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

Mohammad Javad Amiri
Stony Brook University

Chenyuan Wu
University of Pennsylvania

Divyakant Agrawal
UC Santa Barbara

Amr El Abbadi
UC Santa Barbara

Boon Thau Loo
University of Pennsylvania

Mohammad Sadoghi
UC 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 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 [43, 57, 63, 87, 102, 145, 197]. While cloud systems, e.g., Google’s Spanner [87], Amazon’s Dynamo [102], and Facebook’s Tao [63], rely on crash fault-tolerant protocols, e.g., Paxos [164], 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, 26, 29, 30, 32, 45, 67, 82, 127–129, 137, 162, 213, 220, 223], permissionless blockchains [64, 154, 156, 183, 253], distributed file systems [14, 75, 85], locking service [86], firewalls [55, 122, 123, 219, 230, 251], certificate authority systems [257], SCADA systems [41, 153, 205, 256], key-value datastores [53, 106, 126, 140, 219], and key management [187].

BFT protocols use the State Machine Replication (SMR) technique [163, 221] 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 find the protocol that best fits their needs.

This paper presents Bedrock, a unified platform that enables us to analyze, implement, and experimentally evaluate 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 organize protocols in an ordered fashion and provide new insights into the properties of existing BFT protocols.

- **BFT protocols implementation.** Within Bedrock, a wide range of BFT protocols, e.g., PBFT [73], SBFT [131], HotStuff [252], Kauri [202], Themis [149], Tendermint [66], Prime [24], PoE [135], CheapBFT [146], Q/U [5], FaB [190], and Zyzzyva [157], 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 \[\text{[163, 221]}\] 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 \[\text{[226]}\]. In a BFT SMR protocol, all non-faulty replicas execute the same requests in the same order \(\text{(safety)}\) and all correct requests are eventually executed \(\text{(liveness)}\). In an asynchronous system, where replicas can fail, no consensus solutions guarantee both safety and liveness \(\text{(FLP result)}\) \[\text{[117]}\]. As a result, asynchronous consensus protocols rely on techniques such as randomization \[\text{[48, 70, 121, 214]}\], failure detectors \[\text{[80, 185]}\], hybridization/wormholes \[\text{[88, 204]}\] and partial synchrony \[\text{[108, 111]}\] to circumvent the FLP impossibility.

Bedrock assumes the partial synchrony model as it is used in most practical BFT protocols \[\text{[73, 131, 157, 252]}\]. In the partial synchrony model, there exists an unknown global stabilization time \(\text{(GST)}\), after which all messages between correct replicas are received within some known bound \(\Delta\). 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 \(\text{(e.g., n-version programming)}\) \[\text{[40, 118]}\] or placing replicas at different geographic locations \(\text{(e.g., datacenters)}\) \[\text{[51, 112, 229, 241]}\]. Finally, a strong adversary can coordinate malicious replicas and delay communication. However, the adversary cannot subvert cryptographic assumptions.

Usage model. Bedrock aims to help users analyze, implement, and evaluate BFT protocols within one unified platform and find the protocol that fits their needs. 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 users 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 \(\text{(among more than 10 dimensions, as described in Section 3)}\), i.e., number of replicas and number of commitment phases. Each dimension, e.g., number of replicas, can take different values, e.g., \(3f + 1, 5f + 1, \text{or} 7f + 1\). A BFT protocol is then a point in this design space, e.g., \(\text{("3", "3f + 1")}\). 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 \(\text{(○)}\) specify valid points \(\text{(i.e., BFT protocols)}\) while red dots \(\text{(●)}\) show invalid points \(\text{(i.e., impossible protocols)}\). For example, there is no \(\text{(pessimistic)}\) BFT protocol with \(3f + 1\) nodes that commit re-

![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.](image-url)
requests in a single commitment phase. A design choice 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 communication phases, e.g., PBFT [73], to a protocol with \(5f+1\) nodes and 2 communication phases, e.g., FaB [190] (assuming both protocols are pessimistic with clique topology). The domain and range of each design choice are a subset of 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 [73, 75] as a driving example. PBFT, as shown in Figure 3, is a leader-based protocol that operates in a succession of configurations called views [114, 115]. 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 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 and can be recovered after (leader) failures. 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 \(ID = v + 1 \mod n\)) 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 [73] or MACs [75] 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, 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 [157].
- $a_2$. The backups are non-faulty and actively and honestly participate in the protocol, e.g., CheapBFT [146].
- $a_3$. All non-leaf replicas in a tree topology are non-faulty, e.g., Kauri [202].
- $a_4$. 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 [37], and
- $a_6$. The network is synchronous (in a time window), and messages are not lost or delayed, e.g., Tendermint [65].

Optimistic protocols are classified into speculative and non-speculative protocols. In non-speculative protocols, e.g., SBFT [131] and CheapBFT [146], replicas execute a transaction only if the optimistic assumption holds. Speculative protocols, e.g., Zyzzyva [157] and PoE [135], 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 [131], or clients, e.g., Zyzzyva [157], 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 [86], R-Aliph [37], Spinning [240] and RBFT [38], 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 [73, 131, 157, 190] replaces the leader when the leader is suspected to be faulty by other replicas. In the rotating leader mechanism [20, 67, 76–78, 86, 125, 139, 155, 162, 240, 241, 252], 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 [46, 47, 240], improving resilience against slow replicas [86], and minimizing communication delays between clients and the leader [112, 189, 241].

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 [73, 103, 135]. Checkpointing is typically initiated after a fixed window in a decentralized manner without relying on a leader [73].

P 5. Recovery. When there are more than $f$ failures, BFT protocols, apart from some exceptions [84, 178], completely fail and do not give any guarantees on their behavior [103]. BFT protocols perform recovery using reactive or proactive mechanisms (or a combination [230]). Reactive recovery mechanisms detect faulty replica behavior [138] and recover the replica by applying software rejuvenation techniques [95, 142] 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 [103]. 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 [230]. To prevent attackers from disrupting the recovery process, each replica requires a trusted component, e.g., secure coprocessor [75], a synchronous wormhole [239] or a virtualization layer [105, 216], 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 [103].

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 [73], \( 2f + 1 \) in PoE [135] and PBFT (for read-only requests) [75], or \( 3f + 1 \) is Zyzzyva [157]. Using trusted components, e.g., Troxy [175], or threshold signatures, e.g., SBFT [131], 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, 130, 186, 188]. Repairer clients, on the other hand, detect the failure of replicas, e.g., Zyzzyva [157], or even change the protocol configuration, e.g., Scrooge [222], Abstract [37], 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 [59, 60, 91, 111, 170]. Using trusted hardware, the malicious behavior of replicas is restricted and safety can be guaranteed using \( 2f + 1 \) replicas [84, 90, 92, 216, 241, 241, 242]. 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 \( 5f + 1 \) [190] (its proven lower bound, \( 5f − 1 \) [12, 161]) or \( 7f + 1 \) [228]. 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 [104, 146]. Using both trusted hardware and active/passive replication, the quorum size is further reduced to \( f + 1 \) during failure-free situations [104, 105, 146].

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 [157, 252], (2) the clique topology where all (or a subset of) replicas communicate directly with each other resulting in quadratic message complexity [73], (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 [154, 155, 202], or (4) the chain topology where replicas construct a pipeline and each replica communicates with its neighbor replicas [37].

E 3. Authentication. Participants authenticate their messages to enable other replicas to verify a message’s origin. Bedrock support both signatures, e.g., RSA [218], and authenticators [73], i.e., MACs [237]. Constant-sized threshold signatures [70, 224] 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 [36, 208, 209, 225]. 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 [162] and Casper [68]. 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 [131] and Zyzzyva [157].

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 [62, 198, 199]. 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 [73], or assigning a distinct synchronizer component, e.g., Pacemaker in HotStuff [252], 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.

\( \tau_1 \). Waiting for reply messages, e.g., Zyzzyva [157],

\( \tau_2 \). Triggering (consecutive) view-change, e.g., PBFT [73],

\( \tau_3 \). Detecting backup failures, e.g., SBFT [131],

\( \tau_4 \). Quorum construction in an ordering phase, e.g., prevote
and precommit timeouts in Tendermint [65],
τ₅. View synchronization, e.g., Tendermint [65],
τ₆. Finishing a (preordering) round, e.g., Themis [149],
τ₇. Performance check (heartbeat), e.g., Aardvark [86], and
τ₈. Atomic recovery (watchdog timer) to periodically hand
control to a recovery monitor [74], e.g., PBFT [75].

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 [42, 71, 149, 150, 159, 160, 255]. Order-fairness is defined as: “if a large num-
ber of replicas receives a request τ₁ before another request
τ₂, then τ₁ should be ordered before τ₂” [150]. 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 [86], (2) adding a preordering
phase, e.g., Prime [24], where replicas order the received
requests locally and share their orderings with each other,
(3) encrypting requests and revealing the contents only once
their ordering is fixed [35, 69, 193, 232], (4) reputation-based
systems [35, 99, 156, 173] to detect unfair censorship of spe-
cific client requests, and (5) providing opportunities for every
replica to propose and commit its requests using fair elec-
tration [9, 35, 151, 173, 208, 248].

Q 2. Load balancing. The performance of fault-tolerant pro-
tocols is usually limited by the computing and bandwidth
capacity of the leader [18, 21, 56, 81, 195, 196, 202, 245]. 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 ro-
tating leader mechanism, multi-layer, or multi-leader proto-
cols. Using leader rotation, one replica (leader) is still highly
loaded in each consensus instance. In multi-layer protocols
[25, 137, 179, 201, 203], the load is distributed between the
leaders of different clusters. However, the system still suffers
from load imbalance between the leader and backups in each
cluster. In multi-leader protocols [22, 33, 39, 136, 233, 243], all
replicas can initiate consensus to partially order requests in
parallel. However, slow replicas still affect the global ordering
of requests. To resolve the bandwidth limit of small replicas
decomposing consensus into data availability agreement and
block retrieval is proposed [250].

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 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 [131],
HotStuff [252] and HotStuff-2 [184]. The output protocol re-
quires (threshold) signatures for authentication. The collector
collects a quorum of (typically n − f) signatures from repli-
casts and broadcasts its message including the signatures, as
a certificate of having received the required signatures. Using
threshold signatures [69, 70, 215, 224] the collector message
size becomes constant. Some BFT protocols [124, 143, 235]
use linear communication during the ordering phase but fol-
low the quadratic view-change routine of PBFT.

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 [190]. 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, 161].
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, 161].

Design Choice 3. (Leader rotation). This function replaces
the stable leader with the rotating leader mechanism, e.g.,
HotStuff [252], 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 rotat-
ing leader mechanism without adding a new ordering phase (in contrast to design choice 3) while sacrificing responsiv-
ess. The new leader assumes that the network is synchronous
(after GST) and waits for a predefined known upper bound Δ
(Timer τ₈) before initiating the next request. This is needed
to ensure that the new leader is aware of the highest assigned or-
der to the requests, e.g., Tendermint [66, 162] and Casper [68].
As an optimization, if the new leader is aware of the highest assigned order (the leader was part of the quorum), it can initiate the next request right after receiving $2f + 1$ votes (without necessarily waiting for $\Delta$ [184]).

**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, a2). 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 [104, 146]. 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 phase) assuming all replicas are non-faulty, e.g., SBFT [131]. 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 $\tau_3$), 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 [135]. 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, a1 and a2), e.g., Zyzzyva [157]. 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 $\tau_1$) or it receives conflicting messages, the (re-pairer) 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, a4), 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, 97].

**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 [157], 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 $7f + 1$ [228].

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 [230].

**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 [252], $k$ 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 [240] or isolating the incoming traffic of different replicas, and checking the performance of the leader in Aardvark [86].

**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 $\tau_6$), 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.
Table 1: Comparing selected BFT protocols based on different dimensions of Bedrock design space

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PBFT [73]</td>
<td>3/4+</td>
<td>clique</td>
<td>MAC &amp; Sign</td>
<td>τ₂, τ₃, τ₄</td>
<td>pessimistic</td>
<td>3</td>
<td>stable</td>
<td>pro.</td>
<td>Req.</td>
<td></td>
<td></td>
<td>(11)</td>
</tr>
<tr>
<td>Zyzzyva [157]</td>
<td>3/4+</td>
<td>star</td>
<td>MAC &amp; Sign</td>
<td>τ₁, τ₁</td>
<td>optimistic (spec): a₁, a₂</td>
<td>1 (3) stable</td>
<td>-</td>
<td>Req.</td>
<td></td>
<td></td>
<td></td>
<td>8, (11)</td>
</tr>
<tr>
<td>Zyzzyva5 [157]</td>
<td>5/4+</td>
<td>star</td>
<td>MAC &amp; Sign</td>
<td>τ₁, τ₂</td>
<td>optimistic (spec): a₁</td>
<td>1 (3) stable</td>
<td>-</td>
<td>Req.</td>
<td></td>
<td></td>
<td></td>
<td>8, (11)</td>
</tr>
<tr>
<td>PoE [135]</td>
<td>3/4+</td>
<td>star</td>
<td>MAC &amp; T-Sign</td>
<td>τ₁, τ₂</td>
<td>optimistic (spec): a₂</td>
<td>3</td>
<td>stable</td>
<td>- Req.</td>
<td></td>
<td></td>
<td></td>
<td>1, 7, 11</td>
</tr>
<tr>
<td>SBFT [131]</td>
<td>3/4+</td>
<td>star</td>
<td>T-Sign</td>
<td>τ₁, τ₂, τ₃</td>
<td>optimistic: a₂</td>
<td>3 (5) stable</td>
<td>-</td>
<td>Req.</td>
<td></td>
<td></td>
<td></td>
<td>1, 6, 11</td>
</tr>
<tr>
<td>HotStuff [252]</td>
<td>3/4+</td>
<td>star</td>
<td>T-Sign</td>
<td>τ₁, τ₂</td>
<td>pessimistic</td>
<td>7</td>
<td>rotating</td>
<td>- Req.</td>
<td></td>
<td></td>
<td></td>
<td>1, 3, 11</td>
</tr>
<tr>
<td>T tendermint [66]</td>
<td>3/4+</td>
<td>clique</td>
<td>Sign</td>
<td>τ₁, τ₂, τ₃, τ₄</td>
<td>optimistic: a₂</td>
<td>3</td>
<td>rotating</td>
<td>- Req.</td>
<td></td>
<td></td>
<td></td>
<td>4, 11</td>
</tr>
<tr>
<td>Themis [149]</td>
<td>4/4+</td>
<td>star</td>
<td>T-Sign</td>
<td>τ₁, τ₂, τ₃</td>
<td>pessimistic</td>
<td>1+7</td>
<td>rotating</td>
<td>- Req.</td>
<td></td>
<td></td>
<td></td>
<td>1, 3, 11</td>
</tr>
<tr>
<td>Kauri [202]</td>
<td>3/4+</td>
<td>tree</td>
<td>T-Sign</td>
<td>τ₁, τ₂</td>
<td>pessimistic</td>
<td>7h</td>
<td>stable*</td>
<td>- Req.</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>CheapBFT [146]</td>
<td>2/4+</td>
<td>clique</td>
<td>MAC</td>
<td>τ₁, τ₂</td>
<td>optimistic: a₂</td>
<td>3</td>
<td>stable</td>
<td>- Req.</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Prime [24]</td>
<td>3/4+</td>
<td>clique</td>
<td>Sign</td>
<td>τ₁, τ₂, τ₃, τ₄</td>
<td>pessimistic</td>
<td>8</td>
<td>stable</td>
<td>- Req.</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>QU [5]</td>
<td>5/4+</td>
<td>star</td>
<td>MAC</td>
<td>τ₁, τ₂</td>
<td>optimistic: a₁, a₃</td>
<td>1 (3) stable</td>
<td>-</td>
<td>Req.</td>
<td></td>
<td></td>
<td></td>
<td>9, 10</td>
</tr>
<tr>
<td>FLB</td>
<td>5/4-</td>
<td>clique</td>
<td>Sign</td>
<td>τ₁, τ₂</td>
<td>pessimistic</td>
<td>2</td>
<td>stable</td>
<td>- Req.</td>
<td></td>
<td></td>
<td></td>
<td>1, 2, 11</td>
</tr>
<tr>
<td>FTB</td>
<td>5/4-</td>
<td>tree</td>
<td>T-Sign</td>
<td>τ₁, τ₂</td>
<td>pessimistic</td>
<td>3h</td>
<td>stable</td>
<td>- Req.</td>
<td></td>
<td></td>
<td></td>
<td>2, 14, 11</td>
</tr>
</tbody>
</table>

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. Depending on the order-fairness parameter $\gamma (0.5 < \gamma \leq 1)$ that defines the fraction of replicas receiving the requests in that specific order, at least $4f+1$ replicas ($n > \frac{4f}{2f-1}$) replicas are needed to provide order-fairness [149, 150].

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 [202]. 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 P₁, a₃). 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 $o₁, o₂$, and $o₃$), (3) the execution stage, (4) the view-change stages consisting of view-change and new-view phases (labeled by $v₁$ and $v₂$), 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 [42, 100, 101, 119, 148, 231, 247] have emerged 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 [136, 233], 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 Bedrock 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,

![Figure 6: Lines of code in Bedrock and the original implementation 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 lines². Overall, using Bedrock, the code size is reduced by orders of magnitude. Each protocol, in addition to the config file, 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 [157] or handling view-change without using a different process or states in Tendermint [162]. 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 [146] 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., Chained-HotStuff [252] (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 pro-

² We count only the lines of source code related to the core consensus protocol and not the applications or the utilized libraries.
cess and setups all entities by initializing replicas and clients, sending config parameters to executors, enabling plugins to run additional initialization steps, starting and stopping execution 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 under one unified platform. We use 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 (presented in Appendix D).

All protocols listed in Table 1 are implemented in Bedrock. Note that the original implementations of such BFT protocols utilize different (often old, inefficient) libraries, crypto algorithms, etc. Hence, it was unfair to experimentally compare such original implementations with their implementations 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, 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 $\gamma$ ofThemis 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). In Bedrock, Chained-HotStuff has been implemented using the pipelining technique, resulting in lower average latency. 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
with the high cost of achieving order-fairness. Supporting order-fairness (design choice 13) leads to deficient performance of Themis compared to HotStuff (83% lower throughput when \( n = 5 \)). In Themis, replicas 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 to 137 ms as \( n \) increases to 101), mainly due to the overhead of generating the dependency graph and reaching a fair order by the leader. 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 (4\( f+1 \)) 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 \( \Delta = 5\text{ms} \) to receive a reply from all replicas and then the protocol switches to its normal path.

We also run this experiment 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 (4\( f+1 \) vs. 3\( f+1 \)) 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 throughput).
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 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 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 [163,221]. Several approaches [164,207,221] generalize SMR to support crash failures. CFT protocols [19,61,79,81,107,141,144,165,166,168,169, 171,177,181,200,206,207,210,238] utilize the design trade-offs between design dimensions, e.g., Fast Paxos [166] 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. [170]. BFT protocols have been analyzed in several surveys and empirical studies [7,8,23,28,44,50,52,54,72,93,103,120,133,134, 211,226,244,254]. We discuss some of the relevant studies.

Berger and Reiser [50] 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 [156], and their applications in permissionless blockchains, is conducted by Bano et al. [44]. Platania et al. [211] 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. [254]. Finally, Distler [103] 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 [226] is a simulation environment for BFT protocols that leverages a declarative networking system and compares a set of representative protocols using the simulator. Abstract [37] develops each protocol as a sequence of BFT instances, e.g., Azyzzyva, Aliph, and R-Aliph as three protocols where 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 design space for BFT protocols, enabling end-users to analyze, implement, and evaluate different protocols.

In addition to CFT and BFT protocols, consensus with multiple failure modes has also been studied for both synchronous [152,192,227,236], and partial synchronous [31,85,131,182,212,222] models. Finally, leaderless protocols [58,98,109,132,167,193,234] have been proposed to avoid the implications of relying on a leader.

8 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 tools such as DistAlgo [180] or TLAPS [83,96]. Moreover, to ensure the independent failure of replicas, we plan to diversify replica implementation using n-version programming, where Bedrock provides different implementations of the same protocol config. We will further design a constraint checker to automatically find all plausible points (valid combinations of design choices) in the design space based on user queries. Incorporating automatic selection strategies in Bedrock based on the deployment environment and application requirements could be the next step. Machine learning techniques may be useful here in aiding the user in selecting the appropriate BFT protocol, or switch one protocol to another at runtime as system parameters are updated. Finally, we plan to extend the supported protocols to include synchronous and fully asynchronous protocols and expand the design space accordingly.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback and suggestions. This work is funded by NSF grants CNS-2104882, CNS-2107147, and CNS-2245373.
References


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


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 [135]. 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 [202, 252]. 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 [46, 47, 176] 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, 33, 34, 39, 113, 136, 176, 194, 233, 243], 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 [27, 116, 158]. 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 [32, 147]. 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 [32, 147].

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) [75]. 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 [103] such as preventing malicious execution replicas from leaking confidential application state to clients [110, 251], enabling large requests to bypass the ordering stage [85], moving application logic to execution virtual machine [105, 216, 246] or simplifying the parallel ordering of requests [46, 49]. Moreover, while \( 3f + 1 \) replicas are needed for ordering, \( 2f + 1 \) replicas are sufficient to execute transactions [251].

O 7. Trusted hardware. Using trusted execution environments (TEEs) that prevent equivocation, e.g., Intel’s SGX [191], Sanctum [94], and Keystone [172], 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 [105, 216, 241], a multicast ordering service executed on a hardened Linux kernel [89, 90], a centralized configuration service [217], a trusted log [84], an append-only log [249], a trusted platform module, e.g., counter [241, 242], a smart card TrInc [174], or an FPGA [104, 146]. The current version of Bedrock does not support trusted hardware.

O 8. Request/reply dissemination. A client can either multicast its request to all replicas [54, 97, 240] 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 [73, 157]. If the client timer for the request (\( \tau_1 \)) 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 [131]. 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 [103].

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 [157]. Zyzzyva\(^3\) (Figure 11) can be derived from PBFT using the speculative execution function (design choice 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 \(\tau_1\) is expired and the client received matching replies from between \(2f+1\) and \(3f\) replicas, as presented in Figure 12, two more linear rounds of communication are needed to ensure that at least \(2f+1\) replicas have committed the request. Finally, Zyzzyva\(^5\) 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 13 and 14).

AZyzzyva [37, 130] also uses the fast path of Zyzzyva (called ZLight) in its fault-free situations.

PoE [135]. PoE Figure 17 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 [131]. 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 18), 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 \(\tau_3\)), SBFT switches to its slow path (Figure 19) 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 [252]. HotStuff (Figure 15) can be derived from PBFT using the linearization and leader rotation functions (design choices 1 and 3) of Bedrock. Chained-HotStuff (performance optimization 2) benefits from pipelining to reduce the latency of request processing.

Tendermint [65, 66, 162]. 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 \(\tau_4\)), 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 \(\tau_6\)). Note that the original Tendermint uses a gossip all-to-all mechanism and has \(O(n\log n)\) message complexity.

Byzantine failures, we consider a variation of SBFT where \(c = 0\).

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

\(^4\)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 linearization and optimistic phase reduction functions (design choices 1 and 6), SBFT presents an optimistic fast path (Figure 18), 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 \(\tau_3\)), SBFT switches to its slow path (Figure 19) 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.

\(^5\)Tendermint uses a Proof-of-Stake variation of PBFT where each replica has a voting power equal to its stake (i.e., locked coins).
Themis [149]. 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 the order they received to the leader replica and the leader proposes requests in the order received (depending on the order-fairness parameter $\gamma$) [150]. Themis requires at least $4f+1$ replicas (if $\gamma = 1$) to provide order fairness.

Kauri [202]. Kauri (Figure 16) 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 [146]. CheapBFT (Figure 20) and its revised version, ReBFT [104] 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 [190]. FaB* (Figure 21) 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, 161].

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.

Q/U [5]. Q/U (Figure 22) 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 23). Both signatures (for large $n$) and MACs (for small $n$) can be used for authentication in Q/U. Quorum [37] 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 pre-ordering 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 24) 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

---

*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.
5f − 1 replicas (following the lower bound results on fast Byzantine agreement [12, 161]). The ordering stage of FLB is similar to the fast path of SBFT in terms of the linearity 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 [202] 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 25) 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 Impact of a Geo-distributed Setup

In this part, 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 = SU: 33 ms, TY = VA: 148 ms, TY = CA: 107 ms, SU = VA: 175 ms, SU = CA: 135 ms, and VA = 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 26 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 Δ = 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 ~1 ms, request batches are proposed on average every ~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.
E  PBFT DSL Specification in Bedrock

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: prepare
        requestBlock: true
      - prepare
      - commit
      - name: view_change
      states:
        - wait_view_change
        - wait_new_view
      messages:
        - name: checkpoint
        - new_view
        - view_change
        - 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
      response:
      - target: nodes
        message: prepare
      extraTally:
      - role: primary
        message: prepare
  - role: nodes
    state: idle
to:
  - state: wait_prepare
    condition:
      type: msg
      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
      node: sequence
      response:
      - target: nodes
        message: view_change
  - role: nodes
    state: wait_commit
to:
  - state: executed
    update: view
    condition:
      type: timeout
      node: sequence
      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
      response:
      - target: client
        message: reply
  - 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: wait_new_view
    condition:
      type: executed
      update: sequence
      response:
      - target: nodes
        message: new_view
      - 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
      response:
      - target: client
        message: reply
Listing 1: The DSL specification of PBFT Protocol

// view_change
- role: nodes
  state: wait_view_change
to:
  - state: wait_new_view
    condition:
      type: msg
      message: view_change
      quorum: 2f + 1
      response:
      - target: client
        message: reply
  - 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: wait_new_view
    condition:
      type: timeout
      node: stat
  - role: primary
    state: wait_new_view
to:
  - state: executed
    update: sequence
    condition:
      type: msg
      message: new_view
      quorum: 1
      response:
      - target: client
        message: reply
  - role: primary
    state: wait_new_view
to:
  - state: wait_new_view
    condition:
      type: timeout
      node: stat
  - role: primary
    state: wait_new_view
to:
  - state: executed
    update: sequence
    condition:
      type: msg
      message: new_view
      quorum: 1
      response:
      - target: client
        message: reply
Listing 1: The DSL specification of PBFT Protocol
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.

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 [202] 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). The state manager enables the core unit to track the states and possible transitions of each entity according to the protocol.