blocks carry then become the backbone of a causally ordered DAG structure. DAG-based protocols provide higher throughput by separating transaction dissemination (by all replicas) from ordering. One can evolve PBFT to a DAG-based protocol in three steps (using three design choices); linearization, pipelining, and parallelization, with some minor modifications. Linearization makes PBFT linear (design choice **1**), pipelining enables a node to piggyback the messages of a new consensus instance on the second round messages of the previous instance (as it is used in Chained-HotStuff [4]), and parallelization enables multiple replicas to propose messages in parallel (as used in multi-leader protocols [130, 222], discussed in Appendix A, **3**).

Bedrock's utility can go beyond an analysis platform towards a *discovery* tool as well. Appendix C demonstrates two BFT protocols (FLB and FTB) uncovered using Bedrock.

5 Implementation

Bedrock enables users, e.g., application developers, to implement and evaluate different BFT protocols. Bedrock is implemented in Java. The modular design of Bedrock enables a fair and efficient evaluation of BFT protocols using identical libraries, cryptographic functions, etc. The Bedrock platform consists of four main components: the core unit, the state manager, the plugin manager, and the coordination unit.

The core unit defines entities, e.g., clients and nodes, and maintains the application logic and application data. Client transactions are executed using the application logic resulting in updating the data. Entities track the execution of requests through various state variables, e.g., view and sequence number. Within the core unit, different workloads and benchmarks can be defined. Client requests can be initiated using a constant interval or a dynamic interval updated based on a moving average of response times. Different utility classes, such as `Timekeeper` to handle timers, and `BenchmarkManager` to measure and report results are also defined within the core unit.

The state manager enables the core unit to track the states and transitions of each entity according to the utilized BFT protocol, e.g., different stages of a replica or different phases of consensus. Bedrock defines a domain-specific language (DSL) to rapidly prototype BFT protocols. The DSL code written in the protocol config defines different dimensions and the chosen value for each dimension, the list of roles, phases, states, exchange messages, quorum conditions of the protocol, and also, the list of protocol-specific plugins required to run the protocol. The `EO-YAML` and `Apache Commons Lang` libraries are used for parsing, loading, and holding the protocol config data. Appendix E demonstrates the PBFT code using the DSL. The protocol config greatly reduces the effort needed to write a BFT protocol. Figure 6 compares the lines of code in the original open-source implementation of several known protocols and their implementation in Bedrock., e.g., the original Zyzzyva source code includes more than 14000 lines while its config in Bedrock is only 112 lines[2]. Overall, Bedrock results in orders of magnitude reduction in code size. Note that, in addition to the config file, the implementation of each protocol uses a set of plugins defined in Bedrock, as explained in the next part. Chained-HotStuff, as a protocol that uses the most plugins (five), requires only 412 more lines of code to implement its five plugins, several of them are shared with multiple protocols.

The plugin manager serves two purposes. First, it enables the implementation of protocol-specific behaviors that cannot be handled by the protocol config defined in the state manager. For example, the speculative execution in Zyzzyva [150] or handling view-change without using a different process or states in Tendermint [155]. Second, it enables Bedrock users to define their own dimensions/values to support more protocols or to update existing dimensions without requiring changes to the platform code or rebuilding the platform binaries. For example, if a developer wants to use a new digest or signature algorithm for an existing or a new protocol, the algorithm can be implemented within a plugin.

Four types of plugins have been defined in the current version of Bedrock. *Role* plugins that define specific behavior for a certain role in a specific sequence number, view number, state, etc., e.g., message dissemination by the primary node in CheapBFT [139] where nodes are divided into active and passive nodes. *Message* plugins that define specific methods to process incoming or outgoing messages, e.g., perform digest validation. *Transition* plugins that specify an action to be performed during or after a state transition, e.g., how to process checkpoint messages. *Pipeline* plugins that enable manipulating the flow of messages, e.g., the messages of a new consensus instance are piggybacked on the second round messages of the previous instance [194, 240] (as discussed in Appendix A, O **2**)

The coordination unit manages the run-time execution of Bedrock. The coordination unit consists of a coordinator and a set of executors. The coordinator manages the benchmark process and setups all entities by initializing replicas and clients, sending config parameters to executors, enabling plugins to run additional initialization steps, starting and stopping exe-

[2] We count only the lines of source code related to the core consensus protocol and not the applications or the utilized libraries.

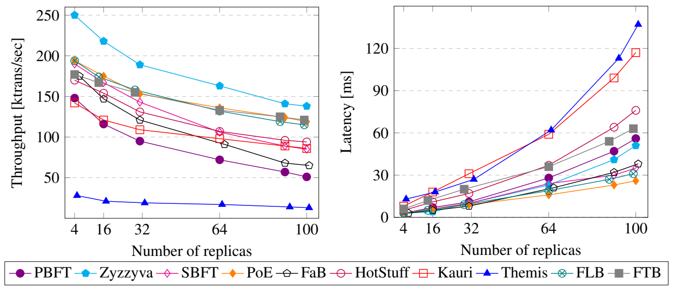


Figure 7: Performance with different number of replicas

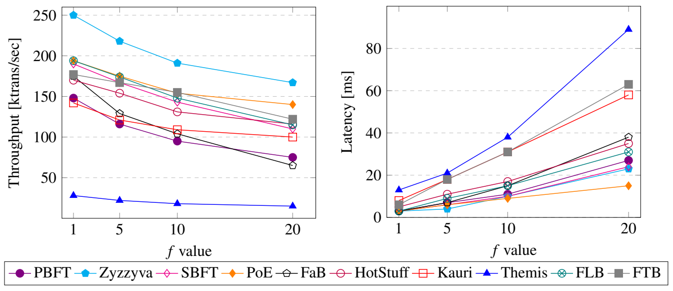


Figure 8: Performance with different f value

cution threads, and reporting results. The executors, on the other hand, run the utilized BFT protocol.

The data (e.g., messages, requests, blocks) for the events and messages transmitted between nodes and clients is defined using the `Google Protocol Buffers` syntax and then compiled using the `protoc` tool.

6 Experimental Evaluation

Our evaluation studies the practical impact of the design dimensions and the exposed trade-offs presented as design choices on the performance of BFT protocols. The goal is to test the capability of the Bedrock design space to analyze the performance of different protocols that were proposed in diverse settings and different contexts under one unified platform. We evaluate the performance of BFT protocols in typical experimental scenarios used for existing BFT protocols and permissioned blockchains, including (1) varying the number of replicas, (2) under a backup failure, (3) multiple request batch sizes, and (4) a geo-distributed setup.

All protocols listed in Table 1 are implemented in Bedrock. Using the platform, we also experimented with many new protocols resulting from the combination of design choices. Due to space limitations, we present the performance evaluation of a subset of protocols. In particular, we evaluate PBFT, Zyzzyva, SBFT, FaB, PoE, (Chained) HotStuff, Kauri, Themis, and two of the more interesting new variants (FLB and FTB). This set of protocols enables us to see the impact of design choices **1**, **2**, **3**, **6**, **7**, **8**, **10**, **11**, **13**, and **14** (discussed in Section 4). We also use the out-of-order processing technique for protocols with a stable leader and the request pipelining technique for protocols with a rotating leader. In our experiments, Kauri and FTB are deployed on trees of height 2 and the order-fairness parameter γ of Themis is considered to be 1 (i.e., $n = 4f + 1$). We use 4 as the base pipelining stretch for both Kauri and FTB and change it depending on the batch size and deployment setting (local vs. geo-distributed).

The experiments were conducted on the Amazon EC2 platform. Each VM is a c4.2xlarge instance with 8 vCPUs and 15GB RAM, Intel Xeon E5-2666 v3 processor clocked at 3.50 GHz. When reporting throughput, we use an increasing number of client requests until the end-to-end throughput is saturated and state the throughput and latency just below saturation. The results reflect end-to-end measurements from the clients. Clients execute in a closed loop. We use micro-benchmarks commonly used to evaluate BFT systems, e.g., BFT-SMaRt. The results are the average of five runs.

6.1 Fault Tolerance and Scalability

In the first set of experiments, we evaluate the performance of the protocols by increasing the number of replicas n (each runs on a separate VM) from 4 to 100 in a failure-free situation. For some protocols, the smallest network size might differ, e.g., FaB requires $5f + 1 = 6$ replicas. We use a batch size of 400 (we discuss this choice later) and a workload with client request/reply payload sizes of 128/128 byte. Figure 7 reports the results.

Zyzzyva shows the highest throughput among all protocols in small networks due to its optimistic ordering stage (design choice **8**). However, as n increases, its throughput significantly reduces as clients need to wait for reply from *all* replicas. Increasing the number of replicas also has a large impact on PBFT and FaB (65% and 63% reduction, respectively) due to their quadratic message complexity.

On the other hand, the throughput of Kauri and FTB is less affected (31% and 32% reduction, respectively) by increasing n because of their tree topology (design choice **14**) that reduced the bandwidth utilization of each replica. Similarly, PoE, SBFT and HotStuff incur less throughput reduction (39%, 55% and 45% respectively) compared to PBFT and FaB due to their linear message complexity (design choice **1**). It should be noted that in Bedrock, chained HotStuff has been implemented using the pipelining technique. Hence, the average latency of requests has been reduced. In comparison to HotStuff, SBFT has slightly lower throughput in large networks (e.g., 8% lower when $n = 100$) because the leader waits for messages from all replicas. SBFT, on the other hand, shows higher throughput compared to HotStuff in smaller networks (e.g., 12% higher when $n = 4$) due to its fast ordering stage (design choice **6**). PoE demonstrates higher throughput compared to both SBFT and HotStuff, especially in larger networks (e.g., 39% higher than SBFT and 26% higher than HotStuff when $n = 100$). This is expected because, in PoE, the leader does not need to wait for messages from all replicas and optimistically combines signatures from $2f + 1$ replicas (design choice **7**). Compared to PBFT, while HotStuff shows better throughput (e.g., 48% higher when $n = 64$), the latency of PBFT is lower (e.g., 32% lower when $n = 64$). One reason behind the high latency of HotStuff is its extra communication round (design choice **3**).

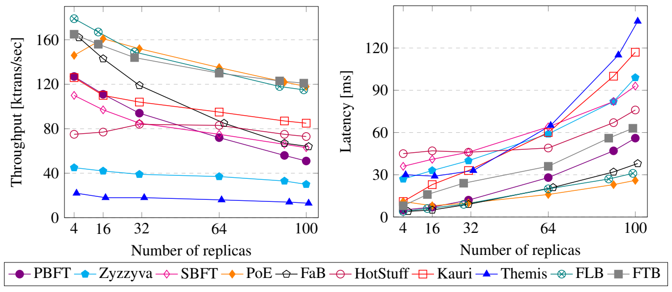


Figure 9: Performance with faulty backups

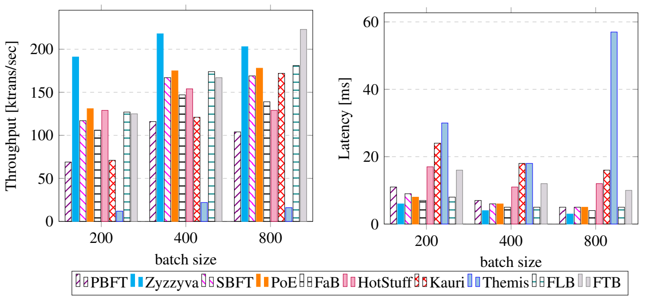


Figure 10: Impact of request batching

Supporting order-fairness (design choice **13**) leads to deficient performance of Themis compared to HotStuff (83% lower throughput when $n = 5$). In Themis, replicas need to order transactions and send batches of transactions to the leader, and the leader needs to generate a fair order. As the number of replicas increases, Themis incurs higher latency (the latency increases from 9 ms to 137 as the n increases from 5 to 101). One main source of latency is the time the leader takes to generate the dependency graph and reach a fair order. Using design choice **2** and reducing the number of communication phases results in 41% higher throughput and 46% lower latency of FTB compared to Kauri in a setting with 99 replicas (100 for Kauri).

Finally, using design choices **1** and **2**, FLB demonstrates better performance for large n (2.25x throughput and 0.55x latency compared to PBFT). This is because FLB reduces both message complexity and communication phases, and replicas do not need to wait for responses from all replicas.

Figure 7 depicts the results with different numbers of replicas. However, with the same number of replicas, different protocols tolerate different numbers of failures. For instance, PBFT requires $3f+1$ and when $n = 100$ tolerates 33 failures while FaB requires $5f+1$ and tolerates 19 failures with $n = 100$. To compare protocols based on the maximum number of tolerated failures, we represent the results of the first experiments in Figure 8. With $f = 20$, Themis incurs the highest latency because it requires 81 ($4f+1$) replicas and deals with the high cost of achieving order-fairness.

6.2 Performance with Faulty Backups

In this set of experiments, we force a backup replica to fail and repeat the first set of experiments. Figure 9 reports the results. Zyzzyva is mostly affected by failures (82% lower throughput) as clients need to collect responses from *all* replicas. A client waits for $\Delta = 5ms$ to receive reply from all replicas and then the protocol switches to its normal path.

We also run this experiment with and without faulty backups on Zyzzyva5 to validate design choice **10**, i.e., tolerating f faulty replicas by increasing the number of replicas. With a single faulty backup, Zyzzyva5 incurs only 8% lower throughput when $n = 6$.

Backup failure reduces the throughput of SBFT by 42%. In the fast path of SBFT, all replicas need to participate, and even when a single replica is faulty, the protocol falls back to its slow path, which requires two more phases. Interestingly, while the throughput of PoE is reduced by 26% in a small network (4 replicas), its throughput is not significantly affected in large networks. This is because the faulty replica (which participates in the quorum construction but does not send reply messages to the clients) has a higher chance of becoming a quorum member in small networks.

Faulty backups also affect the performance of HotStuff, especially in small networks. This is expected because HotStuff uses the rotating leader mechanism. When n is small, the faulty replica is the leader of more views during the experiments, resulting in reduced performance. HotStuff demonstrates its best performance when $n = 31$ (still, 36% lower throughput and 2.7x latency compared to the failure-free scenario). While Themis uses HotStuff as its ordering stage, a single faulty backup has less impact on its performance compared to HotStuff (25% reduction vs. 66% reduction in throughput). This is because Themis has a larger network size ($4f+1$ vs. $3f+1$) that reduces the impact of the faulty replica. In Kauri and FTB, we force a leaf replica to fail in order to avoid triggering a reconfiguration. As a result, the failure of a backup does not significantly affect their performance (e.g., 3% lower throughput with 31 replicas in Kauri). Finally, in small networks, FLB demonstrates the best performance as it incurs only 8% throughput reduction.

6.3 Impact of Request Batching

In the next set of experiments, we measure the impact of request batching. We consider three scenarios with batch sizes of 200, 400 and 800. The network includes 16 non-faulty replicas (17 replicas for Themis, 14 replicas for FLB and FTB). Figure 10 depicts the results. Increasing the batch size from 200 to 400 requests improves the performance of all protocols. This is because, with larger batch sizes, more transactions can be committed while the number of communication phases and exchanged messages is the same and the bandwidth and computing resources are not fully utilized yet. Different protocols behave differently when the batch size increases from 400 to 800. First, Kauri and FTB still process a higher number of transactions (42% and 34% higher throughput) as both protocols balance the load and utilize the bandwidth of all replicas. Second, SBFT and FaB demonstrate similar performance as before; a trade-off between smaller consensus quorums and a higher cost of signature verification and bandwidth utilization. Third, the performance of Themis decreases (24% lower

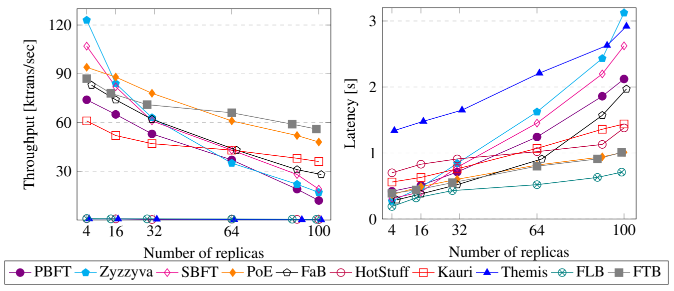


Figure 11: Performance with a geo-distributed setup

throughput and 3.16x latency) compared to a batch size of 400 due to two main reasons. First, the higher cost of signature verification and bandwidth utilization, and second, the higher complexity of generating fair order for a block of 800 transactions (CPU utilization).

6.4 Impact of a Geo-distributed Setup

We measure the performance of protocols in a wide-area network. Replicas are deployed in 4 different AWS regions, i.e., Tokyo (*TY*), Seoul (*SU*), Virginia (*VA*), and California (*CA*) with an average Round-Trip Time (RTT) of $TY \rightleftharpoons SU$: 33 ms, $TY \rightleftharpoons VA$: 148 ms, $TY \rightleftharpoons CA$: 107 ms, $SU \rightleftharpoons VA$: 175 ms, $SU \rightleftharpoons CA$: 135 ms, and $VA \rightleftharpoons CA$: 62 ms. The clients are also placed in Oregon (*OR*) with an average RTT of 97, 126, 68 and 22 ms from *TY*, *SU*, *VA* and *CA* respectively. We use a batch size of 400 and perform experiments in a failure-free situation. In this experiment, the pipelining stretch of Kauri and FTB is increased to 6. Figure 11 depicts the results.

Zyzzyva demonstrates the best performance when n is small. However, when n increases, its performance is significantly reduced (87% throughput reduction and 115x latency when n increases from 4 to 100). This is because, in Zyzzyva, clients need to receive reply messages from *all* replicas. Similarly, SBFT incurs a significant reduction in its performance due to its optimistic assumption that all replicas participate in a timely manner. In both protocols, replicas (client or leader) wait for $\Delta = 500$ ms to receive responses from all replicas before switching to the normal path. This reduction can be seen in PBFT as well (84% throughput reduction when n increases to 100) due to its quadratic communication complexity. PoE incurs a smaller throughput reduction (51%) in comparison to Zyzzyva, SBFT, and PBFT because it does not need to wait for all replicas and it has a linear communication complexity. Increasing the number of replicas does not significantly affect the throughput of FTB compared to other protocols (36% throughput reduction when n increases to 99) due to its logarithmic message complexity and pipelining.

Interestingly, HotStuff shows very low throughput. In HotStuff, the leader of the following view must wait for the previous view's decision before initiating its value. Even though Chained HotStuff is implemented in Bedrock, the leader still needs to wait for one communication round (an RTT). As a result, in contrast to the single datacenter setting where each round takes $\sim$1 ms, request batches are proposed on average every $\sim$190 ms. Similarly, in Themis and FLB, the leader must wait for certificates from $n - f$ replicas before initiating consensus on the next request batch. In Themis, network latency also affects achieving order-fairness as replicas might propose different orders for client requests. This result demonstrates the significant impact of the out-of-order processing of requests on the performance of the protocol, especially in a wide area network.

6.5 Evaluation Summary

We summarize some of the evaluation results as follows. First, optimistic protocols that require all nodes to participate, e.g., Zyzzyva and SBFT, do not perform well in large networks, especially when nodes are far apart. In small networks also, a single faulty node significantly reduces the performance of optimistic protocols. Second, the performance of pessimistic protocols highly depends on the communication topology. While the performance of protocols with quadratic communication complexity, e.g., PBFT and FaB, is significantly reduced by increasing the network size, the performance of protocols with linear complexity, e.g., HotStuff, and especially logarithmic complexity, e.g., Kauri and FTB, is less affected. Interestingly in small networks, protocols that use the leader rotation mechanism show poor performance. This is because the chance of the faulty node becoming the leader is relatively high. Third, the load-balancing techniques, e.g., tree topology, enable a protocol to process larger batches. Finally, in a wide-area network, out-of-order processing of transactions significantly improves performance.

7 Conclusion

Bedrock is a unified platform for BFT protocols analysis, implementation, and experimentation. Bedrock demonstrates how different BFT protocols relate to one another within a design space and along different design dimensions. Using a domain-specific language, the Bedrock facilitates rapid prototyping of BFT protocols. Finally, different BFT protocols proposed in diverse settings and contexts can be experimentally evaluated under one unified platform fairly and efficiently.

As future work, we plan to enable users to check the correctness of their written protocols by transforming the DSL code written in Bedrock to the specification language used by verification tools, e.g., DistAlgo [173] or TLAPS [80,93]. Moreover, to ensure that the failure of replicas is independent of each other, we plan to diversify replica implementation using n-version programming. In this way, Bedrock will be able to provide different implementations of the same protocol config (DSL code). We further plan to explore incorporating automatic selection strategies in Bedrock based on deployment environment and application requirements. Machine learning techniques may be useful here in aiding the user to select the appropriate BFT protocol, or switch one protocol to another at runtime as system parameters are updated.

We discuss related work in Appendix D.

References

[1] Chain. http://chain.com.

[2] Corda. https://github.com/corda/corda.

[3] Hyperledger iroha. https://github.com/hyperledger/iroha.

[4] libhotstuff: A general-purpose bft state machine replication library with modularity and simplicity. https://github.com/hot-stuff/libhotstuff, 2018.

[5] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *Operating Systems Review (OSR)*, 39(5):59–74, 2005.

[6] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected o(1) rounds, expected $o(n^2)$ communication, and optimal resilience. In *Int. Conf. on Financial Cryptography and Data Security*, pages 320–334. Springer, 2019.

[7] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. *arXiv preprint arXiv:1712.01367*, 2017.

[8] Ittai Abraham, Dahlia Malkhi, et al. The blockchain consensus layer and bft. *Bulletin of EATCS*, 3(123), 2017.

[9] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable byzantine consensus. In *21st Int. Conf. on Principles of Distributed Systems (OPODIS)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[10] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118. IEEE, 2020.

[11] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Brief announcement: Byzantine agreement, broadcast and state machine replication with optimal good-case latency. In *Int. Symposium on Distributed Computing (DISC)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[12] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: a complete categorization. In *Symposium on Principles of distributed computing (PODC)*, pages 331–341. ACM, 2021.

[13] Ittai Abraham, Ling Ren, and Zhuolun Xiang. Good-case and bad-case latency of unauthenticated byzantine broadcast: A complete categorization. *arXiv preprint arXiv:2109.12454*, 2021.

[14] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. {FARSITE}: Federated, available, and reliable storage for an incompletely trusted environment. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*, 2002.

[15] Marcos K Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–60, 2018.

[16] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of rdma on agreement. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 409–418, 2019.

[17] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Dalia Papuc, Athanasios Xygkis, and Igor Zablotchi. Frugal byzantine computing. In *Int. Symposium on Distributed Computing*, 2021.

[18] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. Dissecting the performance of strongly-consistent replication protocols. In *SIGMOD Int Conf on Management of Data*, pages 1696–1710, 2019.

[19] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Wpaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):211–223, 2019.

[20] Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *ACM symposium on Operating systems principles (SOSP)*, pages 45–58, 2005.

[21] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Balanced byzantine reliable broadcast with near-optimal communication and improved computation. In *Symposium on Principles of Distributed Computing (PODC)*, pages 399–417. ACM, 2022.

[22] Salem Alqahtani and Murat Demirbas. Bigbft: A multileader byzantine fault tolerance protocol for high throughput. In *Int Performance Computing and Communications Conf (IPCCC)*, pages 1–10. IEEE, 2021.

[23] Salem Alqahtani and Murat Demirbas. Bottlenecks in blockchain consensus protocols. In *Int. Conf. on Omni-Layer Intelligent Systems (COINS)*, pages 1–8. IEEE, 2021.

[24] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011.

[25] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, 7(1):80–93, 2008.

[26] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. ParBlockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 1337–1347. IEEE, 2019.

[27] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Modern large-scale data management systems after 40 years of consensus. In *Int. Conf. on Data Engineering (ICDE)*, pages 1794–1797. IEEE, 2020.

[28] Mohammad Javad Amiri, Sujaya Maiyya, Divyakant Agrawal, and Amr El Abbadi. SeeMoRe: A fault-tolerant protocol for hybrid cloud environments. In *Int. Conf. on Data Engineering (ICDE)*, pages 1345–1356. IEEE, 2020.

[29] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *European Conf. on Computer Systems (EuroSys)*, pages 30:1–30:15. ACM, 2018.

[30] Balaji Arun, Sebastiano Peluso, and Binoy Ravindran. ezbft: Decentralizing byzantine fault-tolerant state machine replication. In *Int Conf on Distributed Computing Systems (ICDCS)*, pages 565–577. IEEE, 2019.

[31] Balaji Arun and Binoy Ravindran. Scalable byzantine fault tolerance via partial decentralization. *Proc. of the VLDB Endowment*, 15(9):1739–1752, 2022.

[32] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. A fair consensus protocol for transaction ordering. In *Int. Conf. on Network Protocols (ICNP)*, pages 55–65. IEEE, 2018.

[33] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM (JACM)*, 41(1):122–152, 1994.

[34] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *Transactions on Computer Systems (TOCS)*, 32(4):12, 2015.

[35] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. Rbft: Redundant byzantine fault tolerance. In *Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 297–306. IEEE, 2013.

[36] Zeta Avarikioti, Lioba Heimbach, Roland Schmid, Laurent Vanbever, Roger Wattenhofer, and Patrick Wintermeyer. Fnf-bft: Exploring performance limits of bft protocols. *arXiv preprint arXiv:2009.02235*, 2020.

[37] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985.

[38] Amy Babay, John Schultz, Thomas Tantillo, Samuel Beckley, Eamon Jordan, Kevin Ruddell, Kevin Jordan, and Yair Amir. Deploying intrusion-tolerant scada for the power grid. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 328–335. IEEE, 2019.

[39] Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep*, 2016.

[40] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Conf. on Innovative Data Systems Research (CIDR)*, 2011.

[41] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Sok: Consensus in the age of blockchains. In *Conf. on Advances in Financial Technologies (AFT)*, pages 183–198. ACM, 2019.

[42] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 2019.

[43] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Consensus-oriented parallelization: How to earn your first million. In *Proceedings of the 16th Annual Middleware Conference*, pages 173–184, 2015.

[44] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: Sgx-based high performance bft. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 222–237, 2017.

[45] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the 2nd ACM Annual Symposium on Principles of Distributed Computing, Montreal, Quebec*, pages 27–30, 1983.

[46] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *Annual international cryptology conference*, pages 701–732. Springer, 2019.

[47] Christian Berger and Hans P Reiser. Scaling byzantine consensus: A broad analysis. In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, pages 13–18, 2018.

[48] Christian Berger, Hans P Reiser, João Sousa, and Alysson Bessani. Resilient wide-area byzantine consensus using adaptive weighted replication. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 183–18309. IEEE, 2019.

[49] Christian Berger, Sadok Ben Toumia, and Hans P Reiser. Does my bft protocol implementation scale? In *Int. Workshop on Distributed Infrastructure for the Common Good*, pages 19–24, 2022.

[50] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *Transactions on Storage (TOS)*, 9(4):12, 2013.

[51] Alysson Bessani, Joao Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[52] Alysson Neves Bessani, Paulo Sousa, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. The crutial way of critical infrastructure protection. *IEEE Security & Privacy*, 6(6):44–51, 2008.

[53] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 111–120. IEEE, 2012.

[54] Kenneth P Birman, Thomas A Joseph, Thomas Raeuchle, and Amr El Abbadi. Implementing fault-tolerant distributed objects. *Trans. on Software Engineering*, (6):502–508, 1985.

[55] Fatemeh Borran and André Schiper. A leader-free byzantine consensus algorithm. In *International Conference on Distributed Computing and Networking*, pages 67–78. Springer, 2010.

[56] Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Symposium on Principles of distributed computing*, pages 154–162, 1984.

[57] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[58] F. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. Consensus in one communication step. In *Int. Conf. on Parallel Computing Technologies (PaCT)*, pages 42–50. Springer, 2001.

[59] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[60] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. Tao: Facebook's distributed data store for the social graph. In *Annual Technical Conf. (ATC)*, pages 49–60. USENIX Association, 2013.

[61] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, August*, 1(15):14, 2016.

[62] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[63] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint arXiv:1807.04938*, 2018.

[64] Yehonatan Buchnik and Roy Friedman. Fireledger: a high throughput blockchain consensus protocol. *Proceedings of the VLDB Endowment*, 13(9):1525–1539, 2020.

[65] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[66] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual Int. Cryptology Conf.*, pages 524–541. Springer, 2001.

[67] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[68] Christian Cachin, Jovana Mićić, and Nathalie Steinhauer. Quick order fairness. In *Int. Conf. on Financial Cryptography and Data Security (FC)*, pages 1–18. Springer, 2022.

[69] Christian Cachin and Marko Vukolić. Blockchain consensus protocols in the wild. In *Int. Symposium on Distributed Computing (DISC)*, pages 1–16, 2017.

[70] Miguel Castro and Barbara Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[71] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[72] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *Symposium on Operating systems design and implementation (OSDI)*, volume 99, pages 173–186. USENIX Association, 1999.

[73] Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Conf. on Advances in Financial Technologies (AFT)*, pages 1–11. ACM, 2020.

[74] TH Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *Cryptology ePrint Archive*, 2018.

[75] TH Hubert Chan, Rafael Pass, and Elaine Shi. Pili: An extremely simple synchronous blockchain. *Cryptology ePrint Archive*, 2018.

[76] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

[77] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[78] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Pigpaxos: Devouring the communication bottlenecks in distributed consensus. In *SIGMOD Int Conf on Management of Data*, pages 235–247, 2021.

[79] JP Morgan Chase. Quorum white paper, 2016.

[80] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the tla+ proof system. In *Int. Conf., on Automated Reasoning (IJCAR)*, pages 142–148. Springer, 2010.

[81] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Operating Systems Review (OSR)*, volume 41-6, pages 189–204. ACM SIGOPS, 2007.

[82] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Symposium on Operating systems principles (SOSP)*, pages 277–290. ACM, 2009.

[83] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Symposium on Networked Systems Design and Implementation (NSDI)*, volume 9, pages 153–168. USENIX Association, 2009.

[84] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, et al. Spanner: Google's globally distributed database. *Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[85] Miguel Correia, Nuno Ferreira Neves, Lau Cheuk Lung, and Paulo Veríssimo. Low complexity byzantine-resilient consensus. *Distributed Computing*, 17(3):237–249, 2005.

[86] Miguel Correia, Nuno Ferreira Neves, Lau Cheuk Lung, and Paulo Veríssimo. Worm-it–a wormhole-based intrusion-tolerant group communication system. *Journal of Systems and Software*, 80(2):178–197, 2007.

[87] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *Int. Symposium on Reliable Distributed Systems (SRDS)*, pages 174–183. IEEE, 2004.

[88] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96, 2006.

[89] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Bft-to: Intrusion tolerance with less replicas. *The Computer Journal*, 56(6):693–715, 2013.

[90] Miguel Correia, Giuliana Santos Veronese, Nuno Ferreira Neves, and Paulo Verissimo. Byzantine consensus in asynchronous message-passing systems: a survey. *International Journal of Critical Computer-Based Systems*, 2(2):141–161, 2011.

[91] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Security Symposium*, pages 857–874. USENIX Association, 2016.

[92] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. A survey of software aging and rejuvenation studies. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 10(1):1–34, 2014.

[93] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. Tla+ proofs. In *Int Symposium on Formal Methods (FM)*, pages 147–154. Springer, 2012.

[94] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Symposium on Operating systems design and implementation (OSDI)*, pages 177–190. USENIX Association, 2006.

[95] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. Dbft: Efficient leaderless byzantine consensus and its application to blockchains. In *Int. Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2018.

[96] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red belly: a secure, fair and scalable open blockchain. In *Symposium on Security and Privacy (S&P'21)*. IEEE, 2021.

[97] George Danezis and David Hrycyszyn. Blockmania: from block dags to consensus. *arXiv preprint arXiv:1809.01620*, 2018.

[98] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *European Conference on Computer Systems (EuroSys)*, pages 34–50, 2022.

[99] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Operating Systems Review (OSR)*, volume 41, pages 205–220. ACM SIGOPS, 2007.

[100] Tobias Distler. Byzantine fault-tolerant state-machine replication from a systems perspective. *ACM Computing Surveys (CSUR)*, 54(1):1–38, 2021.

[101] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. Resource-efficient byzantine fault tolerance. *Transactions on Computers*, 65(9):2807–2819, 2016.

[102] Tobias Distler, Ivan Popov, Wolfgang Schröder-Preikschat, Hans P Reiser, and Rüdiger Kapitza. Spare: Replicas on hold. In *Network and Distributed System Security Symposium (NDSS*, 2011.

[103] Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. Powerstore: Proofs of writing for efficient and robust storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 285–298, 2013.

[104] Dan Dobre, Matthias Majuntke, Marco Serafini, and Neeraj Suri. Hp: Hybrid paxos for wans. In *European Dependable Computing Conf. (EDCC)*, pages 117–126. IEEE, 2010.

[105] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97, 1987.

[106] Sisi Duan, Michael K Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2028–2041, 2018.

[107] Sisi Duan and Haibin Zhang. Practical state machine replication with confidentiality. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 187–196. IEEE, 2016.

[108] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[109] Michael Eischer and Tobias Distler. Latency-aware leader selection for geo-replicated byzantine fault-tolerant systems. In *Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, pages 140–145. IEEE, 2018.

[110] Michael Eischer and Tobias Distler. Scalable byzantine fault-tolerant state-machine replication on heterogeneous servers. *Computing*, 101(2):97–118, 2019.

[111] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. In *SIGACT-SIGMOD symposium on Principles of database systems*, pages 215–229. ACM, 1985.

[112] Amr El Abbadi and Sam Toueg. Availability in partitioned replicated databases. In *SIGACT-SIGMOD symposium on Principles of database systems*, pages 240–251. ACM, 1986.

[113] Ian Aragon Escobar, Eduardo Alchieri, Fernando Luís Dotti, and Fernando Pedone. Boosting concurrency in parallel state machine replication. In *Int. Middleware Conf.*, pages 228–240, 2019.

[114] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[115] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*, pages 67–72. IEEE, 1997.

[116] Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *Conf. on Advances in Financial Technologies*, pages 214–228. ACM, 2019.

[117] Fangyu Gai, Ali Farahbakhsh, Jianyu Niu, Chen Feng, Ivan Beschastnikh, and Hao Duan. Dissecting the performance of chained-bft. In *Int Conf on Distributed Computing Systems (ICDCS)*, pages 595–606. IEEE, 2021.

[118] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Efficient asynchronous byzantine agreement without private setups. In *Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 246–257. IEEE, 2022.

[119] Miguel Garcia, Nuno Neves, and Alysson Bessani. An intrusion-tolerant firewall design for protecting siem systems. In *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 1–7. IEEE, 2013.

[120] Miguel Garcia, Nuno Neves, and Alysson Bessani. Sieveq: A layered bft protection system for critical services. *IEEE Transactions on Dependable and Secure Computing*, 15(3):511–525, 2016.

[121] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

[122] Neil Giridharan, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Bullshark: Dag bft protocols made practical. *arXiv preprint arXiv:2201.05677*, 2022.

[123] Garth R Goodson, Jay J Wylie, Gregory R Ganger, and Michael K Reiter. Efficient byzantine-tolerant erasure-coded storage. In *International Conference on Dependable Systems and Networks, 2004*, pages 135–144. IEEE, 2004.

[124] Gideon Greenspan. Multichain private blockchain-white paper. *URl: http://www. multichain. com/download/MultiChain-White-Paper. pdf*, 2015.

[125] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376, 2010.

[126] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable decentralized trust infrastructure for blockchains. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE/IFIP, 2019.

[127] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 803–818, 2020.

[128] Divya Gupta, Lucas Perronne, and Sara Bouchenak. Bft-bench: Towards a practical evaluation of robustness and effectiveness of bft protocols. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 115–128. Springer, 2016.

[129] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. Proof-of-execution: Reaching consensus through fault-tolerant speculation. In *Int Conf. on Extending Database Technology (EDBT)*, pages 301–312, 2021.

[130] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. Rcc: Resilient concurrent consensus for high-throughput secure transaction processing. In *Int Conf on Data Engineering (ICDE)*, pages 1392–1403. IEEE, 2021.

[131] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. Resilientdb: Global scale resilient blockchain fabric. *Proceedings of the VLDB Endowment*, 13(6):868–883, 2020.

[132] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. The case for byzantine fault detection. In *HotDep*, 2006.

[133] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

[134] James Hendricks, Gregory R Ganger, and Michael K Reiter. Low-overhead byzantine fault-tolerant storage. *ACM SIGOPS Operating Systems Review*, 41(6):73–86, 2007.

[135] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. In *20th International Conference on Principles of Distributed Systems*, 2017.

[136] Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Int. symposium on fault-tolerant computing. Digest of papers*, pages 381–390. IEEE, 1995.

[137] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.

[138] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proc. of the VLDB Endowment*, 1(2):1496–1499, 2008.

[139] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: resource-efficient byzantine fault tolerance. In *European Conf. on Computer Systems (EuroSys)*, pages 295–308. ACM, 2012.

[140] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, Mike Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Symposium on Operating systems design and implementation (OSDI)*, volume 12, pages 237–250. USENIX Association, 2012.

[141] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Symposium on Principles of Distributed Computing (PODC)*, pages 165–175. ACM, 2021.

[142] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. *The Science of Blockchain Conf. (SBC)*, 2022.

[143] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Annual Int. Cryptology Conf.*, pages 451–480. Springer, 2020.

[144] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual Int. Cryptology Conf.*, pages 357–388. Springer, 2017.

[145] Roger M. Kieckhafer and Mohammad H. Azadmanesh. Reaching approximate agreement with mixed-mode faults. *Transactions on Parallel and Distributed Systems*, 5(1):53–63, 1994.

[146] Jonathan Kirsch, Stuart Goose, Yair Amir, Dong Wei, and Paul Skare. Survivable scada via intrusion-tolerant replication. *IEEE Transactions on Smart Grid*, 5(1):60–70, 2013.

[147] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *Security Symposium*, pages 279–296. USENIX Association, 2016.

[148] Eleftherios Kokoris-Kogias. Robust and scalable consensus for sharded distributed ledgers.

[149] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[150] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. *Operating Systems Review (OSR)*, 41(6):45–58, 2007.

[151] Ramakrishna Kotla and Michael Dahlin. High throughput byzantine fault tolerance. In *International Conference on Dependable Systems and Networks*, pages 575–584. IEEE, 2004.

[152] Klaus Kursawe. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *Conf. on Advances in Financial Technologies (AFT)*, pages 25–36. ACM, 2020.

[153] Klaus Kursawe. Wendy grows up: More order fairness. In *Int. Conf. on Financial Cryptography and Data Security (FC)*, pages 191–196. Springer, 2021.

[154] Petr Kuznetsov, Andrei Tonkikh, and Yan X Zhang. Revisiting optimal resilience of fast byzantine consensus. In *Symposium on Principles of distributed computing (PODC)*, pages 343–353. ACM, 2021.

[155] Jae Kwon. Tendermint: Consensus without mining. 2014.

[156] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[157] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[158] Leslie Lamport. Generalized consensus and paxos. 2005.

[159] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[160] Leslie Lamport. Brief announcement: Leaderless byzantine paxos. In *Int. Symposium on Distributed Computing (DISC)*, pages 141–142, 2011.

[161] Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[162] Leslie Lamport and Mike Massa. Cheap paxos. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 307–314. IEEE, 2004.

[163] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[164] Butler Lampson. The abcd's of paxos. In *PODC*, volume 1, page 13. Citeseer, 2001.

[165] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *European Conference on Computer Systems (EuroSys*, pages 1–16, 2020.

[166] Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Fairledger: A fair blockchain protocol for financial institutions. In *23rd Int. Conf. on Principles of Distributed Systems (OPODIS)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2019.

[167] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI*, volume 9, pages 1–14, 2009.

[168] Bijun Li, Nico Weichbrodt, Johannes Behl, Pierre-Louis Aublin, Tobias Distler, and Rüdiger Kapitza. Troxy: Transparent access to byzantine fault-tolerant systems. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 59–70. IEEE, 2018.

[169] Bijun Li, Wenbo Xu, Muhammad Zeeshan Abid, Tobias Distler, and Rüdiger Kapitza. Sarek: Optimistic parallel ordering in byzantine fault tolerance. In *European Dependable Computing Conf. (EDCC)*, pages 77–88. IEEE, 2016.

[170] Harry C Li, Allen Clement, Amitanand S Aiyer, and Lorenzo Alvisi. The paxos register. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 114–126. IEEE, 2007.

[171] Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2007.

[172] Wenyu Li, Chenglin Feng, Lei Zhang, Hao Xu, Bin Cao, and Muhammad Ali Imran. A scalable multi-layer pbft consensus for blockchain. *Transactions on Parallel and Distributed Systems*, 32(5):1146–1160, 2020.

[173] Bo Lin and Yanhong A Liu. Distalgo: A language for distributed algorithms, 2017.

[174] Barbara Liskov and James Cowling. Viewstamped replication revisited. 2012.

[175] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. Xft: Practical fault tolerance beyond crashes. In *Symposium on Operating systems design and implementation (OSDI)*, pages 485–500. USENIX Association, 2016.

[176] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 17–30. ACM, 2016.

[177] Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computations. In *Computer Security Foundations Workshop*, pages 116–124. IEEE, 1997.

[178] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.

[179] Dahlia Malkhi and Michael K Reiter. Secure and scalable replication in phalanx. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281)*, pages 51–58. IEEE, 1998.

[180] Dahlia Malkhi and Michael K Reiter. Survivable consensus objects. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 271–279. IEEE, 1998.

[181] Yanhua Mao, Flavio P Junqueira, and Keith Marzullo. Towards low latency state machine replication for uncivil wide-area networks. In *Workshop on Hot Topics in System Dependability*. Citeseer, 2009.

[182] J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[183] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[184] Fred J. Meyer and Dhiraj K. Pradhan. Consensus with dual failure modes. *Transactions on Parallel and Distributed Systems*, (2):214–222, 1991.

[185] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 31–42, 2016.

[186] Zarko Milosevic, Martin Biely, and André Schiper. Bounded delay in byzantine-tolerant state machine replication. In *Int. Symposium on Reliable Distributed Systems (SRDS)*, pages 61–70. IEEE, 2013.

[187] Iulian Moraru, David G Andersen, and Michael Kaminsky. Egalitarian paxos. In *ACM Symposium on Operating Systems Principles*, 2012.

[188] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, 2013.

[189] Louise E Moser, Peter M Melliar-Smith, Priya Narasimhan, Lauren A Tewksbury, and Vana Kalogeraki. The eternal system: An architecture for enterprise applications. In *Int. Enterprise Distributed Object Computing Conf. (EDOC)*, pages 214–222. IEEE, 1999.

[190] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. *arXiv preprint arXiv:1909.05204*, 2019.

[191] Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. In *Int.l Symposium on Distributed Computing (DISC)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[192] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *SIGMOD Int. Conf. on Management of Data*, pages 1221–1236. ACM, 2018.

[193] Faisal Nawab and Mohammad Sadoghi. Blockplane: A global-scale byzantizing middleware. In *2019 IEEE 35th Int. Conf. on Data Engineering (ICDE)*, pages 124–135. IEEE, 2019.

[194] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 35–48, 2021.

[195] Ray Neiheiser, Daniel Presser, Luciana Rech, Manuel Bravo, Luís Rodrigues, and Miguel Correia. Fireplug: Flexible and robust n-version geo-replication of graph databases. In *Int Conf on Information Networking (ICOIN)*, pages 110–115. IEEE, 2018.

[196] Nuno Ferreira Neves, Miguel Correia, and Paulo Verissimo. Solving vector consensus with a wormhole. *IEEE Transactions on Parallel and Distributed Systems*, 16(12):1120–1131, 2005.

[197] André Nogueira, Miguel Garcia, Alysson Bessani, and Nuno Neves. On the challenges of building a bft scada. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 163–170. IEEE, 2018.

[198] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Symposium on Principles of distributed computing (PODC)*, pages 8–17. ACM, 1988.

[199] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *Annual Technical Conf. (ATC)*, pages 305–319. USENIX Association, 2014.

[200] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *31 Int. Symposium on Distributed Computing*, page 6, 2017.

[201] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Annual Int. Conf. on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2018.

[202] Fernando Pedone. Boosting system performance with optimistic distributed protocols. *Computer*, 34(12):80–86, 2001.

[203] Marco Platania, Daniel Obenshain, Thomas Tantillo, Yair Amir, and Neeraj Suri. On choosing server-or client-side solutions for bft. *ACM Computing Surveys (CSUR)*, 48(4):1–30, 2016.

[204] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *European Conf. on Computer Systems (EuroSys)*, page 8. ACM, 2015.

[205] Ji Qi, Xusheng Chen, Yunpeng Jiang, Jianyu Jiang, Tianxiang Shen, Shixiong Zhao, Sen Wang, Gong Zhang, Li Chen, Man Ho Au, et al. Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks. In *Symposium on Operating Systems Principles (SOSP)*, pages 18–34. ACM SIGOPS, 2021.

[206] Michael O Rabin. Randomized byzantine generals. In *Symposium on Foundations of Computer Science (SFCS)*, pages 403–409. IEEE, 1983.

[207] HariGovind V Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *International Conference On Principles Of Distributed Systems*, pages 88–102. Springer, 2005.

[208] Hans P Reiser and Rudiger Kapitza. Hypervisor-based efficient proactive recovery. In *Int. Symposium on Reliable Distributed Systems (SRDS)*, pages 83–92. IEEE, 2007.

[209] Robbert van Renesse, Chi Ho, and Nicolas Schiper. Byzantine chain replication. In *International Conference On Principles Of Distributed Systems*, pages 345–359. Springer, 2012.

[210] Ronald L Rivest, Adi Shamir, and Leonard M Adleman. *A method for obtaining digital signatures and public key cryptosystems*. Routledge, 2019.

[211] Tom Roeder and Fred B Schneider. Proactive obfuscation. *ACM Transactions on Computer Systems (TOCS)*, 28(2):1–54, 2010.

[212] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[213] Marco Serafini, Péter Bokor, Dan Dobre, Matthias Majuntke, and Neeraj Suri. Scrooge: Reducing the costs of fast byzantine replication in presence of unresponsive replicas. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 353–362. IEEE, 2010.

[214] Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.

[215] Nibesh Shrestha, Ittai Abraham, Ling Ren, and Kartik Nayak. On the optimality of optimistic responsiveness. In *ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 839–857, 2020.

[216] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. Bft protocols under fire. In *NSDI*, volume 8, pages 189–204, 2008.

[217] Hin-Sing Siu, Yeh-Hao Chin, and Wei-Pang Yang. A note on consensus on dual failure modes. *Transactions on Parallel and Distributed Systems*, 7(3):225–230, 1996.

[218] Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In *Int. Symposium on Distributed Computing (DISC)*, pages 438–450. Springer, 2008.

[219] João Sousa and Alysson Bessani. Separating the wheat from the chaff: An empirical design for geo-replicated state machines. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 146–155. IEEE, 2015.

[220] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, 2009.

[221] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: The partially synchronous version. *arXiv preprint arXiv:2209.05633*, 2022.

[222] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-bft: High-throughput bft for blockchains. *arXiv preprint arXiv:1906.05552*, 2019.

[223] Chrysoula Stathakopoulou, Signe Rüsch, Marcus Brandenburger, and Marko Vukolić. Adding fairness to order: Preventing front-running attacks in bft protocols using tees. In *Int Symp on Reliable Distributed Systems (SRDS)*, pages 34–45. IEEE, 2021.

[224] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up bft with acid (transactions). In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 1–17, 2021.

[225] Philip Thambidurai, You-Keun Park, et al. Interactive consistency with multiple failure modes. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 93–100. IEEE, 1988.

[226] Gene Tsudik. Message authentication with one-way hash functions. *ACM SIGCOMM Computer Communication Review*, 22(5):29–38, 1992.

[227] Robbert Van Renesse, Nicolas Schiper, and Fred B Schneider. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing*, 12(4):472–484, 2014.

[228] Paulo E Veríssimo. Travelling through wormholes: a new look at distributed systems models. *ACM SIGACT News*, 37(1):66–81, 2006.

[229] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Int. Symposium on Reliable Distributed Systems (SRDS)*, pages 135–144. IEEE, 2009.

[230] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Ebawa: Efficient byzantine agreement for wide-area networks. In *Int. Symposium on High Assurance Systems Engineering (HASE)*, pages 10–19. IEEE, 2010.

[231] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *Transactions on Computers*, 62(1):16–30, 2013.

[232] Gauthier Voron and Vincent Gramoli. Dispel: Byzantine smr with distributed pipelining. *arXiv preprint arXiv:1912.10367*, 2019.

[233] Xin Wang, Sisi Duan, James Clavin, and Haibin Zhang. Bft in blockchains: From protocols to use cases. *ACM Computing Surveys (CSUR)*, 54(10s):1–37, 2022.

[234] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated state machines with compartmentalization. *Proceedings of the VLDB Endowment*, 14(11):2203–2215, 2021.

[235] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. Zz and the art of practical bft execution. In *Conf. on Computer systems*, pages 123–138. ACM, 2011.

[236] Jiang Xiao, Shijie Zhang, Zhiwei Zhang, Bo Li, Xiaohai Dai, and Hai Jin. Nezha: Exploiting concurrency for transaction processing in dag-based blockchains. In *Int Conf on Distributed Computing Systems (ICDCS)*, pages 269–279. IEEE, 2022.

[237] David Yakira, Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, and Ronen Tamari. Helix: A fair blockchain consensus protocol resistant to ordering manipulation. *IEEE Transactions on Network and Service Management*, 18(2):1584–1597, 2021.

[238] Sravya Yandamuri, Ittai Abraham, Kartik Nayak, and Michael K Reiter. Communication-efficient bft using small trusted hardware to tolerate minority corruption. In *Int. Conf. on Principles of Distributed Systems*, 2022.

[239] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. *Operating Systems Review (OSR)*, 37(5):253–267, 2003.

[240] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Symposium on Principles of Distributed Computing (PODC)*, pages 347–356. ACM, 2019.

[241] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *SIGSAC Conf. on Computer and Communications Security*, pages 931–948. ACM, 2018.

[242] Gengrui Zhang, Fei Pan, Michael Dang'ana, Yunhao Mao, Shashank Motepalli, Shiquan Zhang, and Hans-Arno Jacobsen. Reaching consensus in the byzantine empire: A comprehensive review of bft consensus algorithms. *arXiv preprint arXiv:2204.03181*, 2022.

[243] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 633–649. USENIX Association, 2020.

[244] Lidong Zhou, Fred Schneider, Robbert VanRenesse, and Zygmunt Haas. Secure distributed on-line certification authority, August 22 2002. US Patent App. 10/001,588.

[245] Lidong Zhou, Fred B Schneider, and Robbert Van Renesse. Coca: A secure distributed online certification authority. *ACM Transactions on Computer Systems (TOCS)*, 20(4):329–368, 2002.

A Performance Optimization Dimensions

We present a set of optimization dimensions that target the performance of a BFT protocol.

O 1. Out-of-order processing. The out-of-order processing mechanism enables the leader to continuously propose new requests even when previous requests are still being processed by the backups [129]. Out-of-order processing of requests is possible if the leader does not need to include any certificate or hash of the previous request (block) in its next request.

O 2. Request pipelining. Using request pipelining, the messages of a new consensus instance are piggybacked on the second round messages of the previous instance [194, 240]. This technique is especially efficient when a protocol rotates the leader after every consensus instance.

O 3. Parallel ordering. Client requests can be ordered in parallel by relying on a set of independent ordering groups [43, 44, 169] where each group orders a subset of client requests and then all results are deterministically merged into the final order. Similarly, in multi-leader protocols [22, 30, 31, 36, 110, 130, 169, 186, 222, 232], different replicas are designated as the leader for different consensus instances in parallel and then a global order is determined.

O 4. Parallel execution. Transactions can be executed in parallel to improve the system's overall performance. One approach is to detect non-conflicting transactions and execute them in parallel [26, 113, 151]. This approach requires *a priori* knowledge of a transaction's read-set and write-set. Switching the order of agreement and execution stages and optimistically executing transactions in parallel is another approach [29, 140]. If the execution results are inconsistent (due to faulty replicas, conflicting transactions, or nondeterministic execution), replicas need to rollback their states and sequentially and deterministically re-execute the requests. switching the order of agreement and execution stages also enables replicas to detect any nondeterministic execution [29, 140].

O 5. Read-only requests processing. In pessimistic protocols, replicas can directly execute read-only requests without establishing consensus. However, since replicas may execute the read requests on different states, even non-faulty replicas might not return identical results. To resolve this, the number of required matching replies for both normal and read-only requests needs to be increased from $f+1$ to $2f+1$ in order to ensure consistency (i.e., quorum intersection requirement) [71]. This, however, results in a liveness challenge because f non-faulty replicas might be slow (or in-dark) and not receive the request. As a result, the client might not be able to collect $2f+1$ matching responses (since Byzantine replicas may not send a correct reply to the client).

O 6. Separating ordering and execution. The ordering and execution stages can be separated and implemented in different processes. This separation leads to several advantages [100] such as preventing malicious execution replicas from leaking confidential application state to clients [107, 239], enabling large requests to bypass the ordering stage [82], moving application logic to execution virtual machine [102, 208, 235] or simplifying the parallel ordering of requests [43, 46]. Moreover, while $3f+1$ replicas are needed for ordering, $2f+1$ replicas are sufficient to execute transactions [239].

O 7. Trusted hardware. Using trusted execution environments (TEEs) that prevent equivocation, e.g., Intel's SGX [183], Sanctum [91], and Keystone [165], the number of required replicas can be lowered to $2f+1$ because the trusted component prevents a faulty replica from sending conflicting messages to different replicas without being detected. A trusted component may include an entire virtualization layer [102, 208, 230], a multicast ordering service executed on a hardened Linux kernel [86, 87], a centralized configuration service [209], a trusted log [81], an append-only log [238], a trusted platform module, e.g., counter [230, 231], a smart card TrInc [167], or an FPGA [101, 139].

O 8. Request/reply dissemination. A client can either multicast its request to *all replicas* [51,94,229] where each replica relays the request to the leader or optimistically send its request to a *contact replica*, typically the leader. The contact replica is known to the client through a reply to an earlier request [72, 150]. If the client timer for the request (τ_1) expires, the client multicasts its request to all replicas. This optimistic mechanism requires fewer messages to be sent from clients to the replicas. However, this comes at the cost of increased network traffic between replicas, because the leader needs to disseminate the full request to other replicas to enable them to eventually execute it.

On the other hand, all replicas can send the results to clients in their reply messages. This, however, leads to significant network overhead for large results. A protocol can optimistically rely on a designated responder replica (chosen by the client or servers) to send the full results. Other replicas then either send the hash of the results to the client or send a signed message to the responder enabling the responder to generate a proof for the results, e.g., SBFT [126]. While this technique reduces network overhead, the client might not receive the results if the responder replica is faulty, the network is unreliable, or the responder replica was in-dark and skipped the execution and applied a checkpoint to catch up [100].

B Case Studies on Protocol Evolution

In this section, we provide insights into how each BFT protocol, mentioned in Figure 4 and Figure 5, maps into the Bedrock design space and relates to one another through using design choices. For illustrative purposes, we describe each protocol relative to PBFT, along with one or more design choices.

Zyzzyva [150]. Zyzzyva[3] (Figure 12) can be derived from PBFT using the speculative execution function (design choice

[3]The view-change stage of the Zyzzyva protocol has a safety violation as described in [7]

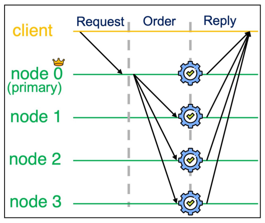


Figure 12: Zyzzyva

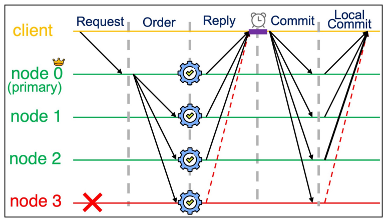


Figure 13: Zyzzyva (slow)

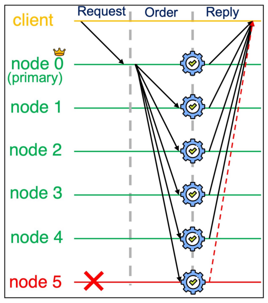


Figure 14: Zyzzyva5

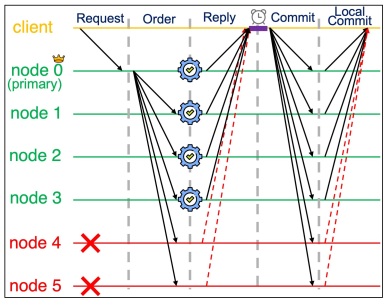


Figure 15: Zyzzyva5 (slow)

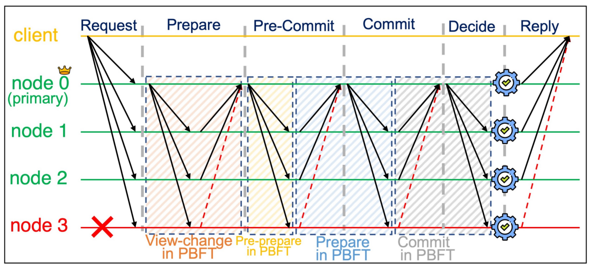


Figure 16: HotStuff

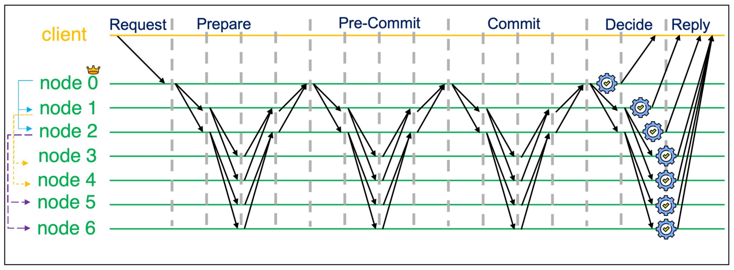


Figure 17: Kauri

8) of Bedrock where assuming the leader and all backups are non-faulty, replicas speculatively execute requests without running any agreement and send reply messages to the client. The client waits for $3f+1$ matching replies to accept the results. If the timer τ_1 is expired and the client received matching replies from between $2f+1$ and $3f$ replicas, as presented in Figure 13, two more linear rounds of communication are needed to ensure that at least $2f+1$ replicas have committed the request. Finally, Zyzzyva5 is derived from Zyzzyva by using the resilience function (design choice **10**) where the number of replicas is increased to $5f+1$ and the protocol is able to tolerate f and $2f$ failures during its fast and slow path respectively (presented in (Figures 14 and 15) AZyzzyva [34, 125] also uses the fast path of Zyzzyva (called ZLight) in its fault-free situations.

PoE [129]. PoE Figure 18 uses the linearization and speculative phase reduction functions (design choices **1** and **7**). PoE does not assume that all replicas are non-faulty and constructs a quorum of $2f+1$ replicas possibly including Byzantine replicas. However, since a client waits for $2f+1$ matching reply messages, all $2f+1$ replicas constructing the quorum need to be well-behaving to guarantee client liveness in the fast path.

SBFT [126]. Bedrock derives SBFT[4] from PBFT using the linearization and optimistic phase reduction functions (design choices **1** and **6**). SBFT presents an optimistic fast path (Figure 19), assuming all replicas are non-faulty. If the leader does not receive messages from *all* backups (in the prepare phase) and its timer is expired (i.e., non-responsiveness timer τ_3), SBFT switches to its slow path (Figure 20) and requires two more linear rounds of communication (commit phase). The Twin-path nature of SBFT requires replicas to sign each message with two schemes (i.e., $2f+1$ and $3f+1$). To send replies to the client, a single (collector) replica receives replies from all replicas and sends a single (threshold) signed reply message.

HotStuff [240]. HotStuff (Figure 16) can be derived from PBFT using the linearization and leader rotation functions (design choices **1** and **3**) of Bedrock. The Chained HotStuff (performance optimization **2**) benefits from pipelining to reduce the latency of request processing.

Tendermint [62, 63, 155]. Tendermint[5] leverages the non-responsive leader rotation function (design choice **4**) to rotate leaders without adding any new phases. The new leader, however, needs to wait for a predefined time (timer τ_4), i.e., the worst-case time it takes to propagate messages over a wide-area peer-to-peer gossip network, before proposing a new block. Tendermint also uses timers in all phases where a replica discards the request if it does not receive $2f+1$ messages before the timeout (timer τ_6). Note that the original Tendermint uses a gossip all-to-all mechanism and has $O(n \log n)$ message complexity.

Themis [142]. Themis is derived from HotStuff using the fair function (design choice **13**). Themis add a new all-to-all preordering phase where replicas send a batch of requests in

[4]SBFT tolerates both crash and Byzantine failure ($n = 3f+2c+1$ where c is the number of crashed replicas). Since the focus of this paper is on Byzantine failures, we consider a variation of SBFT where $c = 0$.

[5]Tendermint uses a Proof-of-Stake variation of PBFT where each replica has a voting power equal to its stake (i.e., locked coins).

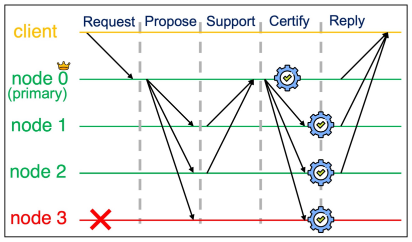


Figure 18: PoE

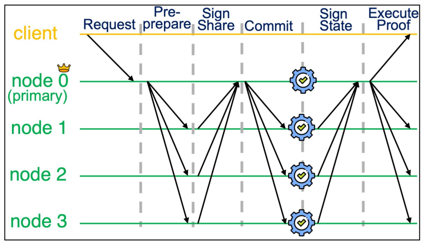


Figure 19: SBFT

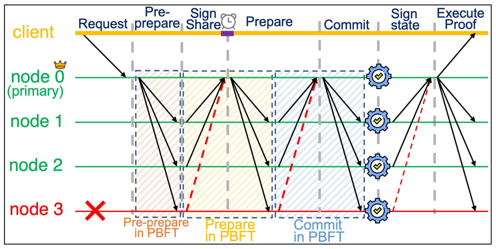


Figure 20: SBFT slow path (Linear PBFT)

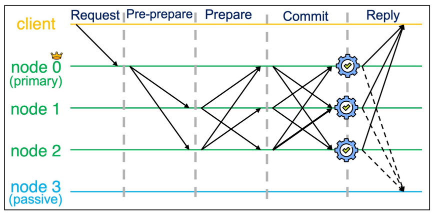


Figure 21: CheapBFT

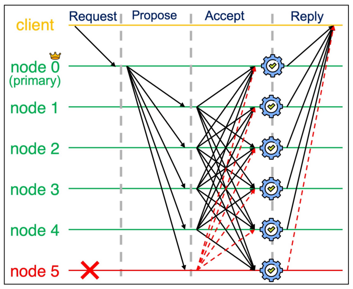


Figure 22: FaB

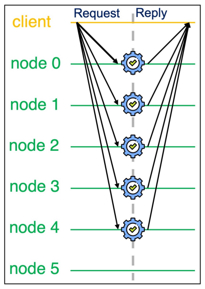


Figure 23: Q/U

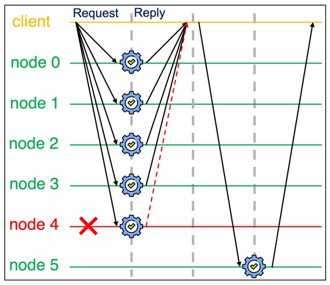


Figure 24: Q/U slow path

the order they received to the leader replica and the leader proposes requests in the order received (depending on the order-fairness parameter γ) [143]. Themis requires at least $4f+1$ replicas (if $\gamma=1$) to provide order fairness.

Kauri [194]. Kauri (Figure 17) can be derived from HotStuff using the loadbalancer function (design choice **14**) that maps the star topology to the tree topology. The height of the tree is $h=\log_d n$ where d is the fanout of each replica.

CheapBFT [139]. CheapBFT (Figure 21) and its revised version, REBFT [101] is derived from PBFT using the optimistic replica reduction function (design choice **5**). Using trusted hardware (performance optimization O7), a variation of REBFT, called RWMINBFT, processes requests with $f+1$ active and f passive replicas in its normal case (optimistic) execution.

FaB [182]. FaB[6] (Figure 22) uses the phase reduction function (design choice **2**) to reduce one phase of communication while requiring $5f+1$ replicas. Fab does not use authentication in its ordering stage, however, requires signatures for the view-change stage (design choice **11**). Note that using authentication, $5f-1$ replicas is sufficient to reduce one phase of communication [12, 154].

Prime [24]. Prime is derived from PBFT using the robust functions (design choice **12**). In prime, a preordering stage is added where replicas exchange the requests they receive from clients and periodically share a vector of all received requests, which they expect the leader to order requests following those vectors. In this way, replicas can also monitor the leader to order requests in a fair manner.

[6]FaB, similar to the family of Paxos-like protocols, separates proposers from acceptors. In our implementation of FaB, however, replicas act as both proposers and acceptors.

Q/U [5]. Q/U (Figure 23) utilizes optimistic conflict-free and resilience functions (design choices **9** and **10**). Clients play the proposer role and replicas immediately execute an update request if the object has not been modified since the client's last query. Since Q/U is able to tolerate f faulty replicas, a client can optionally communicate with a subset $(4f+1)$ of replicas (preferred quorum). The client communicates with additional replicas only if it does not receive reply from all replicas of the preferred quorum (Figure 24). Both signatures (for large n) and MACs (for small n) can be used for authentication in Q/U. Quorum [34] uses a similar technique with $3f+1$ replicas, i.e., only the conflict-free function (design choices **9**) has been used.

C Discovering New Protocol Using Bedrock

Bedrock provides a systematic way to explore new valid points in the design space and help BFT researchers uncover *novel* BFT protocols. We uncover several such new protocols, although not all are necessarily practical or interesting. For example, simply making a protocol fair by adding the preordering phase of fairness results in a new protocol. While this is an interesting insight, the resulting protocol may have limited practical impact. We select as highlights two new BFT protocols (FLB and FTB) that are new and have practical value that we have uncovered using Bedrock.

Fast Linear BFT (FLB). FLB (Figure 25) is a fast linear BFT protocol that commits transactions in two phases of communication with linear message complexity. To achieve this, FLB uses the linearization and phase reduction through redundancy functions (design choices **1** and **2**). FLB requires $5f-1$ replicas (following the lower bound results on fast Byzantine agreement [12, 154]). The ordering stage of FLB is similar to the fast path of SBFT in terms of the linearity

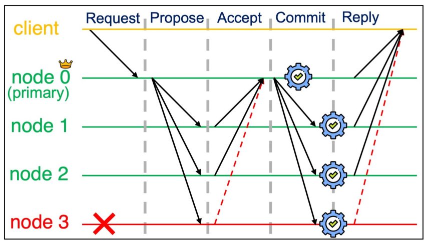


Figure 25: FLB

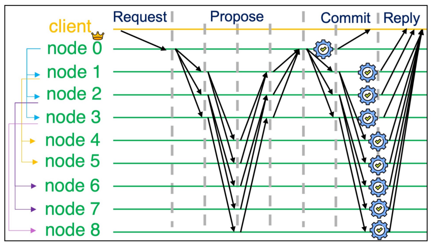


Figure 26: FTB

of communication and the number of phases. However, FLB expands the network size to tolerate f failures (in contrast to SBFT, which optimistically assumes all replicas are non-faulty).

Fast Tree-based balanced BFT (FTB). A performance bottleneck of consensus protocols is the computing and bandwidth capacity of the leader. While Kauri [194] leverages a tree communication topology (design choice **14**) to distribute the load among all replicas, Kauri requires $7h$ phases of communication to commit each request, where h is the height of the communication tree.

FTB (Figure 26) reduces the latency of Kauri based on two observations. First, we noticed that while Kauri is implemented on top of HotStuff, it does not use the leader rotation mechanism. As a result, it does not need the two linear phases of HotStuff ($2h$ phases in Kauri) that are added for the purpose of leader rotation (design choice **3**). Second, similar to FLB, we can use the phase reduction through redundancy function (design choice **2**) to further reduce $2h$ more phases of communication. FTB establishes agreement with $5f-1$ replicas in $3h$ phases. FTB also uses the pipelining stretch mechanism of Kauri, where the leader continuously initiates consensus instances before receiving a response from its child nodes for the first instance (similar to the out-of-order processing used by many BFT protocols).

D Related Work

SMR regulates the deterministic execution of requests on multiple replicas, such that every non-faulty replica executes every request in the same order [156,212]. Several approaches [157,199,212] generalize SMR to support crash failures. CFT protocols [19,58,76,78,104,135,135,137,158,159,161,162,164,170,174,192,198,199,202,227] utilize the design trade-offs between design dimensions, e.g., Fast Paxos [159] adds f replicas to reduce a communication phase.

Byzantine fault tolerance refers to nodes that behave arbitrarily after the seminal work by Lamport, et al. [163]. BFT protocols have been analyzed in several surveys and empirical studies [7,8,23,27,41,47,49,51,69,90,100,117,128,203,216,233,242]. We discuss some of the more relevant studies.

Berger and Reiser [47] present a survey on BFT protocols used in blockchains where the focus is on scalability techniques. Similarly, a survey on BFT protocols consisting of classical protocols, e.g., PBFT, blockchain protocols, e.g., PoW, and hybrid protocols, e.g., OmniLedger [149], and their applications in permissionless blockchains, is conducted by Bano et al. [41]. Platania et al. [203] classify BFT protocols into client-side and server-side protocols depending on the client's role. The paper compares these two classes of protocols and analyzes their performance and correctness attacks. Three families of leader-based, leaderless, and robust BFT protocols with a focus on message and time complexities have been analyzed by Zhang et al. [242]. Finally, Distler [100] analyzes BFT protocols along several main dimensions: architecture, clients, agreement, execution, checkpoint, and recovery. The paper shares several dimensions with Bedrock.

A recent line of work [10–13] also study good-case latency of BFT protocols. Bedrock, in contrast to all these survey and analysis papers, provides a design space, systematically discusses design choices (trade-offs), and, more importantly, provides a tool to analyze BFT protocols experimentally.

BFTSim [216] is a simulation environment for BFT protocols that leverages a declarative networking system. The paper also compares a set of representative protocols using the simulator. Abstract [34] presents a framework to design and reconfigure BFT protocols where each protocol is developed as a sequence of BFT instances. Abstract presents AZyzzyva, Aliph, and R-Aliph as three BFT protocols. Each protocol itself is a composition of Abstract instances presented to handle different situations (e.g., fault-free, under attack). In contrast to such studies, Bedrock develops a unified design space for BFT protocols, enabling end-users to choose a protocol that best fits their application requirements.

In addition to CFT and BFT protocols, consensus with multiple failure modes has also been studied for both synchronous [145,184,217,225], and partial synchronous [28,82,126,175,204,213] models. Finally, leaderless protocols [55,95,106,127,160,185,224] have been proposed to avoid the implications of relying on a leader.

E PBFT DSL Specification in Bedrock

This section shows the implementation of PBFT using the DSL defined by Bedrock. Listing 1 demonstrates the code. The specification has two main parts: the protocol and the plugins used by the protocol. The plugins, as discussed earlier, are categorized into four groups: role, message, transition, and pipeline. For each category, several plugins have been implemented in Bedrock that can be used by different protocols.

```
plugins:
  role: primary
  message:
    - digest
    - mac
    - checkpoint
  transition:
    - checkpoint
  pipeline: direct

protocol:
  general:
    leader: stable
    requestTarget: primary

  roles:
    - primary
    - nodes
    - client

  phases:
    - name: normal
      states:
        - idle
        - wait_prepare
        - wait_commit
        - executed
      messages:
        - name: request
          requestBlock: true
        - name: reply
          requestBlock: true
        - name: preprepare
          requestBlock: true
        - prepare
        - commit
    - name: view_change
      states:
        - wait_view_change
        - wait_new_view
      messages:
        - view_change
        - new_view
    - name: checkpoint
      messages:
        - checkpoint

  transitions:
    from:
      - role: client
        state: idle
        to:
          - state: executed
            update: sequence
            condition:
              type: msg
              message: reply
              quorum: 2f + 1

      - role: primary
        state: idle
        to:
          - state: wait_prepare
            condition:
              type: msg
              message: request
              quorum: 1
            response:
              - target: nodes
                message: preprepare
            extra_tally:
              - role: primary
                message: prepare

      - role: nodes
        state: idle
        to:
          - state: wait_prepare
            condition:
              type: msg
              message: preprepare
              quorum: 1
            response:
              - target: nodes
                message: prepare
            extraTally:
              - role: primary
                message: prepare

      - role: nodes
        state: wait_prepare
        to:
          - state: wait_commit
            condition:
              type: msg
              message: prepare
              quorum: 2f + 1
            response:
              - target: nodes
                message: commit
          - state: wait_view_change
            update: view
            condition:
              type: timeout
              mode: sequence
            response:
              - target: nodes
                message: view_change

      - role: nodes
        state: wait_commit
        to:
          - state: executed
            update: sequence
            condition:
              type: msg
              message: commit
              quorum: 2f + 1
            response:
              - target: client
                message: reply

// view-change
      - role: nodes
        state: wait_view_change
        to:
          - state: wait_new_view
            condition:
              type: msg
              message: view_change
              quorum: 2f + 1
      - role: primary
        state: wait_new_view
        to:
          - state: executed
            update: sequence
            response:
              - target: nodes
                message: new_view
              - target: client
                message: reply
      - role: nodes
        state: wait_new_view
        to:
          - state: executed
            update: sequence
            condition:
              type: msg
              message: new_view
              quorum: 1
            response:
              - target: client
                message: reply
          - state: wait_new_view
            update: view
            condition:
              type: timeout
              mode: stat
```

Listing 1: The DSL specification of PBFT Protcol

Users can also define their plugins or update existing ones. For example, the pipeline of messages could be `direct`, as it is used in most protocols including PBFT, or `chained` as it is used in chained-HotStuff [4] or Kauri [194] where messages of consecutive requests are pipelined. The implementation of Digests, MACs, and checkpointing is presented as plugins to enable developers to update them quickly and reuse them in multiple protocols.

The protocol code defines roles, phases, transitions, and the view-change routine. Each phase itself consists of different states, e.g., `idle`, `wait-prepare`, `wait-commit`, and `executed`, and messages, e.g., `request`, `reply`, `preprepare`, `prepare`, and `commit`. The transitions between different states and the condition for each transition are specified in transitions. For example, `node` goes from `idle` state to `wait-prepare` by receiving a `single` `preprepare` message and in response to this event, `node` sends a `prepare` message to all other `nodes` (as shown in listing 1, lines 75-85).